Information Preservation in XML-to-Relational Mappings

Denilson Barbosa[1], Juliana Freire[2], and Alberto O. Mendelzon[1]

[1] Department of Computer Science – University of Toronto
{dmb,mendel}@cs.toronto.edu
[2] OGI School of Science and Engineering – Oregon Health and Sciences University
juliana@cse.ogi.edu

Abstract. We study the problem of storing XML documents using relational mappings. We propose a formalization of classes of mapping schemes based on the languages used for defining functions that assign relational databases to XML documents and vice-versa. We also discuss notions of information preservation for mapping schemes; we define *lossless* mapping schemes as those that preserve the structure and content of the documents, and *validating* mapping schemes as those in which valid documents can be mapped into legal databases, and all legal databases are (equivalent to) mappings of valid documents. We define one natural class of mapping schemes that captures all mappings in the literature, and show negative results for testing whether such mappings are lossless or validating. Finally, we propose a lossless and validating mapping scheme, and show that it performs well in the presence of updates.

1 Introduction

The problem of storing XML documents using relational engines has attracted significant interest with a view to leveraging the powerful and reliable data management services provided by these engines. In a mapping-based XML storage system, the XML document is represented as a relational database and queries (including updates) over the document are translated into queries over the database. Thus, it is important that the translation of XML queries and updates into SQL transactions be correct. In particular, only updates resulting in *valid* documents should be allowed. To date, the focus of the work in designing mapping schemes for XML documents has been on ensuring that XML queries over the documents can be answered using their relational representations. In this paper, we study the problem of designing mapping schemes that ensure only valid update operations can be executed.

An important requirement in mapping systems is that any query over the document must be computable from the database that represents the document. In particular, the mapping must be *lossless*, that is, the document itself must be recoverable from its relational image. When updates are considered, one must be able to test whether an operation results in the representation of a valid document *before* applying the operation to the database. This amounts

Z. Bellahsène et al. (Eds.): XSym 2004, LNCS 3186, pp. 66–81, 2004.

to checking whether the resulting database represents a well-formed document (i.e., a tree) and that this document conforms to a document schema. Several works [4,22] exploit information about a document schema for designing better mapping schemes; the metric in most of these is query performance, for example, the number of joins required for answering certain queries. However, to the best of our knowledge, there has been no work on ensuring that only valid updates can be processed over relational representations of documents.

1.1 Related Work

There is extensive literature on defining *information preserving* mappings for relational databases [17]; our notion of *lossless* mapping schemes is inspired by that of lossless decompositions of relations. The incremental checking of relational view/integrity constraints has also been widely studied [15,12]. Some of the view maintenance techniques have been extended to semi-structured models that were precursors of XML [23]. Another related problem, the incremental validation of XML documents w.r.t. XML schema formalisms, has been studied in [3,5,19]; we use the incremental algorithms proposed in [3] in this work.

The literature on mapping XML documents to relational databases is also vast [16]. One of the first proposals was the Edge [14] scheme, a generic approach that explicitly stores all the edges in a document tree. Departing from generic mappings, several specialized strategies have been proposed which make use of schema information about the documents to generate more efficient mappings. Shanmugasundaram *et al.* [22] describe three specialized strategies which aim to minimize data fragmentation by *inlining*, whenever possible, the content of certain elements as columns in the relation that represents their parents. LegoDB [4] is a cost-based tool that generates relational mappings using inlining as well as other operations [21]; the goal there is to find a relational configuration with lowest cost for processing a given query workload on a given XML document. STORED [11] is a hybrid method for mapping semistructured data in which relational tables are used for storing the most regular and frequent structures, while *overflow* graphs store the remaining portions of the semistructured database. None of these works deals with checking the validity of updates.

Orthogonal to designing mapping approaches is the translation of constraints in the document schema to the relational schema for the mappings. For instance, the propagation of keys [9] and functional dependencies [8] have been studied. Besides capturing the semantics of the original document schema, these techniques have been shown to improve the mappings by, e.g., reducing the storage of redundant information [8]. However, these works do not consider the translation of the constraints defined by the content models (i.e., regular expressions) in the document schema, which is the focus of this work. We note that these techniques can be applied directly in our method.

Designing relational mappings for XML documents can be viewed as the reincarnation of the well-known problem of "simulating" semantic models in relational schemas [1], and there are mapping schemes that follow this "classical" approach. However, the semantic mismatch between the XML and relational

models is much more pronounced than in previously considered models. For example, XML data is inherently ordered; moreover, preserving this ordering is crucial for deciding the *validity* of the elements in the document. Mani and Lee [18] define an extended (yet unordered) ER model for describing XML data, and provide algorithms for translating these models into relational schemas.

For this study, we had to fix an update language; we settled for a simple language, as the goal was to verify the feasibility of our approach rather than proposing a new language. For a discussion on updating XML, see [24].

