Designing Information-Preserving Mapping Schemes for XML

Denilson Barbosa*
University of Calgary
denilson@cpsc.ucalgary.ca

Juliana Freire
University of Utah
juliana@cs.utah.edu

Alberto O. Mendelzon
University of Toronto
mendel@cs.toronto.edu

Abstract

An XML-to-relational mapping scheme consists of a procedure for *shredding* documents into relational databases, a procedure for *publishing* databases back as documents, and a set of constraints the databases must satisfy. In previous work, we defined two notions of information preservation for mapping schemes: *losslessness*, which guarantees that any document can be reconstructed from its corresponding database; and *validation*, which requires every legal database to correspond to a valid document. We also described one information-preserving mapping scheme, called Edge^{++}, and showed that, under reasonable assumptions, losslessness and validation are both undecidable. This leads to the question we study in this paper: how to design mapping schemes that are information-preserving. We propose to do it by starting with a scheme known to be information-preserving and applying to it equivalence-preserving transformations written in weakly recursive ILOG. We study an instance of this framework, the LILO algorithm, and show that it provides significant performance improvements over Edge^{++} and introduces constraints that are efficiently enforced in practice.

1 Introduction

In order to use relational engines for managing XML data, we need a *mapping scheme* providing a procedure for shredding the documents into relational databases, a procedure for publishing the databases as documents, and a set of constraints that those databases must satisfy. As with any other mapping strategy, it is important to study the information preservation properties of XML-to-relational mapping schemes in order to understand their suitability for a given application [26]. Although there is a rich literature on mapping schemes [6, 14, 17, 22, 31, 23, 33], to date little attention has been given to their information-preservation capabilities. In previous work [3], we defined *lossless* mapping schemes as those that allow the reconstruction of the original documents, and *validating* mapping schemes as those in which all legal database instances correspond to a valid XML document. We argued that while losslessness is enough for applications involving only queries over the documents, both losslessness *and* validation are required if the documents must conform to an XML schema and the application involves both queries and updates to the documents. We also described the Edge^{++} mapping scheme, in which both losslessness and validation are guaranteed by constraints in the relational schema.

This paper addresses the problem of *designing* information-preserving mapping schemes. Note that previous mapping design algorithms [6, 31] do not guarantee information preservation; moreover, since losslessness and validation are undecidable for a large class of mapping schemes [3], arbitrary design procedures cannot guarantee information preservation. We propose a sound framework that can serve as the basis for design algorithms: repeatedly applying equivalence-preserving transformations [26] to an initial mapping scheme *known* to be information-preserving. In this way, information preservation is guaranteed *by construction.* Our framework is extensible and allows any transformation that can be written in weakly recursive ILOG with stratified negation [20] (wrec-ILOG$^{\neg}$), which is powerful enough for expressing most of transformations proposed in the literature (e.g., [6, 29]). We also present an instance of our framework: the LILO (for Lossless Inlining, Lossless Outlining) algorithm, which uses Edge^{++} as starting point and defines several equivalence preserving transformations (some of which are extensions of transformations in the literature). Our experimental results show that LILO results in mapping schemes that outperform the previous Edge^{++} substantially.

Information Preservation in Mapping Schemes. The following example illustrates the need for information preservation in mapping schemes.

Example 1. Consider the schema in Figure 1(a), inspired by the Mondial Database[1], describing cities and countries. Each city is described by its name; the name

[*] Most of this work was done while this author was a Ph.D. Student at the University of Toronto.

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

[1] http://dbis.informatik.uni-goettingen.de/Mondial

$$\begin{aligned}
\Sigma = \{&mondial, cities, country, city,\\
&province, state, name, official\}\\
root =\ &mondial\\
\mathbf{R} = \{&mondial{\leftarrow}cities, country{*};\ \ cities{\leftarrow}city{*};\\
&city{\leftarrow}name, (province|state), official{+};\\
&country{\leftarrow}name, capital;\\
&name{\leftarrow}\#PCDATA; province{\leftarrow}\#PCDATA;\\
&state{\leftarrow}\#PCDATA; official{\leftarrow}\#PCDATA;\\
&capital{\leftarrow}\#PCDATA\}
\end{aligned}$$

(a)

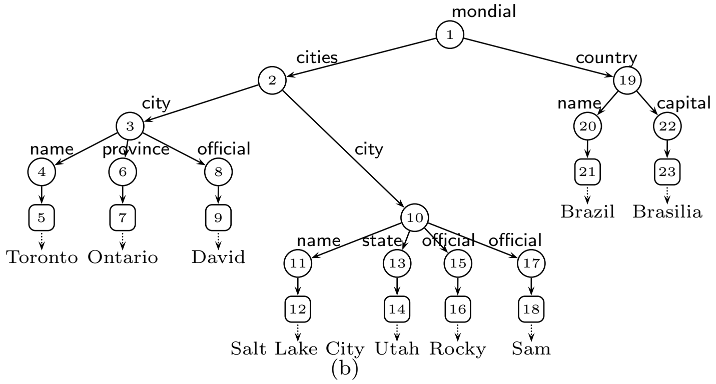


Figure 1: (a) A DTD for the Mondial document; (b) an XML document. Node ids are shown as numbers; elements are shown as circles and text nodes are shown as boxes. Elements labels and the values of the text nodes are also shown.

of the province or state where the city is located; and one or more government officials. Countries are described by their name and the name of their capitals. A typical relational schema derived from applying inlining techniques to this schema is as follows[2]:

```
city (cityId, name, ord, provinceId, stateId)
official (officialId, cityId, name, ord)
country (countryId, name, capital)
```

