Managing XML Data with Evolving Schema

B V N Prashant and P Sreenivasa Kumar

Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai, India-600036
Email: {bvnprash,psk}@cse.iitm.ernet.in

Abstract

XML databases evolve during their lifetime to address new requirements and reflect changes in the real world. The structure of XML data defined by either DTD or XML schema, undergoes changes to accommodate changing user needs. In this paper, we focus on the problem of DTD change operations which will help users to perform necessary changes to the schema. We propose a set of three high-level operators as an extension to existing set of primitive DTD change operators [3]. These operators are supported by algorithms to generate XSLT scripts which transform instances of current DTD to conform with the changed DTD.

1 Introduction

Database systems must have the ability to respond to changes in the real world by allowing the schema to evolve, but in practice database designers freeze the schema design prior to the development of a database application. Our goal is to provide a set of schema change operators through an easy-to-use interface, to the user, for changing the schema. This interface automates the process of changing schema along with its data, and hides the unnecessary details from the user. XML fits well as a data model for applications whose schema requirements change frequently. We have used DTD to describe the schema of XML data.
The contributions of this paper are:

1. We propose a set of three high-level DTD evolution operators and study their effect on DTD.
2. We devise algorithms for generating XSLT scripts that transform instances of current DTD to conform with the changed DTD.

2 Related work

The problem of schema evolution in XML databases was first studied by Kramer et al [3]. They proposed a set of primitive DTD evolution operators, which ensured that documents and their schema remained consistent even after changes were made to them. Their work does not include any high-level operators on DTD, but they claim that any high-level DTD change operation could be performed as a series of primitive operations. Though, in principle it is true, lot of additional work needs to be done by the user to correctly incorporate the effect of the high-level operation. This point will become clear later in the paper, when we discuss the specific high-level operators. Guerrini et al [2, 1] also proposed a set of primitive change operators which were based on XML schema. The focus of their work was mainly on the aspect of document revalidation which is an important part of schema evolution. Efficient techniques were introduced to reduce the time taken for re-validating the documents after every change during the process of schema evolution. In our work, we have augmented the set of primitive operators proposed by Kramer et al [3] with high-level operators and defined the semantics of these operations.

3 Background

3.1 Schema model

We represent DTD as a graph G(N,E), where N is the set of nodes and E is the set of edges.

- Nodes can be classified into three categories
  - Element type nodes
  - Attribute type nodes
  - Particle nodes
    - Group node: It groups a set of element types. Content of the group node can be of "sequence" type or "choice".
    - Element type reference node: It refers to an element type node.
- Edges are of two types

International Conference on Management of Data
COMAD 2006, Delhi, India, December 14–16, 2006

– Containment edges
– Element type reference edges

Few elements such as *Colleges*, *Students*(shown in Figure 3) play the role of only holding other elements. We name such elements as *container* elements.

3.2 Relationships

Relationships between elements of a DTD can exist in the following ways:

- **Implicit relationship:** The relationships may be implicit in the hierarchy of the DTD. If an element E_2 is a descendant of another element E_1, it implicitly indicates a relation between E_2 and E_1. If E_1 is a container element then E_2 is a member of E_1 collection or otherwise E_2 is a component of E_1.
- **Explicit relationship:** Two kinds of explicit relationship can exist between two elements E_1 and E_2.
  – Relationships can be made explicit by using IDREF attributes.
  – DTDs do not have any provision that helps in referring to an element without an ID attribute. Therefore, we assume that every element (except the *Container* element) without an ID attribute has a key sub-element called *keyLeafEle*(analogous to primary key in RDBMS). The *keyLeafEle* can be referred to by its value.

The DTD framework does not facilitate keeping track of relationships amongst the elements, so we have devised a way to record these relationships. Consider *Advisor-Advisee* relationship in Figure 3(a), it is captured as follows:

Role1: Advisor
Role2: Advisee
From: /Univ/Colleges/College/Departments /Dept/FacultyMembers/Faculty/Advisee/Rno/ @Rollref
To: /Univ/Colleges/College/Departments/Dept /Students/Student/@Rollno
Cardinality: one–to–many

An explicit reference may start at an element or an attribute and end at another element or attribute. It follows that four kinds of explicit relationships can exist between any two elements(See Figure 1).

4 Schema Evolution Operations

Primitive schema evolution operations on DTD can be broadly classified into operations used for creation, insertion, deletion, and modification. We build on the primitive operations proposed by Kramer et al [3] and define three high-level operations.

