Aggregation and Accumulation of XML Data*

Kristin Tufte[1,2], David Maier[2]
[1]University of Wisconsin-Madison, [2]Oregon Graduate Institute
tufte, maier@cse.ogi.edu

Abstract

XML is rapidly becoming a standard for data exchange on the Internet. While XML's document management roots have led many to focus on querying and processing of large documents, we believe much XML data will be in the form of streams. One can envision streams of XML data flowing throughout the Internet: a stream of stock quotes or minute-by-minute updates on positions of a fleet of vehicles - one XML fragment per vehicle report. In this paper, we propose a new operation, Merge, which provides the capability to create aggregates over streams of data and the ability to take XML documents from different inputs and piece them together to create a new XML document. The Merge operation effectively handles highly-nested, semi-structured data and was designed to be used in an environment where there are long-running queries and stream-based data sources. We describe a flexible mechanism, called a Merge Template, which we have developed to specify how to merge two XML documents.

1 Introduction

XML is fundamentally changing the nature of data on the Internet. A prominent feature of this change is the appearance of data streams. Data no longer lives in local files that can be read and re-read at the discretion of the DBMS. In fact, some XML data exists only in streams on the Internet which must be processed as they arrive. Examples of this kind of data include stock quotes and B2B and B2C messages. In this paper, we present a new operation, Merge, which provides a step towards effectively processing XML data streams.

The Merge operation combines two XML documents into a result document. One application of Merge is accumulating and aggregating data from an XML stream. Consider a stream of stock quotes from the NYSE expressed in XML; each quote contains a ticker symbol and trade price. Merge can be used to create and maintain an XML document containing today's current, high, low and average prices for all stocks traded on the NYSE. This evolving document is called an Accumulator and it can be queried like any XML document stored in a DBMS. Figure 1 shows a graphical representation of such an Accumulator. Figure 2 shows a stock quote for IBM that has just arrived on the NYSE stock ticker (stream) and Figure 3 shows the result when this quote is Merged into the Accumulator. Note the functions that the Merge operation performs. The first time a ticker symbol (i.e. IBM) appears in the quote stream, Merge inserts a Stock element for IBM into the Accumulator.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF ITR award IIS0086002.

This new element contains IBM's ticker symbol and elements for IBM's low, high, average and current stock prices, the values of which are all set to the initial trade price. As additional trades occur on IBM stock, the updates arrive as small XML documents and are merged into the existing element for IBM: The current stock price is replaced with the trade price and the aggregate values low, high and average are updated appropriately.

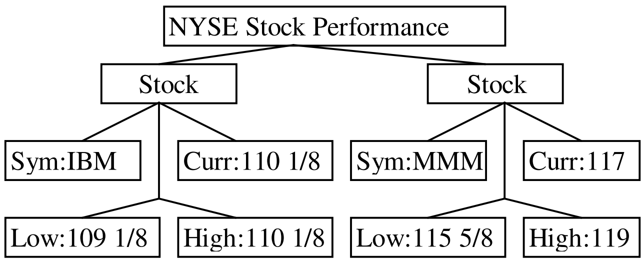

Figure 1: NYSE Stock Quote Accumulator

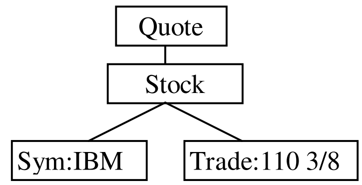

Figure 2: Stock Quote for IBM

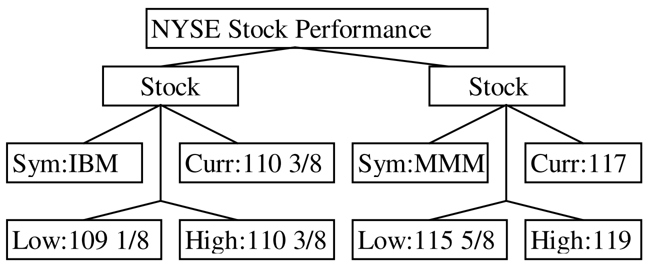

Figure 3: Accumulator after Merge of Stock Quote

A new mechanism, called a Merge Template, specifies what actions should occur when combining two XML documents. Merge Templates are very flexible and provide for operations including content update, aggregation and sub-tree replacement. A Merge Template also incorporates a simple form of equality predicates, called Match Templates, to allow a user to specify when two elements represent the same entity.