For the document in Figure 1(b), we would have the following database instance under this mapping:

```
city (1,'Toronto',1,'Ontario',NULL)
city (4,'Salt Lake City',2,NULL,'Utah')
official (2,1,'David',1)
official (5,4,'Rocky',1)
official (6,4,'Sam',2)
capital (7,'Brazil','Brasilia')
```

This mapping scheme is lossless as the original document can be reconstructed from the database (note that the ordering of many elements in the document is fixed by the schema and does not need to be stored). However, by applying the following *legal* update:

```
UPDATE city SET province='Utah'
WHERE name='Salt Lake City'
```

we arrive at database that no longer corresponds to a valid document. This anomaly is due to the fact that the mapping scheme does not preserve the semantics of the choice (|) construct, i.e., that each city is part of either a province or state, but not both. ■

In a lossless and validating mapping scheme, the relational schema is *equivalent* to the document schema; constraints in the relational schema prevent the anomalies above by disallowing updates resulting in databases that represent invalid documents. An alternative to using an information-preserving mapping scheme is to validate the document resulting from each update before committing to any changes. However, this approach has drawbacks. Note that testing whether even simple updates such as[3]:

```
update
delete //city[name='Toronto']/official[last()]
```

result in a valid document cannot be done statically. In this case, the legality of the update depends on the number of officials already associated with the city. Also, while we could test the legality of an update by reconstructing (a fragment of) the document, applying the update, and checking the validity of the result, doing so has many disadvantages. Validation is computationally expensive [27] and incremental validation [4, 28] techniques require considerable auxiliary information. Moreover, this approach requires special-purpose tools whose functionality overlaps with the DBMS's constraint checking facilities.

Outline and Contributions. We argue in this paper that it is feasible to augment the relational schemas in mapping schemes with constraints for ensuring the *validity* of the elements in the XML documents with respect to the content models in an XML schema. We start by discussing element and document validity, XML mapping schemes and information preservation (Section 2). We propose a sound and extensible framework for designing information preserving mapping schemes (Section 3), which consists of applying equivalence preserving transformations to an information-preserving mapping scheme. We show how a mapping scheme and an arbitrary transformation written in wrec-ILOG$^{\neg}$ can be rewritten as another mapping scheme in a mechanical way (Section 3.3). We discuss several equivalence preserving transformations that result in mapping schemes defining simpler relational constraints for ensuring element validity, and introduce the LILO algorithm (Section 4). We show that LILO provides significant performance improvements over Edge^{++} and that the constraints it introduces can be efficiently enforced in practice (Section 5). We conclude with a discussion (Section 6).

2 Definitions and Terminology

XML documents are modeled as ordered labeled trees whose nodes represent either elements or textual con-

[2] Primary keys are underlined while *nullable* columns are shown in italics; the ord attributes capture the element ordering.

[3] Using the syntax of [24].

tent, thus capturing the essential data representation components of XML [8]. We use τ, λ and ν to denote the *type*, the *label* and the *value* of nodes in the tree, respectively. More precisely, let $\mathcal{I}$, $\mathcal{D}$ be two disjoint, countably infinite sets of node ids and values:

Definition 1. *An* XML Document *is a tuple $\langle T, \lambda, \tau, \nu \rangle$, where T is an ordered tree whose nodes are elements of $\mathcal{I}$; $\tau : \mathcal{I} \rightarrow \{element, text\}$ assigns types to nodes in T, such that all text nodes are leafs in T and all internal nodes of T are elements; $\lambda : \mathcal{I} \rightarrow \mathcal{D}$ assigns labels to nodes in T such that the label of all text nodes is #PCDATA; and $\nu : \mathcal{I} \rightarrow \mathcal{D}$ assigns values to text nodes in T.*

For brevity, we will refer to subtrees rooted at *element* nodes simply as "elements". We denote by $\mathcal{X}$ the set of all XML documents.

Our focus in this work is on capturing the *element validity* constraint, which is identical in DTDs and XML Schemas and is checked as follows. First, each element in the document is a assigned a *type* (see below); an element of type t is said to be *valid* if its typed content (i.e., the string formed by concatenating the types of the children of the element) spells a word in a 1-unambiguous regular language [9]. One difference between DTDs and XML Schemas is that DTDs assign types to elements based on their labels only, while XML Schemas take the *context* in which the element appears in consideration as well. While our method handles both formalisms seamlessly, we restrict our discussion to DTDs for simplicity, and refer the reader to [3, 4] for details on handling context-dependent type specialization.

Definition 2. *An XML* DTD *is a triple $\langle \Sigma, r, \mathbf{R} \rangle$ where Σ is a set of element labels, $r \in \Sigma$ is a distinguished label and $\mathbf{R}$ is a mapping associating to each $a \in \Sigma$ a content model expressed as a 1-unambiguous regular expression over* $\Sigma \cup \{\#\text{PCDATA}\}$.

Figure 1(a) shows a DTD. The validity of a document D with respect to a DTD $X = \langle \Sigma, r, \mathbf{R} \rangle$ is done as follows. Let e be an element and $c_1, \ldots, c_n$ be its ordered children; the *content* of e is the string $\lambda(c_1) \cdots \lambda(c_n)$. We say that e is *valid* with respect to X if its content matches the regular expression associated with $\lambda(e)$ in $\mathbf{R}$. A document D is valid with respect to X, written $D \in L(X)$, if all of its elements are valid with respect to X and the label of its root element is r. Thus, the validation problem can be stated as: given D and X, is it the case that $D \in L(X)$?