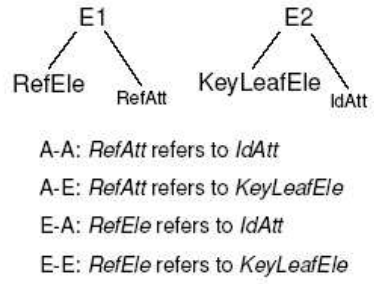

Figure 1: Possible Relations between elements E_1 and E_2

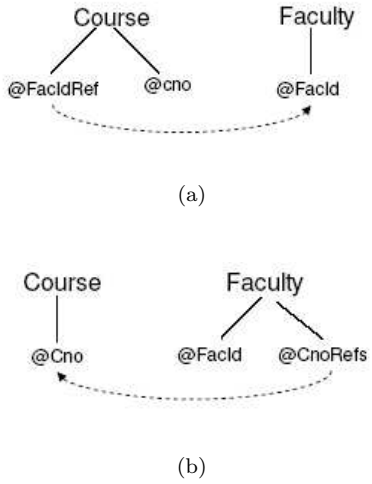

Figure 2: Shows the status of relationship between *Course* and *Faculty* "before" and "after" *relationship inverse*

1. **SubtreeMoveUp (r, n):** Subtree rooted at node r is moved up in the tree to node n where node n lies on the path from the root to the node r. For example, a subtree rooted at *Student* is moved and placed under *College*(Figure 3(a) and 3(b)).

2. **SubtreeMoveDown (r, n):** The subtree rooted at node r is moved down in the tree to node n. For example, subtree rooted at *Courses* is moved and placed under *Faculty*(Figure 3(a) and 3(b)).

3. **RelationshipInverse (n_1, n_2):** This operator is used to inverse the relationship between element nodes whose path expressions are given by n_1 and n_2. For example(Figure 2(a)), shows relationship between *Course* and *Faculty*. When this relationship is reversed then *Faculty* element will have a set of references that refer to *Course* elements as shown in Figure 2(b).

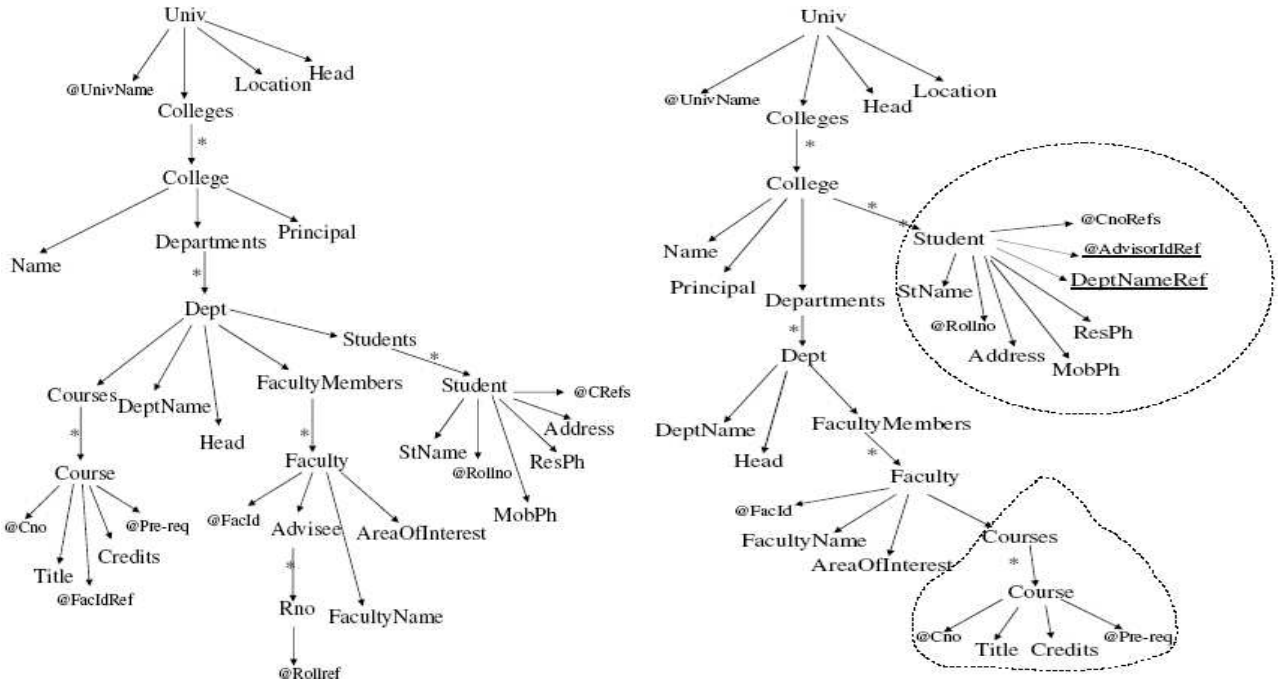

Figure 3: DTD versions S_1 and S_2

5 Effect of operations on DTDs

In this section we analyze the effect of the above operations on the DTD and propose methods for DTD transformation. Each one of the operations explained below refer to Figure 3(a) and Figure 3(b).

5.1 Subtree MoveUp

If the subtree rooted at some node r is moved up in the DTD graph and placed under one of the nodes t on the path from DTD root to the node r, then the references to the elements along that path r to t are added as children to r. For example as shown in Figure 3(b) *Student* subtree has been moved to the position under *College*. A new reference node referring to *Dept* has been added as a child to it.