Contributions and Outline. In Section 3, we provide a formalization of mapping schemes as pairs of functions, the mapping and publishing functions, that assign relational databases to XML documents and vice-versa. We introduce in Section 3.1 the notion of parameterized classes of mapping schemes, which are defined by the languages used for specifying the mapping function, the publishing function, and the relational constraints in the target schema. In Section 3.2 we introduce $\mathcal{XDS}$: a natural class of mapping schemes powerful enough to express all, to the best of our knowledge, mapping schemes in the literature. In Section 4, we characterize mapping schemes with respect to information preservation. In particular, we define *lossless* mapping schemes as those that ensure all XML queries over a document can be executed over its corresponding database; and *validating* mapping schemes, which ensure only valid updates can be effected. We also show that testing both properties for $\mathcal{XDS}$ mapping schemes is undecidable. In Section 5 we propose an $\mathcal{XDS}$ mapping scheme that is both lossless and validating, and discuss the incremental maintenance of the constraints in such mappings in the presence of updates in Section 6.

2 Preliminaries

We model XML documents as ordered labeled trees with element and text nodes. The root of the tree represents the whole document and has one child, which represents the outermost element. Following the XML standard [27], the *content* of an element node e is the (possibly empty) list of element or text nodes that are children of e. For simplicity, in an element with *mixed* content (i.e., element and text nodes as its children), we replace each text node by an element node whose label is #PCDATA and whose only child is the given text node. Thus, each element in the tree has either text or element nodes as its content; also, all text nodes are leaf nodes in the tree.

More precisely, we define the following. Let $\mathcal{I}$, $\mathcal{D}$ be two disjoint, countably infinite sets of node ids and values.

Definition 1 (XML Document) *An XML document is a tuple $\langle T, \lambda, \tau, \nu \rangle$, such that T is an ordered tree whose nodes are elements of $\mathcal{I}$, $\lambda : \mathcal{I} \to \mathcal{D}$ is an assignment of labels to nodes in T, $\tau : \mathcal{I} \to \{element, text\}$ is an assignment of node types to the nodes in T, and $\nu : \mathcal{I} \to \mathcal{D}$ is an assignment of values to text nodes in T (i.e., ν is undefined for a node e if $\tau(e) \neq text$).*

We model document schemas as sets of rules which constrain the structure and the labels of the elements in the document. Such *element declaration rules* assign *content models* to *element types*. An element type is defined by a path expression of the form $[c_i]/l_i$ where l_i is an element label and c_i is an optional path expression specifying the *context* in which l_i occurs. We restrict contexts to be path expressions matching $E := a \mid E/E \mid E//E$, where a is either an element label or the wildcard $*$ for matching elements of any label. We say an element is of type t_i if it is in the result set of evaluating $[c_i]/l_i$ over the document. Following the XML standard, a content model is specified by a 1-unambiguous regular expression [6] of the form $E := \varepsilon \mid a \mid \#\text{PCDATA} \mid (E) \mid E|E \mid E, E \mid E* \mid E+ \mid E?$, where ε represents the empty string, a is an element label, and #PCDATA represents textual content. Note that this captures all four content models for XML elements [27].

Definition 2 (Document Schema) *A document schema X is a set of element declaration rules of the form $t_i \leftarrow r_i$ where t_i is an element type and r_i is a content model, such that for any document D that is* valid *with respect to X (as defined below), for each element e in D, there is exactly one element type t_i such that e is of type t_i.*

Let e be an element of type t_i, and $c_1, \ldots, c_n$ be its (ordered) children; we say e is *valid* with respect to rule $t_i \leftarrow r_i$ if the string $\lambda(c_1) \cdots \lambda(c_n)$ matches r_i. We say a document D is valid with respect to a document schema X, denoted $D \in L(X)$ if all elements in D are valid with respect to the rules in X corresponding to their respective types.

Let $\mathcal{I}$, $\mathcal{D}$ be two disjoint countably infinite sets of surrogates and constants. We define relational database as follows:

Definition 3 (Relational Database) *Each relation scheme R has a set (possibly empty) of attributes of domain $\mathcal{I}$, called the* surrogate attributes of *R, and a set (possibly empty) of attributes with domain $\mathcal{D}$. Everything else is defined as customary for the relational model.*

Intuitively, renaming node ids in a document does not yield a new document. Similarly, renaming surrogates in a database does not create a new mapping. In order to capture these properties, we define:

Definition 4 (Document Equivalence) *XML documents $D_1 = \langle T_1, \lambda_1, \tau_1, \nu_1 \rangle$, and $D_2 = \langle T_2, \lambda_2, \tau_2, \nu_2 \rangle$, are equivalent, denoted by $D_1 \equiv D_2$, if there exists an isomorphism ϕ between T_1 and T_2 s.t. $\lambda_1(v) = \lambda_2(\phi(v))$, $\tau_1(v) = \tau_2(\phi(v))$, and $\nu_1(v) = \nu_2(\phi(v))$, for all $v \in T_1$.*

Definition 5 (Database Equivalence) *Two database instances I_1, I_2 are equivalent, written $I_1 \equiv I_2$ if there exists a bijection on $\mathcal{I} \cup \mathcal{D}$ that maps $\mathcal{I}$ to $\mathcal{I}$, is the identity on $\mathcal{D}$, and transforms I_1 into I_2.*

(The notion of database equivalence discussed above has been called *OID equivalence* in the context of object databases [1].)