2.1 XML-to-relational Mapping Schemes

Relational databases are modeled using two attribute domains: one for *surrogates* to node ids (e.g.,`cityId` in Example 1) and another for all other constants. Denote by $\mathfrak{I}$, $\mathfrak{D}$ two disjoint countably infinite sets of surrogates and constants, respectively. A *relational* schema is a set of relation schemes and constraints; each relation scheme R has a set (possibly empty) of attributes of domain $\mathfrak{I}$—called the *surrogate attributes of R*, and a set (possibly empty) of attributes of domain $\mathfrak{D}$. Instances are defined as customary [1, 25]. A *constraint* is expressed as a boolean query and is said to be violated if that query evaluates to true. A relational database instance is *legal* if it does not violate any constraint in its schema. We denote by $\mathcal{R}(S)$ the set of legal instances of S.

No meaning is assigned to node ids the document tree, nor to the surrogates used for representing them; furthermore, no relationship between node ids and surrogates is assumed either. That is, renaming node ids in Figure 1(b) does not yield a new document; similarly, renaming surrogates in the database in Example 1 does not create a new database. These properties are captured as follows:

Definition 3. *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 $D_1 \equiv_{\mathsf{X}} D_2$, if there exists an isomorphism $\phi : \mathcal{I} \rightarrow \mathcal{I}$ between T_1 and T_2 such that $\lambda_1(v) = \lambda_2(\phi(v))$, $\tau_1(v) = \tau_2(\phi(v))$, and $\nu_1(v) = \nu_2(\phi(v))$, for all $v \in T_1$.*

Definition 4. *Database instances I_1, I_2 are equivalent, written $I_1 \equiv_{\mathsf{R}} I_2$ if there is a bijection on $\mathfrak{I} \cup \mathfrak{D}$ that maps $\mathfrak{I}$ to $\mathfrak{I}$, is the identity on $\mathfrak{D}$, and transforms I_1 into I_2.*

The notion of database equivalence above has been called *OID equivalence* in object databases [1]. $[D]$ denotes the equivalence class of document D; that is $[D] = \{D' \in \mathcal{X} \mid D' \equiv_{\mathsf{X}} D\}$; similarly, $[I]$ denotes the equivalence class of database I.

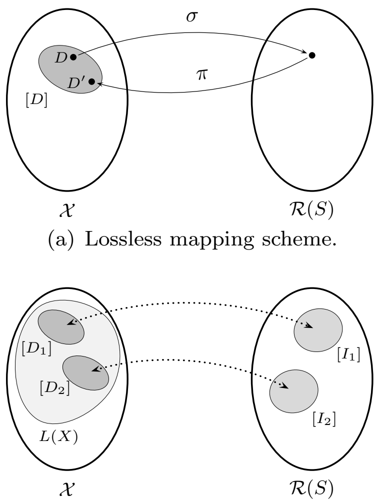
An XML-to-relational *mapping scheme* [3] is defined as a triple $\mu = (\sigma, \pi, S)$, where S is a relational schema; σ is a *mapping function* that assigns instances of S to XML documents; and π is a *publishing function* that assigns XML documents to instances of S.

Definition 5. *An XML-to-relational mapping scheme is a triple $\mu = (\sigma, \pi, S)$, where $\sigma : \mathcal{X} \rightarrow \mathcal{R}(S)$ is a partial function; $\pi : \mathcal{R}(S) \rightarrow \mathcal{X}$ is a total function; and the following hold: (1) for all $D_1, D_2 \in \mathcal{X}$ we have that $D_1 \equiv_{\mathsf{X}} D_2$ implies $\sigma(D_1) \equiv_{\mathsf{R}} \sigma(D_2)$; and (2) for all $I_1, I_2 \in \mathcal{R}(S)$ we have that $I_1 \equiv_{\mathsf{R}} I_2$ implies $\pi(I_1) \equiv_{\mathsf{X}} \pi(I_2)$.*

Defining σ as a partial function accommodates mapping schemes customized for a specific DTD (e.g., [6, 31]); on the other hand, defining π to be total ensures that any legal database *represents* (i.e., can be published as) a document. Conditions (1) and (2) ensure that both σ and π are *generic*: they map equivalent documents to equivalent databases and vice-versa.

2.2 The $\mathcal{XDS}$ Class of Mapping Schemes

A *class* of mapping schemes is defined by the languages used for specifying σ, S, and π; the expressive power of these languages determines what kinds



(a) Lossless mapping scheme.

(b) Lossless *and* validating mapping scheme.

Figure 2: Information preservation in mapping schemes.

of mappings belong to the class. The $\mathcal{XDS}$ class of mapping schemes [3] is defined as follows.

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.

The Constraint Language. Constraints are boolean programs in Datalog with stratified negation [1]. This language allows easy expression of standard relational constraints (e.g., functional dependencies and referential integrity), as well as graph connectivity—required for ensuring the database encodes a tree and element validity (Section 3.2).

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 [16] and XPERANTO [10].

$\mathcal{XDS}$ is, by design, a powerful mapping tool. In fact, $\mathcal{XDS}$ can express all mapping schemes that we are aware of in the literature.

2.3 Information Preservation in $\mathcal{XDS}$

For completeness, we revisit losslessness and validation here, and refer the reader to [3] for details. A mapping scheme is lossless if it allows the complete reconstruction of any (fragment of a) document from the database assigned to it (see Figure 2(a)):

Definition 6. *A mapping scheme* $\mu = (\sigma, \pi, S)$ *is* lossless *if for all* $D \in \mathrm{Dom}(\sigma)$*,* $\pi(\sigma(D)) \equiv_{\mathsf{X}} D$*.*

The following is easy to verify:

Proposition 1. $\mu = (\sigma, \pi, S)$ *is lossless if and only if* $\pi(\sigma(\cdot))$ *is the identity on equivalence classes in* $\mathrm{Dom}(\sigma)$*.*

Validation is defined in terms of a DTD X. A validating mapping scheme is one in which every legal database instance corresponds to a document in $L(X)$:

Definition 7. *A mapping scheme* $\mu = (\sigma, \pi, S)$ *is validating with respect to DTD* X *if* σ *is total on* $L(X)$*, and for all* $I \in \mathcal{R}(S)$*, there exists* $D \in L(X)$ *such that* $I \equiv_{\mathsf{R}} \sigma(D)$*.*

For a validating $\mu = (\sigma, \pi, S)$, every *successful* update on an instance of S results in a database that represents a valid document; thus, only permissible updates over the original document can be effected on its corresponding relational database. As discussed in Section 1, if μ is lossless, testing if an update is permissible can be done by materializing $\pi(\sigma(D)))$, effecting the update, and validating the resulting document, which is prohibitively expensive in most cases.