In addition to aggregating streaming XML, Merge is useful for replication, cache maintenance, and incremental query processing. Merge is specifically designed to work on irregular and incomplete data and can function without a DTD. A single Merge Template can handle a range of input structures. Finally, a merge-based accumulate operator has been implemented in Version 1.0 of the Niagara Query Engine [8]. (See the Niagara paper in this volume for an overview of the system.)

The rest of this paper is organized as follows. Section 2 describes the Merge operation and Merge and Match Templates, Section 3 discusses related work, and Section 4 concludes and discusses future work.

2 The Merge Operation and Merge Templates

Formally, the Merge operation merges two XML documents into a single result document. In this section, we use a simple example to explain the operation of Merge and to illustrate the structure and function of Merge Templates. In short, a Merge Template specifies the structure of the merge result, predicates to apply and operations to perform, and can itself be represented in XML. For simplicity, we ignore XML attributes in this section; extending Merge Templates to handle attributes is straightforward.

2.1 Merge Templates

Consider an XML document that consists of information about classes offered in the Computer Science Department at the University of Wisconsin. Figure 4 shows a graphical representation of such a document. Each box in the figure represents an XML element and the tag names and content (PCDATA) of the elements are written as TagName:Content. Figure 5 shows a diagram of a small XML document representing the addition of a student to a class. The Merge operator can be used to merge the Add Record (Figure 5) into the Class List (Figure 4) to produce the document shown in Figure 6. The Merge process is driven by a Merge Template.