3 XML-to-Relational Mappings

We define an XML-to-relational *mapping scheme* as a triple $\mu = (\sigma, \pi, S)$, where S is a relational schema; σ is a *mapping function* that assigns databases to XML documents; and π is a *publishing function* that assigns XML documents to databases. More precisely, let $\mathcal{X}$ be the set of all XML documents; we define:

Definition 6 (Mapping Scheme) *A mapping scheme is a triple $\mu = (\sigma, \pi, S)$, where $\sigma : \mathcal{X} \to \mathcal{R}(S)$ is a partial function, and $\pi : \mathcal{R}(S) \to \mathcal{X}$ is a total function. Moreover:*

1. *for all D_1, D_2 $D_1 \equiv D_2$ implies $\sigma(D_1) \equiv \sigma(D_2)$,*
2. *for all I_1, I_2 $I_1 \equiv I_2$ implies $\pi(I_1) \equiv \pi(I_2)$.*

Defining σ to be a partial function allows mapping schemes where the relational schemas are customized for documents conforming to a given document schema X [4,22]; in other words, mapping schemes where $\mathrm{Dom}(\sigma) = L(X)$. On the other hand, requiring π to be total ensures that any legal database (as defined below) can be mapped into a document. Conditions 1 and 2 in Definition 6 ensure that both σ and π are *generic* (in the sense of database theory [1]): they map equivalent documents to equivalent databases and vice-versa.

3.1 Parameterized Classes of Mapping Schemes

We define classes of mapping schemes based on the languages used for specifying σ, S, and π. The power of these languages determines what kinds of mappings can be specified. For example, some sort of counting mechanism in the mapping language is required for specifying mapping functions that encode "interval-based" element ordering [10].

3.2 The $\mathcal{XDS}$ Class of Mappings

We now describe one particular class of mapping schemes that captures all mappings proposed in the literature. In summary, we use an XQuery-like language for defining σ, we allow boolean datalog queries with inequality and stratified negation to be specified in the relational schema, and we use XQuery coupled with a standard publishing framework such as SilkRoute [13]. We will call this class of mappings $\mathcal{XDS}$, (for XQuery, Datalog, and SilkRoute).

The Mapping Language. The language consists of XQuery augmented with a clause `sql ... end` for specifying SQL insert statements, and to be used instead of the `return` clause in a FLOWR expression. The semantics of the mapping expressions is defined similarly to the usual semantics of FLOWR expressions: the `for, let, where, order by` clauses define a list of tuples which are passed, one at a time, to the `sql ... end` clause, and one SQL transaction is issued per such tuple. Unlike a query, a mapping expression does not return any values, and is declared within a `procedure`, instead of an XQuery function.

The Relational Constraints. In $\mathcal{XDS}$ mapping schemes, each constraint in the relational schema is a boolean query that must evaluate to false unless the constraint is violated. We say that a relational instance is *legal* if it does not violate any of the constraints. Each of these queries is expressed as a set of datalog rules, augmented with stratified negation and not-equals. This language allows easy expression of standard relational constraints such as functional dependencies and referential integrity, while the recursion in datalog can be used to express, for example, that the children of an element conform to the element's content model, as shown in the Edge^{++} mapping of Section 5.

The Publishing Language. Publishing functions are arbitrary XQuery expressions over a "canonical" XML view of a relational database. That is, each relation is mapped into an element whose children represent the tuples in that relation in the standard way (i.e., one element per column). This is the approach taken by SilkRoute [13] and XPERANTO [7]. Of course, there are several different "canonical" views (i.e., documents) that represent the same database, as no implicit ordering exist among tuples or relations in the database.

It is easy to see that fairly complex mapping schemes are possible in the $\mathcal{XDS}$ class. In fact, all mapping schemes that we are aware of in the literature are in $\mathcal{XDS}$. Below, we give an example of such a mapping scheme.