Losslessness and validation are orthogonal and one does not imply the other. Also, applications involving both queries and updates and in which documents must conform to a DTD, require mapping schemes that have both properties [3]. When a lossless and validating mapping scheme is used, queries and all permissible updates over the documents can be done using (SQL queries over) the databases.

Proposition 2. $\mu = (\sigma, \pi, S)$ *is lossless and validating with respect to DTD* X *if and only if* σ *and* π *are bijective, and* π *is the inverse of* σ *(up to equivalence).*

Proof. It is easy to verify that μ is both lossless and validating if σ and π are as above. For the other implication, note that since μ is validating with respect to X and both σ and π are generic (Definition 5), it follows that σ defines a bijection between equivalence classes of documents in $L(X)$ and database instances in $\mathcal{R}(S)$. A similar argument applies for π. Since μ is also lossless, Proposition 1 implies that σ and π are the inverse of each other up to equivalence. □

We say a mapping scheme is *information-preserving* if it is both lossless and validating with respect to a DTD understood from the context. Note that an information-preserving mapping scheme defines a bijection among equivalence classes of valid documents and legal relational databases, as shown in Figure 2(b).

3 Designing Information-Preserving Mapping Schemes

We now discuss a general and sound framework for designing information-preserving mapping schemes based on applying *structural transformations* [19, 26]

$$\begin{tikzcd}
X \arrow[r, shift left, "\sigma"] & S_0 \arrow[l, shift left, "\pi"] \arrow[r, shift left, "\alpha_1"] & S_1 \arrow[l, shift left, "\beta_1"] \arrow[r, shift left, "\alpha_2"] & \cdots \arrow[l, shift left, "\beta_2"] \arrow[r, shift left, "\alpha_k"] & S_k \arrow[l, shift left, "\beta_k"] \\
 & \mu_0 & \mu_1 & \cdots & \mu_k
\end{tikzcd}$$

Figure 3: Designing of an information-preserving mapping scheme: X is a DTD, μ_0 is the Edge^{++} mapping scheme, and each μ_i is the result of applying an information-preserving schema transformation.

to an existing schema. We start with an initial mapping scheme μ_0 that is known to be information-preserving, and subsequently transform it until the desired one μ_k, defined as

$$\mu_k = \big((\alpha_k \circ \alpha_{k-1} \circ \cdots \circ \alpha_1 \circ \sigma),\ (\pi \circ \beta_1 \circ \cdots \circ \beta_k),\ S_k\big)$$

is found, as illustrated by Figure 3. To make this framework concrete we must specify the languages for expressing the (1) mappings between XML documents and database instances, as well as the (2) mappings α_i, β_i between instances of different relational schemas. In our case, the mapping and publishing languages are those in $\mathcal{XDS}$ (Section 2.2), and the mappings between relational instances are given as "weakly recursive" ILOG programs with stratified negation (wrec-ILOG$^\neg$) [20]. Informally, this language corresponds to Datalog with stratified negation augmented with non-recursive "invention" rules that create new surrogates, and is powerful enough for the kinds of transformations we define.

Soundness. Information preservation in our framework is achieved is as follows. First, we always start with a mapping scheme $\mu_0 = (\sigma_0, \pi_0, S_0)$ that is both lossless and validating with respect to a DTD X; as we discuss below, this is the case if and only if X and S_0 are *equivalent* (Proposition 3). Second, *every* transformation applied to a mapping scheme $\mu_i, 0 < i < k$ results in μ_{i+1} whose relational schema is *equivalent* to that of μ_i. Such transformations are called *equivalence preserving* [26]. It follows that the resulting mapping scheme $\mu_k = (\sigma_k, \pi_k, S_k)$ is such that S_k is equivalent to X and thus information preserving.

In the remainder of this section, we relate classical notions of information preservation and relative capacity of schemas [19, 26] to our notions of losslessness and validation [3]. Then, we briefly describe the Edge^{++} mapping scheme, and give a procedure for deriving $\mu_{i+1} = (\sigma_{i+1}, \pi_{i+1}, S_{i+1})$ from $\mu_i = (\sigma_i, \pi_i, S_i)$ and a pair of transformations between S_i and S_{i+1} given as arbitrary wrec-ILOG$^\neg$ programs.

3.1 Dominance and Equivalence of Schemas

Equivalence of schemas S and T (within or across data models) is defined based on properties of the mappings between their instances. Let $\mathbf{I}(S)$ denote the set of all instances of S (e.g., valid documents if S is a DTD or legal database instances if S is a relational schema).

Let $\alpha : \mathbf{I}(S) \to \mathbf{I}(T)$ be a mapping given in some appropriate language. α is said to be *information-preserving* if it is *reversible*; that is, there exists $\beta : \mathbf{I}(T) \to \mathbf{I}(S)$ such that $\beta(\alpha(\cdot))$ is an equivalence relation in $\mathbf{I}(S)$. If this is the case, we say that T *dominates* S (via α, β), denoted $S \preceq T$. (We note that classical notions of schema dominance [19, 26] define $\beta(\alpha(\cdot))$ to be the identity on $\mathbf{I}(T)$; without loss of generality, we use an equivalence relation to account for renamings of element surrogates.)

