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 01609-2280
{maged|lingw|lisading|rundenst}@cs.wpi.edu

ABSTRACT

Modern data sources, including structural and semi-structural sources, often export XML views over base data, and at times may materialize their views by storing the XML query result to provide faster data access. It is typically more efficient to maintain a view by incrementally propagating the base 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. We briefly discuss our XQuery view maintenance system implementation. Our experiments confirm that incremental view maintenance is indeed faster than re-computation.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Management, Algorithms, Performance

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. Views are frequently materialized to speed up querying [4], 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 [6]. 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 [7, 8].

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 [2, 9]. However view maintenance of XML views has received little attention so far. 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.

Recently, [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 instead is based on the XQuery language, a full-featured SQL-style XML query language that has become a World Wide Web Consortium (W3C) recommendation. [10] describes an extension of the XQuery language to support XML updating, and implements these operations using relational technology. In [3], an efficient maintenance technique for materialized views over web data was proposed, using XQL, thus limited to views in XPath expressions.

While many systems employ XQuery views [2, 9, 13], to the best of our knowledge, no solution as of now handles maintenance of materialized XQuery views. In this paper we propose the first solution for incremental XQuery view maintenance. We have designed an algebraic solution approach based on the XAT XML algebra [13]. In our approach, an update to the XML source is transformed into a set of well defined update primitives to be propagated trough the algebra tree. Our work makes the following contributions: (1) Proposal of a set of update primitives for updating XML documents. (2) General strategy for XML view maintenance, based on an algebraic approach for incremental update rewriting. (3) Implementation of the view maintainer as extension of Rainbow system [13]. (4) Preliminary results confirming that our update propagator is indeed more efficient than re-computation.

2. BACKGROUND

XML Fundamentals. Figures 1 and 2 show our running example of two XML documents on books and reviews. 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 published by "Morgan Kaufman Publishers" and their respective reviews. A *materialized XML view* refers to the stored derived XML data. Figure 4 shows the materialized XML

*This work was supported in part by the NSF NYI grant IIS-979624 and the NSF grant IIS-9988776.

WIDM'02, November 4–9, 2002, McLean, Virginia, USA.

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

```
<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.

XML Algebra XAT. We translate our view queries from the XQuery format into algebra expressions based on the XML algebra called XAT [13]. XAT has a set of well-defined algebra operators called XAT operators. The input and output of each XAT operator are XAT tables.

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 contains a sequence of tuples. Each tuple t_j is a vector of cells of type: (1) atomic value; (2) node including XML Element, XML Document, or XML Attribute; or (3) ordered collection of any mixture of (1) and (2). Figure 5 shows the XAT algebra tree for our example view query in Figure 3. The semantics of our query from Figure 5 can best be understood by viewing the intermediate XAT table results that would be produced by each operator during execution for our running example, as depicted in Figure 7 [1]. For details about the XAT algebra the reader is referred to [13].

[1] Please ignore the update-propagation specific parts in Figure 7, 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 [12], where unused columns are removed for optimization purposes.

```
<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.

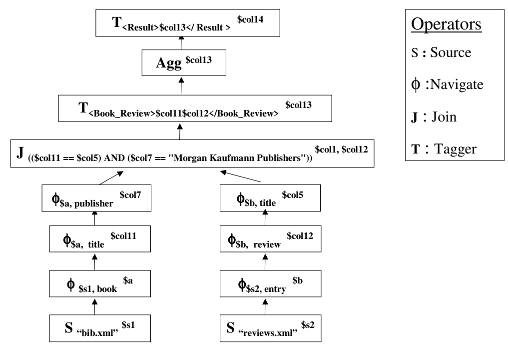

Figure 5: XAT algebra tree of the view query in Figure 3.

3. 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 [5]. 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. 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 helps to keep track of different fragments of the XML document spread over the XAT tables.

3.2 Update Primitives

Updates to an XML source document can be transformed into a sequence of primitive update operations called $Update\ Primitives$. We define two sets of update primitives *up*. Below the parameter *pos* corresponds to an XPath expression used to access elements.

Update Primitives for Updating XML Documents (xup): They 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): Intermediate results in the XAT algebra tree correspond to XAT tables, i.e., they contain collections of fragments. Hence we need to have a way to specify the location of the XML fragment to be updated in 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

The overall update propagation strategy for our system is shown in Figure 6. Each node in the XAT algebra tree has knowledge about the XAT operator associated with it, its input XAT tables, output XAT table and the auxiliary information defined in Section 3.1. The update propagation starts from the bottom of the algebra tree. We assume that the operator associated with the leaf node (generally, the Source operator) receives one update to the source XML document at a time. The Source operator accepts an update of any of the six basic update types *xup*s defined above, and propagates such update into a sequence of zero or more update primitives of type *iup* to its parent operator. All other operators in the algebra tree receive as input sequences XAT-specific update primitives *iup*s from their children nodes, rewrite them into sequences of new update primitives *iup*s and propagate them upwards. The update primitives are 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. We have rules for update propagation for each of the algebra operator (operatorPropagate) [5], these rules are not discussed here due to space limitation.

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

Figure 6: View maintenance algorithm.

3.4 Update Propagation Example

Given our example view query tree depicted in Figure 7, assume 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 "Morgan Kaufmann Publishers". Then we expect a new $Book_{Review}$ element to be inserted into the result view.