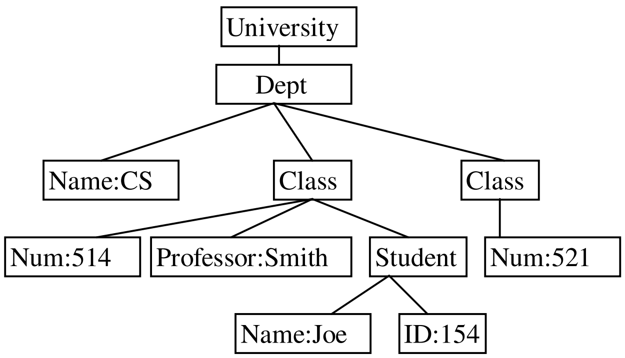

Figure 4: Class List

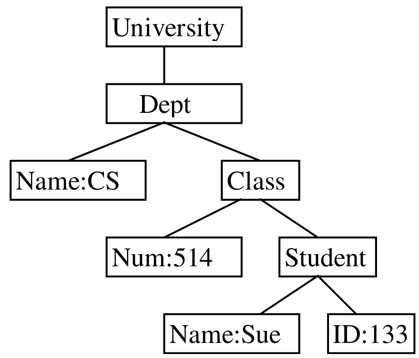

Figure 5: Add Record

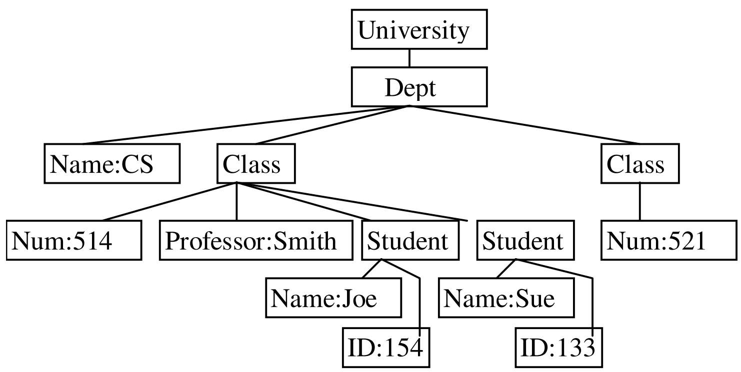

Figure 6: Merged Document

Merge Templates are a general mechanism for specifying how to merge two XML documents together. Figure 7 shows a logical representation of the Merge Template for the Class List example; Figure 8 shows the Merge Template in XML. Conceptually, a Merge Template is a tree of Element Merge Templates. Each Element Merge Template (EMT) specifies how to produce a (possibly empty) set of elements for the result document. For example, the Merge Template for the Class List example contains a Class EMT that specifies how to create the Class elements in the result from the Class elements in the input documents. Thus, each EMT is logically associated with a set of elements in the result document. In Figure 7, the name of each EMT is the name of the result element(s) to be produced using that EMT. In general, this is not a requirement.

Each EMT consists of three parts: an optional Match Template, a required Content Combine Function and a possibly empty set of sub-EMTs. The Match Template is a predicate that indicates when two elements should be deemed equivalent. The Content Combine Function specifies how to create the content of the result element(s) associated with an EMT. The set of sub-EMTs specifies how to produce the result elements' children. Consider the EMT for Class elements. The Match Template, represented by "Match on Value of Num," indicates that that two Class elements match if they have the same number. The Content Combine Function, represented as "Empty," indicates that Class elements in the result should not have any content. Finally, the Class EMT contains a sub-EMT for each type of sub-element (Num, Professor, Student) that a Class may have. The structure of the Merge Template parallels the structure of the result document of the merge. Notice how the "tree" of gray EMTs in Figure 7 is similar to the structure of the Merged Document in Figure 6.

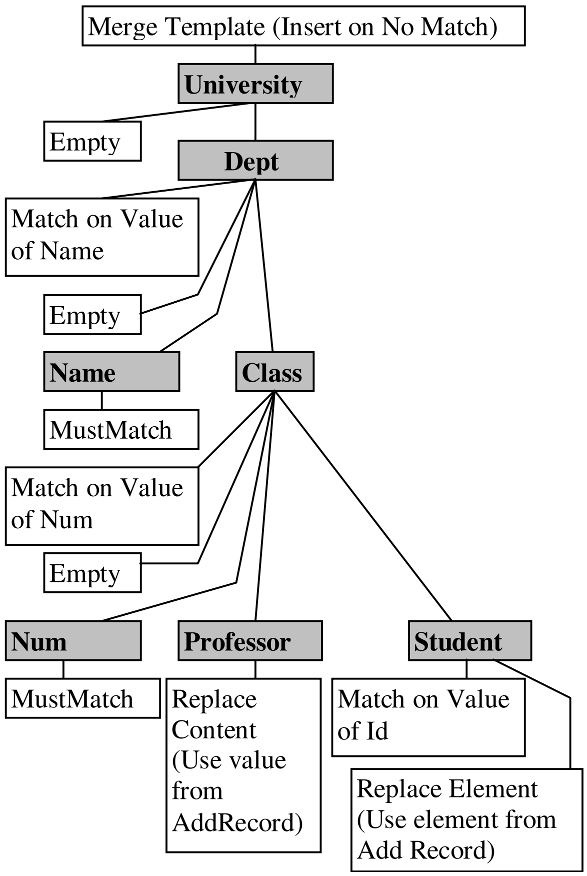

Figure 7: Logical Representation of Merge Template

```
<?xml version="1.0"?>
<!DOCTYPE DocMergeTemplate
                              MergeTemplate.dtd>
<DocMergeTemplate MergeType="outer">
  <EltMergeTemplate, Name = "University">
   <Empty/>

    <EltMergeTemplate, Name = "Dept">
    <MatchTemplate>...</>
    <Empty/>

       <EltMergeTemplate, Name = "Name">
         <MustMatch/> </>

       <EltMergeTemplate, Name = "Class">
       <MatchTemplate>...</>
       <Empty/>

         <EltMergeTemplate, Name = "Num">
         <MustMatch/> </>

         <EltMergeTemplate, Name = "Professor">
         <ShallowContent Function =  "replace"
                        DominantSide ="right" /> </>

         <EltMergeTemplate, Name = "Student">
         <DeepReplace DominantSide="right"/> </>
</> </> </>
```