If both α and β are total and bijective, then $S \preceq T$ via (α, β) and $T \preceq S$ (via β, α). In this case, we say that S and T are *equivalent*, denoted $S \equiv T$, and that β (resp. α) is an *equivalence preserving* transformation. Note that $\preceq$ is a transitive relation, while $\equiv$ is an equivalence relation.

Schema dominance and equivalence establish notions of "relative information capacity" between schemas. If $S \preceq T$, we say that T has *at least the information capacity of* S, since instances of S can represent any instance of T. Similarly, if $S \equiv T$, we say S and T have the *same information capacity.*

Equivalence of DTDs and Relational Schemas. In a mapping scheme $\mu = (\sigma, \pi, S)$, σ and π play the role of α and β above, respectively:

Proposition 3. *If X is a DTD and $\mu = (\sigma, \pi, S)$ is a mapping scheme such that $L(X) \subseteq \mathrm{Dom}(\sigma)$ then: (1) μ is lossless if and only if $X \preceq S$; (2) μ is both lossless and validating with respect to X if and only if $X \equiv S$.*

Proof. (1) follows from Proposition 1, the fact that $L(X) \subseteq \mathrm{Dom}(\sigma)$, and the definition of $\preceq$. (2) follows from Proposition 2 and the definition of $\equiv$. □

Since losslessness and validation are undecidable [3], it follows that:

Corollary 1. *Equivalence of DTDs and relational schemas is undecidable.*

3.2 The Edge^{++} Mapping Scheme

We now describe Edge^{++}, which is used as the initial mapping scheme in our framework, and show it is information-preserving. Edge^{++} extends the Edge mapping scheme [17] (which is lossless) with constraints for ensuring validation with respect to a DTD $X = \langle \Sigma, r, \mathbf{R} \rangle$. Edge^{++} contains additional relations for storing the transition functions of the DFAs of the regular expressions in $\mathbf{R}$, and constraints that check the validity of the elements by simulating the appropriate DFAs on their content.

The Edge^{++} Relational Schema. Recall that $\mathcal{I}$ is the domain of surrogates and $\mathcal{D}$ is the domain of constants. Let $\mathcal{I}' = \mathcal{I} \cup \{\#\}, \# \notin \mathcal{I}$ (the symbol # will be used for marking elements that have no children); let $\mathcal{Q} \subseteq \mathcal{D}$ be a set of surrogates for DFA states; let $\mathcal{T} \subseteq \mathcal{D}$ be a set of surrogates for element types (i.e., symbols in Σ); and let $\mathcal{B} \subseteq \mathcal{D}$ denote the set of boolean constants. Edge^{++} mapping schemes use the following relational schema (the primary keys of all relations are underlined):

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

The **Edge** and **Value** relations store all edges between elements and between elements and text nodes in tree, respectively. Unlike in the Edge mapping, the ordering of the nodes in the document is captured by the *successor* relation: for each element e whose content model is not #PCDATA, we add a tuple (s_e, s_f, s_l) to **FLC** (which stands for "first and last children") consisting of the surrogates of e, and its first and last children; if e has no content (i.e., no children), we add a tuple $(s_e, \#, \#)$ to **FLC**. The **ILS** ("immediate left sibling") relation contains tuples with surrogates of consecutive nodes in the document. Using the successor relation instead of the ordinals of tree nodes was motivated by efficiency reasons, to avoid the reordering all nodes after each update [4]. The **Type** relation contains the types of each element in the document. Finally, **Transition** stores the transition functions of the automata that correspond to the content models in the DTD.

Constraints. Two kinds of constraints are defined in Edge^{++}. The *structural constraints* ensure that every instance of S encodes an XML document (as in Definition 1); to do that, we define constraints for ensuring the database encodes a tree, the ordering of the elements is consistent, etc. These properties are easily encoded as boolean queries in Datalog with stratified negation. The *validation constraints* ensure that every legal instance of S encodes a valid document; to do that, each DTD rule $t_i \leftarrow r_i$ is translated into the following Datalog program:

$$
\begin{aligned}
\mathbf{reach}_{t_i}(p, \#, s) &:- \mathbf{FLC}(p, \#, \#), \mathbf{Type}(p, t_i), && (1)\\
&\qquad \mathbf{Transition}(t_i, q_0, \varepsilon, s, \_)\\
\mathbf{reach}_{t_i}(p, c, s) &:- \mathbf{Edge}(p, c, x), \mathbf{FLC}(p, c, \_), \mathbf{Type}(p, t_i), && (2)\\
&\qquad \mathbf{Transition}(t_i, q_0, x, s, \_)\\
\mathbf{reach}_{t_i}(p, c, s) &:- \mathbf{reach}_{t_i}(p, x, y), \mathbf{ILS}(x, c), \mathbf{Type}(p, t_i), && (3)\\
&\qquad \mathbf{Edge}(p, c, w), \mathbf{Transition}(t_i, y, w, s, \_)\\
\mathbf{accept}_{t_i}(p) &:- \mathbf{reach}_{t_i}(p, c, s), \mathbf{FLC}(p, \_, c), \mathbf{Type}(p, t_i),\\
&\qquad \mathbf{Transition}(t_i, \_, \_, s, \mathit{true})\\
\mathbf{invalid}_{t_i} &:- \mathbf{FLC}(p, \_, \_), \neg\mathbf{accept}_{t_i}(p)
\end{aligned}
$$

Note that $\mathbf{reach}_{t_i}$ simulates the DFA corresponding to r_i on the content of an element p, starting[4] with rule (1) or (2) depending on whether the element has any content; rule (3) advances the DFA to the next child of p if an appropriate transition is defined. Thus, $\mathbf{accept}_{t_i}$ computes all valid elements (i.e., every element p for which an accepting state s is reached after visiting its last child c); $\mathbf{invalid}_{t_i}$ evaluates to true if and only if there is an element of type t_i that is invalid.

$$\begin{tikzcd}
X \arrow[r, bend left, "\sigma"] \arrow[rr, bend left=45, dotted, "\sigma'"] & S \arrow[l, bend left, "\pi"] \arrow[r, bend left, "\alpha"] & S' \arrow[l, bend left, "\beta"] \arrow[ll, bend left=45, dotted, "\pi'"]
\end{tikzcd}$$

Figure 4: Deriving a new mapping scheme.