- When a subtree is moved up it loses its hierarchical relationship with each node it crosses. We can continue maintaining the relationship by adding explicit references. The elements encountered during the move may have ID attribute or *keyLeafEle*. A reference to elements encountered during the move, is added under the subtree root r. In Figure 3(a) when *Student* subtree is moved up, we add *DeptNameRef* element under *Student*. This element refers to *Dept* by using value of its *keyLeafEle*.
- Container elements along the path r to t are ignored. For example, while moving *Students* subtree up, the element *Departments* is ignored.
- The quantifier for the new reference added is decided by the cardinality of the relationship between subtree root and elements encountered during the move. For example, When the *Student* subtree is moved up it crosses *Dept* which has one-to-many relationship with it. So a single reference to *Dept* will suffice, but if a student could belong to many departments then a set of references have to be added.
- The edge from target node t to subtree root r is assigned with the most general quantifier of all the edges that the subtree crosses. *Student* is '*' quantified with *College* as it crosses an edge with '*' as the quantifier.

5.2 Subtree MoveDown:

The subtree rooted at some node r may also be moved down to a node that lies on the path from its parent node to any leaf.

- When a subtree is moved down, new hierarchical relationships are formed. Existing relationships which infer the same meaning as the new relationship can be removed. For example, in Fig-

ure 3(a) when subtree *Courses* is moved down to *Faculty*. *Course* elements have a reference to the *Faculty* which becomes implicit after it is moved under *Faculty*. As a result, the attribute *FacIdRef* referring to the *Faculty* is redundant and can be safely removed.

- The existence of explicit relationship between subtree root r and each element e encountered during the move must be verified in both ways i.e. from r-to-e or e-to-r.

A subtree rooted at some node r may also be moved to some other node of the tree, which is not in the path from r to any leaf. This situation can be handled by *Subtree MoveUp* operation followed by *Subtree MoveDown* operation.

5.3 Relationship Inverse:

This operation makes use of the information stored in *RelationshipStore*. Each entry in the store has the relationship name, path expressions of the participating nodes and cardinality. Consider the example shown in Figure 3, where *Faculty-Advisee* relationship between elements *Faculty* and *Student* is established through attribute *@Rollref* referring to the attribute *@Rollno*.

- First step in *Relationship Inverse* between any element E_1 and E_2 is to remove the nodes(attributes or elements) under the source element E_1 that are directly involved in the relationship R. For example, in the Figure 3(a) *Rollref* attribute is removed.

- The next step is adding the new nodes. This is dependent on the type of relationship. If the relation R is one-to-many from E_1 to E_2, as in the case of *Faculty-Advisee* relationship, then a single IDREF attribute or a sub-element will suffice. For example, in the Figure 3(b) *advisorIdRef* has been added as sub-element, it refers to *FacId*.

6 Automated data translation

XSLT scripts can be used to transform instances of current DTD to conform with the changed DTD. We have designed algorithms to automate the process of script generation. XSLT script for data transformation depends on the type of DTD change operator. Each algorithm uses current version of DTD, data instances and other auxiliary information to generate XSLT script. These algorithms scan the entire DTD tree in the DFS order and generate *match* templates for each element type. For every deleted element type, corresponding empty template is generated. An empty template suppresses the output of the corresponding elements in instance documents, during transformation. New elements or attributes are added using *xsl:attribute* or *xsl:element* constructs.

7 Conclusions and Future Work

We have proposed a set of high-level DTD change operators and described their semantics. We have also implemented system to test the working of the operators. Our system automatically generates an XSLT scripts which transform old data to conform with the new DTD.
As part of our future work, we are looking at another approach of solving the problem of schema evolution i.e. creating DTD versions after every change operation. This approach eliminates the need for data transformation but induces a new problem of *Query Translation* (also called *Query Reformulation*). This problem is due to the fact that users cannot issue different queries on different versions of DTD. Practically, users would be aware of only the most recent DTD version amongst all the existing versions. Currently, we are in the process of developing algorithms for the *Query translation*.

References

[1] G. Guerrini, M. Mesiti, and D. Rossi. Impact of XML schema evolution on valid documents. In *WIDM*, pages 39–44, 2005.

[2] M. Mesiti, R. Celle, M. A. Sorrenti, and G. Guerrini. X-Evolution: A System for XML Schema Evolution and Document Adaptation. In *EDBT*, pages 1143–1146, 2006.

[3] H. Su, D. Kramer, L. Chen, K. T. Claypool, and E. A. Rundensteiner. XEM: Managing the evolution of XML Documents. In *Eleventh International Workshop on Research Issues in Data Engineering: Document Management for Data Intensive Business and Scientific Applications, Heidelberg, Germany, 1-2 April 2001*, pages 103–110. IEEE Computer Society, 2001.