Figure 8: Class List Merge Template in XML

A Merge Template incorporates a form of equality predicates, called Match Templates. Figure 9 shows the Match Template for the Class element. In general, a Match Template contains a list of paths. Each path can reference an element or sub-element value or attribute value. Two elements match if the values referenced by all paths are equal. Merge allows matching on values found through traversal of IDREFs, but does not support merging through IDREFs. We next discuss the Merge operation.

```
<MatchTemplate>
   <MatchNode>
      <Path> Num </>
      <Content ValueType = "String"/>
   </>
</>
```

Figure 9: Match Template

2.2 Merge Operation

The Merge operation works by traversing the Merge Template and the input documents (Class List and Add Record in our example) in parallel and combining the input documents on an element-by-element basis. For each EMT in the Merge Template, a possibly empty set of elements is produced for the result document.

The process begins by using the root EMT in the Merge Template to produce the content of the root element of the result document, by combining the content of the root elements of the input documents. The sub-elements of the result root element are produced using the sub-EMTs of the root EMT. For each EMT encountered, the Merge operation locates two sets of elements - one from each input document (the elements in each set are siblings). These sets are joined based on the Match Template in the EMT. The join is an inner, outer, left-outer or right-outer join as is specified by the Merge Type of the Merge Template and produces a relation containing pairs of elements. We currently require that this be a one-to-one relation, with the exception of the nulls generated by the outer joins. For each pair in the relation, an element is produced for the result document. The content of the result element is produced by combining the content of the two elements in the pair based on the Content Combine Function in the EMT. The possible combine functions include replace content, various aggregates and replace element. Merge Templates support a limited version of typing so mathematical functions such as aggregates can be calculated properly. The sub-elements for the result element are produced by scanning the sub-EMTs of the current EMT and processing them as just described. Very loosely, Merge is a series of nested joins - with potentially one join for each "type" of element in the result document. This proliferation of joins may seem excessive; however, a join is executed only when an EMT produces a set of elements for the result.

Merge is designed to handle a range of structures in its inputs, so it can be used with streams of varied structure and content. For example, we have discussed the Class List Merge Template (Figure 7) in the context of adding a student to a class, but it is not specific to that function. A single instance of a Merge operator using the Class List Merge Template could merge documents representing any of the following events into a Class List: the addition of a student to a class, a change of professor for a class, or the addition of a new class. Note that for each type of input document, different actions must occur. For a professor change, the appropriate Class element must be updated and for a class addition, a new Class element must be inserted into the Class List. This type of functionality is difficult to effect in a relational system, where the type and order of operations must be decided in advance. The flexibility of the Merge operation is possible in part because the Class List Merge Template fully specifies how to merge all of these updates into the Class List document and in part because the Merge operation does not require the input data to exactly match the structure of the Merge Template.

3 Related Work

The Niagara [8] and Lore [6] projects have built database systems designed for querying XML data. The Tukwila [7] project has developed an adaptive query processing system and is currently focusing on processing XML streams. Our work provides a new operation, Merge, which is not currently available in any of these systems. YAT uses XML for data integration [3]. Merge differs from data integration in that we focus solely on aggregating data over XML streams and do not address issues such mediation and query reformulation. Some XML processing systems [3, 8] flatten the data and then process it using relational-style operators, constructing the XML result once the processing is complete. In contrast, Merge works on XML data in a tree-based fashion without flattening it first. Flattening may not be tractable when the data is highly irregular.