Example 1. The Edge mapping scheme [14] belongs to $\mathcal{XDS}$ and can be described as follows. The relational schema S contains the relations (primary keys are underlined):

$$\mathsf{Edge}(\mathit{parent} : \mathcal{I}, \underline{\mathit{child}} : \mathcal{I}, \mathit{ordinal} : \mathcal{D}, \mathit{label} : \mathcal{D}), \quad \mathsf{Value}(\underline{\mathit{element}} : \mathcal{I}, \mathit{value} : \mathcal{D})$$

The Edge relation contains a tuple for each element-to-element edge in the document tree, consisting of the ids of the parent and child, the child's ordinal, and the child's label. The Value relation contains a tuple for each leaf in the tree. The relational schema also contains constraints for ensuring that: there is only one root element; every child has a parent; every element id in Value appears also in Edge; and that the ordinals of nodes in the tree are consistent. Note these constraints are easily expressed as boolean datalog programs.

The mapping function σ is defined by a recursive procedure that visits the children of a given node, storing the element nodes in the Edge relation and the text nodes in Value relation:

```
define procedure map_node($e as node){
  for $n at $i in $e/*
  if ($n instance of element()) then
    sql INSERT INTO Edge VALUES (id($e),id($n),$i,name($n)) end;
    map_node($n);
  if ($n instance of text()) then
    sql INSERT INTO Value VALUES (id($e),$n) end; }

(: to map the document :)
map_node(doc("doc.xml"));
```

```
<a>
  <b>foo</b>
    mixed
  <b>bar</b>
</a>
```

(a) Document

Edge:

parent	*child*	*ordinal*	*label*
0	1	0	`a`
1	2	0	`b`
1	3	1	`#PCDATA`
1	4	2	`b`

Value:

element	*value*
2	`foo`
3	`mixed`
4	`bar`

(b) Edge mapping

Fig. 1. Edge mapping of a document.

The `id()` function used in the procedure above can be any function that assigns a unique value to each node in the tree; for concreteness, we assume the function returns the node's discovery time in a DFS traversal of the tree, and that the discovery time of the document node (i.e., the node that is the parent of the root element in the document) is 0. Figure 1 shows a document and its image under the Edge mapping.

Finally, the publishing function π is defined as follows. The publishing of an element e is done by finding all its children in the `Edge` and `Value` elements in the "canonical" published views, and returning them in the same order as their `ordinal` values. We use the `order by` clause in the FLOWR expression to reconstruct the sequence of element and text nodes in the same order as in the original document.

4 Preserving Document Information

As discussed in Section 1, a mapping-based storage system for an XML document should correctly translate any query over the document, including updates, into queries over the mapped data. In this section, we discuss two properties that ensure that queries and valid updates can always be processed over databases that represent documents.

The generally assumed notion of information preservation for a mapping scheme is that it must be *lossless*; that is, one must be able to reconstruct any (fragment of a) document from its relational image [11]. More precisely, we define:

Definition 7 (Lossless Mapping Scheme) *A mapping scheme $\mu = (\sigma, \pi, S)$ is* lossless *if for all $D \in \mathrm{Dom}(\sigma)$, $\pi(\sigma(D)) \equiv D$.*

Informally, losslessness ensures that all queries over the documents can be answered using their mapped images: besides the naive approach of materializing $\pi(\sigma(D))$ and processing the query, there are several techniques for translating the XML queries into SQL for specific mappings and XML query languages [16]. The following is easy to verify:

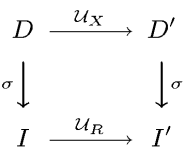

Fig. 2. Updating documents and their mapping images.

Proposition 1 *The Edge mapping scheme is lossless.*

While losslessness is necessary for document reconstruction, it does not ensure that databases represent valid documents, nor that updates to representations of valid documents result in databases that also represent a valid documents. As an example, consider a mapping scheme for documents conforming to the schema $X : a \leftarrow (b|\#\text{PCDATA})*;\ b \leftarrow \#\text{PCDATA}$, and recall the (lossless) Edge mapping scheme discussed in Example 1. Figure 1(a) shows a valid document with respect to X, and its relational representation is shown in Figure 1(b). It is easy to see that any *permissible* update (i.e., an update resulting in a valid document) to the document can be translated into an equivalent SQL transaction over the document's representation. However, it is also possible to update the representation in a way that results in a legal database that does not represent a valid document with respect to X. For instance, the SQL transaction that inserts the tuple (3,4,0,`c`) into Edge and (4,0,`foo`) into Value, which results in a legal database, has no equivalent permissible XML update because it corresponds to inserting an element labeled c as a child of the second b element, which violates the document schema.

We define a property of mapping schemes that ensures that all and only valid documents can be represented:

Definition 8 (Validating Mapping Scheme) $\mu = (\sigma, \pi, S)$ *is* validating with respect to document schema X *if* σ *is total on* $L(X)$*, and for all* $I \in \mathcal{R}(S)$*, there exists* $D \in L(X)$ *such that* $I \equiv \sigma(D)$*.*

For simplicity, we drop the reference to the document schema X when discussing a validating mapping scheme, and just say that a document is valid if it is valid with respect to X. Intuitively, a mapping scheme is *validating* with respect to X if it maps all valid documents to some legal database and if every legal database is (equivalent to) the image of some valid document under the mapping. This implies that all permissible updates to documents can be translated into equivalent updates over their mappings, and vice-versa, as depicted in the diagram in Figure 2.