The Mapping and Publishing Functions. Both σ and π in Edge^{++} are straightforward; due to space constraints, we briefly describe how they work and refer the reader to [3] for details. The mapping function σ creates a database instance by iterating over a stream of elements, according to the global document ordering; π, on the other hand, reconstructs the ordering of the elements using the `order by` clause of XQuery on **FLC** and **ILS**. Note that since σ is defined for a specific DTD, the mapping of **Transition** and **Type** can be easily hard-coded.

Proposition 4. *If $\mu = (\sigma, \pi, S)$ is the Edge^{++} mapping scheme for a DTD X, then μ is both lossless and validating with respect to X.*

3.3 Rewriting Mappings

Conceptually, a mapping scheme $\mu_k = (\sigma_k, \pi_k, S_k)$ in our framework stores a document D by applying an initial mapping function σ followed by k wrec-ILOG$^\neg$ programs $\alpha_1, \ldots, \alpha_k$ that transform $\sigma(D)$ into a final database instance $I \in \mathcal{R}(S_k)$. Similarly, μ_k publishes each instance $I \in \mathcal{R}(S_k)$ by reversing each transformation until arriving at the original $\sigma(D)$, to which π can be applied, as depicted in Figure 3. In the remainder of this section, we describe a much more efficient way of implementing μ_k, which consists of "compiling" the wrec-ILOG$^\neg$ programs directly into the mapping and publishing functions σ_k and π_k.

More precisely, let $\mu = (\sigma, \pi, S)$ be a mapping scheme in $\mathcal{XDS}$, and let $\alpha : \mathcal{R}(S) \rightarrow \mathcal{R}(S')$ and $\beta : \mathcal{R}(S') \rightarrow \mathcal{R}(S)$ be arbitrary wrec-ILOG$^\neg$ programs. We derive $\mu' = (\sigma', \pi', S')$ from σ, π, α and β (Figure 4) as follows. In summary, σ' works in two steps: first, it "materializes" $\sigma(D)$ as an intermediate XML document and then applies α, rewritten as a mapping expression. On the other hand, π' applies β (rewritten as a publishing function) to an instance of S', resulting in a "canonical view" of an instance of S, to which π can be applied unchanged.

Recall that wrec-ILOG$^\neg$ programs are ILOG programs with stratified negation without recursive invention of surrogates. We assume α, β are in normal form; that is, neither program defines cascading invention of new surrogates[5], and they can be stratified in a way that all invention rules occur *after* all non-invention

[4] The constant q_0 denotes the starting state of the DFA.

[5] A program has cascading invention of surrogates if there are two invention rules r_i, r_j such that the surrogates generated for r_i are "used" for generating surrogates for r_j.

rules. We note that for every wrec-ILOG$^\neg$ program there is an equivalent one in normal form [20].

Defining σ'. Without loss of generality, we assume that each mapping expression in σ populates a single relation in S, and that each relation in S is populated by a single mapping expression in σ. (We can replace each procedure p populating relations $r_1, \ldots, r_n$ by n "copies" of p such that copy p_i populates relation r_i only. Also, we can replace n mapping expressions $p_1, \ldots, p_n$ where each populate relation r by a new procedure p' and n functions $f_1, \ldots, f_n$, such that each f_i return the list of tuples inserted by p_i and p' inserts into r the "union" of the results of $f_1, \ldots, f_n$.)

The first step of σ' materializes an "instance" of S (which we refer to as I_S) as follows. Let $R_1, \ldots, R_n$ be the relation names in S and $p_1, \ldots, p_n$ be the mapping expressions in σ that populate them; we convert each p_i into a function f_i that returns the sequence of "tuples" that would be inserted in R_i. The result of each f_i is sorted lexicographically and duplicate tuples are removed. For the second step of σ', we convert α into a mapping expression that takes I_S and produces the desired instance of S'. Let $r_1, \ldots, r_i, \ldots, r_n$ be the rules in α stratified in a way that all invention rules are $r_{i+1}, \ldots, r_n$. We define α as two standard XQuery programs f_P and f_Q: f_P computes $P = r_1, \ldots, r_i$ on I_S using the conventional algorithm for fixpoint semantics for Datalog [1]; f_Q is applied to the result of f_P and computes $Q = r_{i+1}, \ldots, r_n$.

Defining f_P. f_P is a recursive function that takes a "database instance" d (represented as an XML document) as input, and computes a new "instance" d' (also represented as an XML document) that contains a copy of d and the results of the functions that compute the rules in P. If, after an iteration of f_P, d and d' are not lexicographically identical, we iterate again using d' as input; we stop when we reach a fixpoint. The translation of the rules in P is as follows.