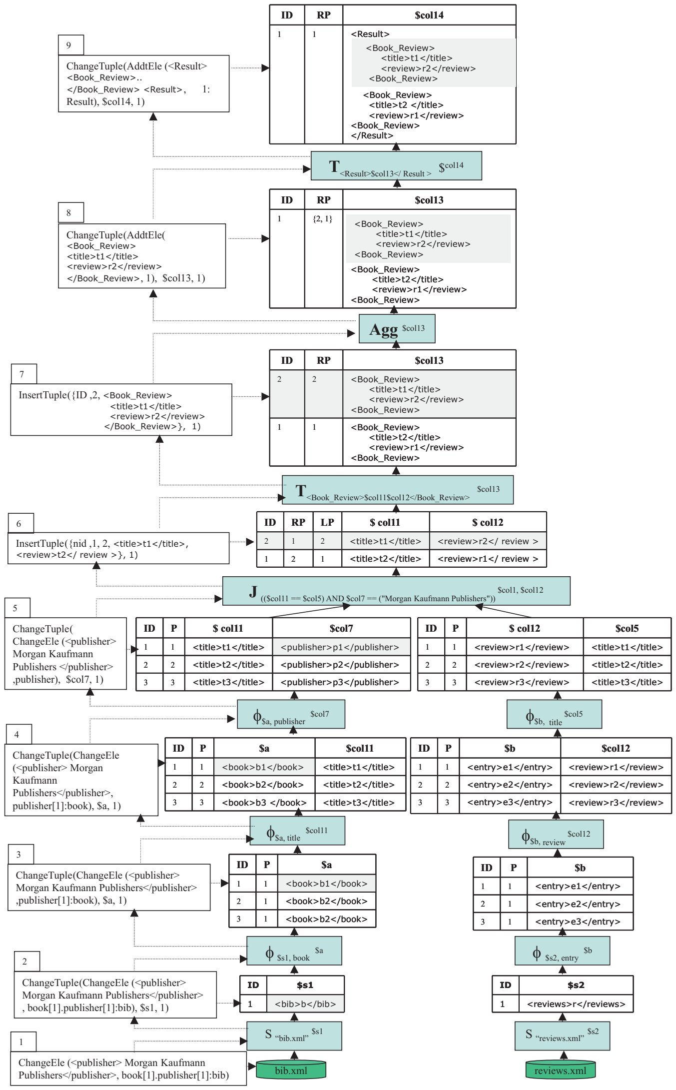

Figure 7: Propagation example.

At the bottom of Figure 7, update (1) is applied to the input XML document bib.xml, changing the publisher element in the specified XPath to the new publisher element. The update is then passed to the Source operator $S^{\$s1}_{bib.xml}$, which generates update (2) by encapsulating the input XML update primitive *up* (1) into a ChangeTuple update with *up*, *$s1* and 1 as its *xup*, *ucol* and *id* argument respectively. Update (2) is propagated to the parent operator $\phi^{\$a}_{\$s1,book}$ which generates a new update primitive (3). Update (3) has a new *col* argument *$a* which is derived from the *col'* argument in the navigate operator itself, and a new *pos* that is relative to the extracted element book in the new column *$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 [5].

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 *pos = Null:publisher* (or publisher in short) since the element to be updated is at the root in the update column. Next Join 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. Update (6) is next 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).

Finally update (8) is passed to the second Tagger node that generates update (9) by applying the tagging pattern to the newly added element and modifying *pos* to be *1:Result*, i.e., referring to the first XML fragment branches from the Result element. The final effect on the materialized view will be the insertion of a $Book\_Review$ element at position 1.

4. EXPERIMENTAL RESULTS

The view maintainer has been built as an extension of the Rainbow query engine [11]. The latter generates the XAT algebra tree 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 for each algebra node propagates updates.
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 test system was a Intel(R) Celeron(TM) 733MHz processor, 128M memory, running Windows2000 and Java 1.3.0_01.

Figure 8 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. Other experiments (not shown here) confirm similar trends for other conditions and settings. We thus conclude that incremental maintenance is a valuable technique for XML view maintenance compared to re-computation.

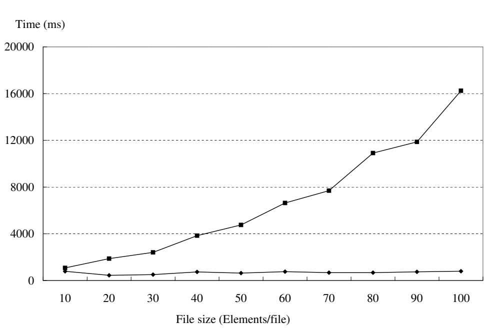

Figure 8: Performance for different file sizes.

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. Thereafter we designed rules for each algebra node to propagate updates through the XAT algebra tree up to the result view. The preliminary results confirm that incremental view maintenance is indeed faster than re-computation in most cases.

Acknowledgements. We thank Xin Zhang, Brad Pielech, Brian Murphy, and all Rainbow team members at WPI for providing us with the core Rainbow system.

6. 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] 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.

[3] L. Chen and E. A. Rundensteiner. Aggregate Path Index for Incremental Web View Maintenance. In *WECWIS*, pages 231–238, June 2000.

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

[5] M. EL-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. An Algebraic Approach for Incremental Maintenance of Materialized XQuery. Technical report, Computer Science Department, WPI, 2002. in progress.

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

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

[8] 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.

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

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

[11] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and E. A. Rundensteiner. Rainbow: Mapping-Driven Xquery Processing System. In *Demo Session Proceedings of SIGMOD*, page 614, 2002.

[12] X. Zhang, B. Pielech, , and E. Rundensteiner. Honey, I Shrunk the Xquery!- An XML Algebra Optimization Approach. In *WIDM*, 2002 to appear.

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