We conclude this section by showing that there are theoretical impediments to the goal of automatically designing lossless and/or validating $\mathcal{XDS}$ mappings. The following results are direct consequences of the interactions of context-free

languages and chain datalog programs[1] [25,26]. The proofs are omitted in the interest of space and are given in the full version of the paper [2].

Theorem 1 *Testing losslessness of $\mathcal{XDS}$ mapping schemes is undecidable.*

Theorem 2 *Testing validation of $\mathcal{XDS}$ mapping schemes is undecidable.*

5 A Lossless and Validating $\mathcal{XDS}$ Mapping Scheme

In this section we introduce Edge^{++}: a lossless and validating mapping scheme in $\mathcal{XDS}$. The Edge^{++} mapping scheme is an extension of the Edge mapping scheme that includes constraints for ensuring the validation property. In this section we fully describe the procedure for creating a mapping scheme $\mu = (\sigma, \pi, S)$ given a document schema X, and show that μ is both lossless and validating with respect to X.

5.1 The Relational Schema

Recall the definition of relational databases in Section 2. Let $\mathcal{I}' = \mathcal{I} \cup \{\#\}, \# \notin \mathcal{I}$ (the symbol # will be used for marking elements that have no children); also, let $\mathcal{Q} \subseteq \mathcal{D}$ be a set of states, $\mathcal{T} \subseteq \mathcal{D}$ be a set of type identifiers, and $\mathcal{B} \subseteq \mathcal{D}$ denote the set of boolean constants. The relational schema of Edge^{++} is as follows:

$$\mathsf{Edge}(\textit{parent} : \mathcal{I}, \underline{\textit{child}} : \mathcal{I}, \textit{label} : \mathcal{D}), \quad \mathsf{FLC}(\underline{\textit{parent}} : \mathcal{I}, \textit{first} : \mathcal{I}', \textit{last} : \mathcal{I}'),$$
$$\mathsf{ILS}(\textit{left} : \mathcal{I}, \textit{right} : \mathcal{I}), \quad \mathsf{Value}(\underline{\textit{element}} : \mathcal{I}, \textit{value} : \mathcal{D}), \quad \mathsf{Type}(\underline{\textit{element}} : \mathcal{I}, \textit{type} : \mathcal{T}),$$
$$\mathsf{Transition}(\underline{\textit{type}} : \mathcal{T}, \underline{\textit{from}} : \mathcal{Q}, \underline{\textit{symbol}} : \mathcal{D}, \textit{to} : \mathcal{Q}, \textit{isAccepting} : \mathcal{B})$$

The Edge and Value relations are used essentially as in the Edge mapping scheme, except that the ordering of the element nodes is not explicitly stored. Instead, we use the FLC and ILS relations to represent the successor relation among element nodes that are children of the same parent element (ILS stands for "immediate-left-sibling" and FLC stands for "first-last-children"). The choice of keeping the ordering of the elements using the FLC and ILS relations is motivated by the fact that they allow faster updates to the databases, as we do not need to increase (resp. decrease) the ordinals of potentially all children of an element after an insertion (resp. deletion). The constraints that we discuss here require only that we can access the next sibling of any element, and, of course, can be adapted to other ordering schemes (e.g., interval-based).

More precisely, the FLC relation contains a tuple for each element e in the document whose content model is not #PCDATA, consisting of the id of e and the ids of its first and last children; if e has no content (i.e., no children), a tuple $(s_e, \#, \#)$ is stored in FLC, where s_e is the surrogate to e's id. The ILS relation contains a tuple consisting of the ids of consecutive element nodes that

[1] Chain datalog programs seek pairs of nodes x and y in a graph such that there exists a path $x \rightsquigarrow y$ whose labels spell a word in an associated language.

are children of the same parent. Thus, if $c_1, \ldots, c_n$ are the ordered children of an element e; we have $(s_e, s_{c_1}, s_{c_n}) \in \mathsf{FLC}$, and $\{(s_{c_1}, s_{c_2}), \ldots, (s_{c_{n-1}}, s_{c_n})\} \subseteq \mathsf{ILS}$. The Type relation contains the types of each element in the document. Finally, the Transition relation stores the transition functions of the automata that correspond to the content models in the document schema.

Next, we discuss the two kinds of constraints defined in the relational schema.