A rule $r_i : A(\bar{x}) \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n)$ defining a conjunctive query where each B_j appears positively is implemented as a function f_i using nested loops to iterate over the cartesian product of relations $B_1, \ldots, B_n$, thus computing all possible valuations to the variables in r_i, and returns the "tuples" that satisfy the conjunctive query in the body of the rule. Negation is dealt with in the usual way for Datalog with stratified negation: $\neg R(\bar{x})$ holds if and only if $\bar{x}$ is in the active domain but not in R. A set of rules

$$A(\bar{x}) \leftarrow \cdots, \quad \ldots, \quad A(\bar{x}) \leftarrow \cdots$$

computing a union is implemented as a separate function for each rule (as above); the "union" is done by concatenating with duplicate elimination and lexicographic sorting the individual "relations" (i.e., sequences of tuples) returned by each function.

Defining f_Q. Since we do not allow recursive generation of surrogates, we treat each invention rule $A(*, \bar{x}) \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n)$ like a standard non-recursive rule as in f_P, except that an invented surrogate is added to each tuple in the result set, computed by a *Skolem* function $f_A(\bar{x})$. Note that the mapping expressions that populate an instance of S' from the result of f_Q are straightforward.

Edge$_0$

pid	*eid*	*label*
1	19	`country`
19	20	`name`
19	22	`capital`

FLC$_0$

pid	*first*	*last*
19	20	22

ILS$_0$

left	*right*
20	22

Value$_0$

eid	*value*
20	`Brazil`
22	`Brasilia`

(a) Edge^{++} mapping of the Mondial XML document.

Country$_1$

country	*name*	*capital*
19	20	22

Value$_1$

country	*value*
20	`Brazil`
22	`Brasilia`

(b) Inlining element ids.

Country$_2$

country	*name*	*capital*
19	`Brazil`	`Brasilia`

(c) Inlining element content.

Figure 5: Applying mapping scheme transformations.

Defining π'. We obtain π' by converting β into XQuery functions as discussed above; note that, on a "canonical view" of an instance of S', these functions compute a "canonical view" of the corresponding instance of S, to which we can apply π unchanged.

4 The LILO Algorithm

In this section we describe LILO (Lossless Inlining, Lossless Outlining), a mapping scheme design algorithm based on the framework discussed in Section 3. LILO uses Edge^{++} as its initial mapping scheme and defines several equivalence-preserving transformations, some of which are similar to those defined in the literature. We start with an example that illustrates the use of two transformations to the Edge^{++} mapping of the document in Figure 1(b). Next, we discuss in detail the transformations used in LILO and argue why they are equivalence-preserving. Finally, the LILO algorithm is discussed.

Example 2. Recall the fragment of the mondial database shown in Figure 1(b) and its DTD in Figure 1(a). In this example, we focus on country elements and the constraints defined for them. For clarity, we omit in the figures all relations not involved in the discussion; also, we use subscripts to distinguish relations with the same name in different schemas (e.g., **Value**$_1$ is a relation in S_1).

Edge^{++} Mapping. Figure 5(a) shows the Edge^{++} mapping of the country element; we refer to its relational schema as S_0 in the remainder of the example. In S_0, we have a recursive constraint to check that each element e labeled country has has exactly two children c_1, c_2; c_1 is labeled name; c_2 is labeled capital; and $c1$ precedes $c2$. Moreover, S_0 also has a structural constraint requiring both c_1, c_2 to have a matching tuple in the **Value** relation.

Inlining Elements. Consider a schema S_1 that adds to S_0 a relation $\mathbf{Country_1}(\underline{country}, name, capital)$ for inlining country elements and their children, and constraints for enforcing the functional dependencies $name \rightarrow country$ and $capital \rightarrow country$, for ensuring that each name and capital element has a single country element as its parent in the tree. The constraints for validating name and capital elements (Section 3.2) are also modified to use $\mathbf{Country}_1$. Intuitively, a mapping scheme defined on S_1 has one advantage over Edge^{++}: the validation constraints for country elements can be specified directly using SQL [13], and thus enforced efficiently by relational database engines. The mapping $\alpha_1 : \mathcal{R}(S_0) \rightarrow \mathcal{R}(S_1)$ is:

$$\begin{aligned}
\mathbf{Diff}(e) &:- \mathbf{Edge_0}(\_, e, \texttt{'country'}) \\
\mathbf{Diff}(e) &:- \mathbf{Edge_0}(\_, e, \texttt{'capital'}) \\
\mathbf{Diff}(e) &:- \mathbf{Edge_0}(\_, c, \texttt{'country'}), \mathbf{Edge_0}(c, e, \texttt{'name'}) \\
\mathbf{Country_1}(e, n, c) &:- \mathbf{Edge_0}(e, n, \texttt{'name'}), \mathbf{Edge_0}(e, c, \texttt{'capital'}) \\
\mathbf{Edge_1}(e, c, l) &:- \mathbf{Edge_0}(e, c, l), \neg\mathbf{Diff}(e) \\
\mathbf{FLC_1}(p, f, l) &:- \mathbf{FLC_0}(p, f, l), \neg\mathbf{Diff}(p) \\
\mathbf{ILS_1}(l, r) &:- \mathbf{ILS_0}(l, r), \neg\mathbf{Diff}(l) \\
\mathbf{Value_1}(e, v) &:- \mathbf{Value_0}(e, v)
\end{aligned}$$