The Accumulator described in the Introduction, which maintained current, high, low and average prices for a set of stocks, can be seen as a view over the incoming stream of stock quote data. As data (stock quotes) arrive on the stream the "view" (Accumulator) is updated. In this way, our work is similar to work on Materialized View Updates. View updates have been studied extensively in the context of relational database systems [4]. Research has been done on view updates for semi-structured data; however, much of that work supposes that the updates are identified by OIDs [1, 12]. Updates to XML documents may be XML documents, Updategrams [11], or update operations as proposed by Tatarninov *et al.* [10]. In any case, OIDs are unlikely to be available.

Buneman *et al.* have proposed two operators: Deep Union and Deep Update, which are similar in nature to Merge [2]. Deep Union and Deep Update are used to combine and update edge-labeled trees that are similar, but not identical to, XML trees. The WHAX view update system uses Deep Union as a mechanism for updating views [5]. Deep Union and Deep Update resemble the Union and Join operators of the Verso Algebra [9], which uses a data model based on Nested Relations. These data models require that elements (tree nodes in the models used by Deep Union and WHAX and tuples in the Verso model) have keys. Neither work has the concept of an explicit external Merge specification such as our Merge Template. In these systems, the functions that occur when two documents are unioned or joined is solely defined by the operator definition and the structure of the document. The difference is similar to that between natural join and traditional explicit join.

4 Conclusion and Future Work

We have described a new operation, Merge, that pieces together two XML documents in a flexible, general fashion. Repeated application of Merge can be used to accumulate and aggregate data from an incoming stream. Two key features of Merge are that its actions depend directly on the structure of the input data and that it can easily and effectively handle irregular and incomplete input. Documents are not required to have DTDs to be merged and IDREFs are supported. A Merge-based Accumulator has been implemented in the Niagara Internet Query System and performance tests are in process. In addition, we are in the process of creating a formal specification for the Merge operation based on lattice theory.

In the future, we expect to add support for deletion and ordered documents to Merge. Currently, we do not have a mechanism for updating the Accumulator to reflect, for example, that a student has dropped a class. In addition, the concept of order needs to be integrated into the Merge definition so that Merge can support XML documents that convey semantic meaning by the order of their elements.

References

[1] S. Abiteboul, J. McHugh, M. Rys,V. Vassalos, J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. *Proceedings of the 1998 VLDB Conference*, New York, NY, August 1998.

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

[3] V. Christophides, S. Cluet, and J. Simeon. On Wrapping Query Languages and Efficient XML Integration. In *Proceedings of the 2000 ACM SIGMOD Conference*, Dallas, TX, May 2000.

[4] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):3-18, June 1995.

[5] H. Liefke and S. B. Davidson. View Maintenance for Hierarchical Semistructured Data. *2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2000)*, London-Greenwich, United Kingdom, September 2000.

[6] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54-66, September 1997.

[7] Z. Ives, A. Levy, D. Weld, D. Florescu, M. Friedman. Adaptive Query Processing for Internet Applications. *IEEE Data Engineering Bulletin*, 23(2):19-26, June 2000.

[8] J. Naughton, D. DeWitt, D. Maier, *et al*. The Niagara Internet Query System. http://www.cs.wisc.edu/niagara.

[9] M. Scholl, et. al. VERSO: A Database Machine Based on Nested Relations. *Nested Relations and Complex Objects, Papers from the Workshop "Theory and Applications of Nested Relations and Complex Objects"*, Darmstadt, Germany, April 1987. Lecture Notes in Computer Science, Vol. 361, Springer 1989.

[10] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proceedings of the 2001 ACM-SIGMOD Conference*, Santa Barbara, CA, May 2001.

[11] http://msdn.microsoft.com/xml/general/updategrams.asp

[12] Y. Zhuge, H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *Proceedings of the 14th IEEE Conference on Data Engineering (ICDE '98)*, Orlando, February 1998.