Structural Constraints. Similarly to the Edge mapping scheme, the structural constraints ensure that any legal database encodes a well-formed XML document (i.e., an ordered labeled tree). We also need to ensure that the ordering of the element nodes is consistent, and that each element has a type. We note that these constraints are easily encoded as boolean datalog programs. For instance, the following constraint ensures that no element that is marked as having no children is the parent of any other element:

$$\mathsf{invalid} :\!- \mathsf{FLC}(p, \#, \#), \mathsf{Edge}(p, \_, \_)$$

Validation Constraints. The validation constraints encode the rules in the document schema into equivalent constraints over the relations in S, to ensure that the document represented by the mapping is indeed valid. For each rule $t_i \leftarrow r_i$ in the document schema, we define the following datalog program:

$$\mathsf{reach}_{t_i}(p, \#, s) :\!- \mathsf{FLC}(p, \#, \#), \mathsf{Type}(p, t_i), \mathsf{Transition}(t_i, q_0, \varepsilon, s, \_) \quad (1)$$

$$\mathsf{reach}_{t_i}(p, c, s) :\!- \mathsf{Edge}(p, c, x), \mathsf{FLC}(p, c, \_), \mathsf{Type}(p, t_i), \mathsf{Transition}(t_i, q_0, x, s, \_) \quad (2)$$

$$\mathsf{reach}_{t_i}(p, c, s) :\!- \mathsf{reach}_{t_i}(p, x, y), \mathsf{ILS}(x, c), \mathsf{Type}(p, t_i), \mathsf{Edge}(p, c, w), \quad (3)$$
$$\mathsf{Transition}(t_i, y, w, s, \_)$$

$$\mathsf{accept}_{t_i}(p) :\!- \mathsf{reach}_{t_i}(p, c, s), \mathsf{FLC}(p, \_, c), \mathsf{Type}(p, t_i), \mathsf{Transition}(t_i, \_, \_, s, \mathit{true})$$

$$\mathsf{invalid}_{t_i} :\!- \mathsf{FLC}(p, \_, \_), \neg \mathsf{accept}_{t_i}(p)$$

The boolean view $\mathsf{invalid}_{t_i}$ evaluates to true iff there is some element p of type t_i whose contents are invalid with respect to r_i. The recursive view reach_{t_i} simulates the automaton using the labels of the children of each element p as follows. The simulation starts with rules (1) or (2); rule (1) applies only if the element has no content (the constant q_0 denote the starting state of the automaton for r_i). Rule (3) carries out the recursion over the children of p. Finally, the accept_{t_i} rule checks that the computation is accepting if the state s reached after inspecting the last child c (note that c could be # if the element has no content) is accepting.

5.2 The Mapping Function

As in Edge, we define a recursive function that maps all nodes in the document tree. Besides that, the mapping procedure in Edge^{++} also assigns types to the elements in the document. Of course, the types assigned to elements must match those stored in the Transition relation; for concreteness, we assume that type t_i is represented by the integer i. The mapping of the document is then defined by the following function shown in Figure 3.

```
define procedure map_node($e as node){
  let $cc := $e/*
  for $n in $cc
  if ($n instance of text()) then
    sql INSERT INTO Value VALUES (id($e), data($e)) end;
  else
  if ($n instance of element()) then
    if count($cc)=0 then (:the element has no children:)
      sql INSERT INTO FLC VALUES (id($e),'#','#') end;
    else
      let $first := $cc[1], $last := $cc[count($cc)]
      for $c at $i in $cc
      sql INSERT INTO Edge VALUES (id($e),id($c), name($c));
          INSERT INTO FLC VALUES (id($e), id($first), id($last));
      end;
      if ($i > 1) then
        let $j := $cc[$i-1];
        sql INSERT INTO ILS VALUES (id($c),id($j)) end;
      else ()
    map_node($n);
}
map_node(doc("doc.xml"));
(: the following assign the types to elements :)
for $e in $d/t1 sql INSERT INTO Type VALUES (id($e),1) end;
...
for $e in $d/tn sql INSERT INTO Type VALUES (id($e),n) end;
```

Fig. 3. Mapping function for Edge^{++}.

5.3 The Publishing Function

The publishing function π in Edge^{++} is straightforward. Note that publishing a single element e of type x is done by visiting all elements in the published view that have e as parent. In order to retrieve the elements in their original order, we use a recursive function that returns the next sibling according to the successor relation stored in ILS. In order to publish a subtree one can recursively apply the simple method above.

The following is easy to verify:

Proposition 2 *Let X be a document schema, then the Edge^{++} mapping scheme $\mu = (\sigma, \pi, S)$ is lossless and validating with respect to X.*

6 Updates and Incremental Validation

Any update (i.e., SQL transaction) over an Edge^{++} mapping succeeds only if the resulting database satisfies the constraints in the schema (i.e., represents a valid document). Evidently, testing all constraints after each update is inefficient and, in most cases, unnecessary [15]. In this section, we discuss efficient ways of checking the validity of the Edge^{++} constraints in the presence of updates. We start by briefly discussing a simple update language for XML documents which can be effectively translated into SQL transactions over Edge^{++} mappings.

6.1 The Update Language

The update language provided to the user has a significant impact on the performance of a mapping-based storage system. In particular, there may be constraints in the relational schema that can be dropped if the user is not allowed to issue arbitrary SQL updates over the mapped data.

Consider the constraint in the Edge mapping scheme discussed in Example 1 that specifies that "every element that appears in the Value relation must also appear in the Edge relation". This constraint ensures that every #PCDATA value stored in the database is the content of some element in the document. Note that this constraint is necessary only if the user can write arbitrary SQL update statements that modify the Value relation, but can be dropped if inserting elements with textual content is always done by a transaction that inserts tuples in Edge and Value relations at the same time.

We note that it is reasonable to assume the user updates are issued in some XML update language, and these are translated into SQL transactions in a way that the "structural" constraints are preserved, so they need not be checked after each transaction.

Update Operations. To date, there is no standard update language for XML, and proposing a proper update language is outside the goals of this work; instead, we use a minimal set of atomic operations, consisting basically of insertions and deletions of subtrees.

- **Append**(p,y), where both p and y are elements, results in inserting y as the last child of p;
- **InsertBefore**(x,y), where both x and y are elements, results in inserting y as the immediate left sibling of x; this operation is not defined if x is the root of the document being updated;
- **Delete**(x), where x is an element, results in deleting x from the document.

For the Edge^{++} mapping scheme, it is straightforward to translate the primitive operations above into SQL transactions in a way that preserves structural validity.

6.2 Incremental Checking of Validating Constraints

The validating constraints in Edge^{++} mappings are recursive datalog programs that test membership in regular languages, which is a problem that has been shown to have low complexity. Patnaik and Immerman [20] show that membership in regular languages can be incrementally tested for insertion, deletion or single-symbol renaming in logarithmic time. By viewing the sequence of children of an element as a string generated by the regular expression for that element, Barbosa et al. [3] give a constant time incremental algorithm for matching strings to certain 1-unambiguous regular expressions. The classes of regular expressions considered in that work capture those most commonly used in real-life document schemas.

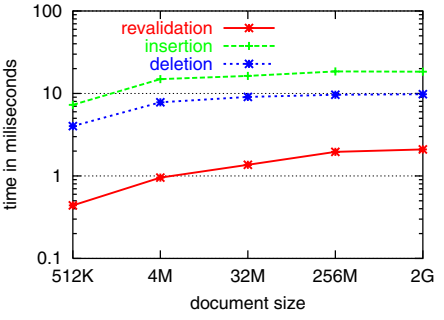

Fig. 4. Updates and incremental validation times.

Essentially, the idea behind the algorithms in [20,3] is to store the *paths* through the automata (i.e., sequence of states) together with the strings. Testing whether an update to the string is valid amounts to testing whether the corresponding path can be updated accordingly, while yielding another accepting path. As shown in [3], this test can be easily implemented as an efficient datalog program.

7 Experimental Evaluation

We show preliminary experimental results for the incremental maintenance of the validation constraints as defined in Section 5.1. Our experiments were run on a Pentium-4 2.5 GHz machine running DB2 V8.1 on Linux. We used several XMark documents of varying sizes (512KB, 4MB, 32MB, 256MB and 2GB); we note that the regular expressions used in the XMark DTD follow the syntactic restrictions discussed in the previous section, and, thus, are amenable to the simple algorithm discussed.

The implementation used in our experiments uses a more efficient relational schema than the one discussed in Section 5, which consists of horizontally partitioning the Edge relation based on the type of the parent element in each edge. That is, for each element type t_i, we define $\mathsf{Edge}_{t_i}(p, c, l)$:-$\mathsf{Edge}(p, c, l), \mathsf{Type}(p, t_i)$; the Transition table is partitioned in a similar fashion. Note that this results in eliminating some of the joins in the validating constraint. The workload in our experiments consists of 100 insertions and deletions of items for auctions in the North America region, each performed as a separate transaction. Each element inserted is valid and consists of an entire subtree of size comparable to those already in the document, and each delete operation removes one of the items inserted.

Figure 7 shows the times for executing the insertions, deletions and for incrementally recomputing (and updating) the validation constraints. The graph shows that, in practice, Edge^{++} can achieve good performance: per-update costs are dominated by SQL insert and delete operations; and the costs scale well with

document size, which is consistent with the results in [3]. It is worthy of note that, in this experiment, all transactions were executed at the "user level", thus incurring overheads (e.g., compiling and optimizing the queries for incremental maintenance of the validating constraints) that can be avoided in a native (inside the database engine) implementation. Even with these overheads, the cost for maintaining the validating constraints is roughly 10 times smaller than the cost of performing the actual update operations.

8 Conclusion

In this paper we have proposed a simple formal model for XML-to-relational mapping schemes. Our framework is based on classes of mapping schemes defined by the languages used for mapping the documents, specifying relational constraints, and publishing the databases back as XML documents. We introduced a class of mappings called $\mathcal{XDS}$, which captures all mapping schemes in the literature. We proposed two natural notions of information preservation for mapping schemes, which ensure that queries and valid updates over the documents can be executed using these mappings. We showed that testing either property for $\mathcal{XDS}$ mappings is undecidable. Finally, we have proposed a lossless and validating $\mathcal{XDS}$ mapping scheme, and shown, through preliminary experimental evaluation that it performs well in practice.

We are currently working on designing *information preserving* transformations to derive lossless and validating mapping schemes from other such mappings. We have observed that virtually all mapping transformations proposed in the literature (e.g., inlining) can be modified to preserve the losslessness and validation properties in a simple way. We also hope that our simple formalization can be used by other researchers for studying other classes of mapping schemes, using other languages and possibly other notions of document (or database) equivalence.

Acknowledgments. This work was supported in part by grants from the Natural Science and Engineering Research Council of Canada. D. Barbosa was supported in part by an IBM PhD. Fellowship. J. Freire was supported in part by The National Science Foundation under grant EIA-0323604.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
2. D. Barbosa, J. Freire, and A. O. Mendelzon. Information Preservation in XML-to-Relational Mappings. Technical report, University of Toronto, 2004.
3. D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient Incremental Validation of XML Documents. In *International Conference on Data Engineering*, pages 671–682, Boston, MA, USA, 2004.

4. P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *Proceedings of the 18th International Conference on Data Engineering*, pages 64–75, February 26-March 1 2002.
5. B. Bouchou and M. Halfeld-Ferrari-Alvez. Updates and Incremental Validation of XML Documents. In *9th International Workshop on Database Programming Languages*, pages 216–232, Potsdam, Germany, September 6-8 2003.
6. A. Brüggemann-Klein and D. Wood. One-Unambiguous Regular Languages. *Information and Computation*, 142:182–206, 1998.
7. 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 *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 646–648, Cairo, Egypt, September 10-14 2000.
8. Y. Chen, S. Davidson, C. S. Hara, and Y. Zheng. RRXF: Redundancy reducing XML storage in relations. In *Proceedings of 29th International Conference on Very Large Data Bases*, pages 189–200, Berlin, Germany, September 9-12 2003.
9. S. Davidson, W. Fan, C. Hara, and J. Qin. Propagating XML Constraints to Relations. In *Proceedings of the 19th International Conference on Data Engineering*, Bangalore, India, March 5 - 8 2003.
10. D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proceedings of the 2003 ACM SIGMOD International Conference on on Management of Data*, pages 623–634, San Diego, California, June 9-12 2003.
11. A. Deutsch, M. Fernández, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 431–442, Philadelphia, Pennsylvania, USA, May 31-June 03 1999.
12. G. Dong and J. Su. Incremental Maintenance of Recursive Views Using Relational Calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.
13. M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.
14. D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), September 1999.
15. A. Gupta and I. S. Mumick, editors. *Materialized Views - Techniques, Implementations and Applications*. MIT Press, 1998.
16. R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL Query Translation Literature: the State of the Art and Open Problems. In *Proceedings of the First International XML Database Symposium*, pages 1–18, Berlin, Germany, September 8 2003.
17. D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
18. M. Mani and D. Lee. XML to Relational Conversion Using Theory of Regular Tree Grammars. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web. VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb. Revised Papers*, pages 81–103. Springer, 2002.
19. Y. Papakonstantinou and V. Vianu. Incremental Validation of XML Documents. In *Proceeedings of The 9th International Conference on Database Theory*, pages 47–63, Siena, Italy, January 8-10 2003.
20. S. Patnaik and N. Immerman. Dyn-FO: A Parallel, Dynamic Complexity Class. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 210–221, Minneapolis, Minnesota, May 24-26 1994.

21. M. Ramanath, J. Freire, J. R. Haritsa, and P. Roy. Searching for Efficient XML-to-Relational Mappings. In *Proceedings of the First International XML Database Symposium*, pages 19–36, Berlin, Germany, September 8 2003.
22. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 302–314, Edinburgh, Scotland, UK, September 7-10 1999.
23. D. Suciu. Query Decomposition and View Maintenance for Query Languages for Unstructured Data. In *Proceedings of 22th International Conference on Very Large Data Bases*, pages 227–238, Mumbai (Bombay), India, September 3-6 1996.
24. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 413 – 424, Santa Barbara, California, United States, 2001.
25. J. D. Ullman. The Interface Between Language Theory and Database Theory. In *Theoretical Studies in Computer Science*, pages 133–151. Academic Press, 1992.
26. J. D. Ullman and A. V. Gelder. Parallel Complexity of Logical Query Programs. *Algorithmica*, 3:5–42, 1988.
27. Extensible Markup Language (XML) 1.0 - Second Edition. W3C Recommendation, October 6 2000. Available at: `http://www.w3.org/TR/2000/REC-xml-20001006`.