Diff computes all elements that are mapped into $\mathbf{Country}_1$ instead of $\mathbf{Edge}_1$, $\mathbf{FLC}_1$ and $\mathbf{ILS}_1$. Figure 5(b) shows the resulting database instance. The instance of S_0 is recovered by $\beta_1 : \mathcal{R}(S_1) \rightarrow \mathcal{R}(S_0)$:

$$\begin{aligned}
\mathbf{Edge_0}(e, c, l) &:- \mathbf{Edge_1}(e, c, l) \\
\mathbf{Edge_0}(e, c, l) &:- \mathbf{Edge_1}(e, \_, \texttt{'country'}), \mathbf{Country}(c, \_, \_), \\
&\quad\; l = \texttt{'country'} \\
\mathbf{Edge_0}(e, c, l) &:- \mathbf{Country_1}(e, c, \_), l = \texttt{'name'} \\
\mathbf{Edge_0}(e, c, l) &:- \mathbf{Country_1}(e, \_, c), l = \texttt{'capital'} \\
\mathbf{FLC_0}(p, f, l) &:- \mathbf{FLC_1}(p, f, l) \\
\mathbf{FLC_0}(p, f, l) &:- \mathbf{Country}(p, f, l) \\
\mathbf{ILS_0}(l, r) &:- \mathbf{ILS_1}(l, r) \\
\mathbf{ILS_0}(l, r) &:- \mathbf{Country_1}(\_, l, r) \\
\mathbf{Value_0}(e, v) &:- \mathbf{Value_1}(e, v)
\end{aligned}$$

It is easy to see that $S_0 \equiv S_1$: $S_0 \preceq S_1$ via (α_1, β_1) and that $S_1 \preceq S_0$ via (β_1, α_1)

Inlining Textual Content. We can further modify S_1 by storing the actual names and capitals of the countries instead of their surrogates; in this new schema S_2, there is no need to enforce the functional dependencies in S_1, nor the validating constraints for name and capital elements. The validation constraints for name elements that are not children of country are not affected by either transformation. Furthermore, no joins are required for finding names or capitals of the countries when processing queries. Instances of S_2 are computed by $\alpha_2 : \mathcal{R}(S_1) \rightarrow \mathcal{R}(S_2)$:

$$\begin{aligned}
\mathbf{Diff}(e) &:- \mathbf{Country_1}(p, e, \_), \mathbf{Value_1}(e, \_) \\
\mathbf{Diff}(e) &:- \mathbf{Country_1}(p, \_, e), \mathbf{Value_1}(e, \_) \\
\mathbf{Edge_2}(e, c, l) &:- \mathbf{Edge_1}(e, c, l) \\
\mathbf{FLC_2}(p, f, l) &:- \mathbf{FLC_1}(p, f, l) \\
\mathbf{ILS_2}(l, r) &:- \mathbf{ILS_1}(l, r) \\
\mathbf{Country_2}(e, n, c) &:- \mathbf{Country_1}(e, v_1, v_2), \\
&\quad\; \mathbf{Value_1}(v_1, n), \mathbf{Value_1}(v_2, c) \\
\mathbf{Value_2}(e, v) &:- \mathbf{Value_1}(e, v), \neg\mathbf{Diff}(e)
\end{aligned}$$

Figure 5(c) shows the resulting instance. To reconstruct the instances of S_1, $\beta_2 : \mathcal{R}(S_2) \rightarrow \mathcal{R}(S_1)$ uses invention rules (marked with *) to replace the surrogates lost by α_2:

$$\begin{aligned}
\mathbf{Edge_1}(e, c, l) &:- \mathbf{Edge_2}(e, c, l) \\
\mathbf{FLC_1}(p, f, l) &:- \mathbf{FLC_2}(p, f, l) \\
\mathbf{ILS_1}(l, r) &:- \mathbf{ILS_2}(l, r) \\
\mathbf{PName}(*, e, n) &:- \mathbf{Country_2}(e, n, \_) && (*) \\
\mathbf{PCapital}(*, e, c) &:- \mathbf{Country_2}(e, \_, c) && (*) \\
\mathbf{Country_1}(e, n, c) &:- \mathbf{PName}(n, e, \_), \mathbf{PCapital}(c, e, \_) \\
\mathbf{Value_1}(e, v) &:- \mathbf{Value_2}(e, v) \\
\mathbf{Value_1}(e, v) &:- \mathbf{PName}(e, v, \_) \\
\mathbf{Value_1}(e, v) &:- \mathbf{PCapital}(e, \_, v)
\end{aligned}$$

Again, note that $S_2 \equiv S_1$. Thus, the mapping scheme resulting from applying the two transformations above to Edge^{++} is information-preserving. ■

4.1 Equivalence Preserving Transformations

We now present the equivalence preserving transformations used in LILO. Our goal in defining them is twofold: reduce the number of joins required for navigating the databases; and replace the validation constraints defined by Edge^{++} by simpler ones that can be expressed in SQL. In the interest of space and for clarity of exposition, we omit the wrec-ILOG$^{\neg}$ programs in the discussion below; instead, we informally describe each transformation and argue why they are equivalence preserving.

Recall that the Edge^{++} mapping scheme for a DTD X includes validating constraints corresponding to each rule $t_i \leftarrow r_i$ in X. Since the transformations we define below work by replacing some of these constraints by simpler ones, they can be intuitively described as *changes* to the corresponding DTD rules. For example, suppose $\mu = (\sigma, \pi, S)$ is the Edge^{++} mapping scheme for the DTD in Figure 1(a) and we use inlining on *name* elements that are children of *city* elements. Intuitively, this can be viewed as replacing $city \leftarrow name, (province|state), official+$ in the DTD by $city^1 \leftarrow (province|state), official+$, and modifying σ (resp., π) for storing (resp., fetching) the name elements in a separate relation. We also use transformations that *mix* the content of different elements.

For example, we can nest (defined below) the content of cities elements as children of mondial, thus replacing *mondial* $\leftarrow$ *cities*, *country*$*$ and *cities* $\leftarrow$ *city*$*$ by *mondial* $\leftarrow$ *city*$*$, *country*$*$. In this case, we change σ to "skip" the cities element and π to "introduce" that element back as a child of the mondial element.

All transformation in LILO are equivalence-preserving for two reasons. First, every DTD rule $t \leftarrow r$ to which a transformation applies is replaced by a "simpler" $t \leftarrow r'$ and relational constraints that capture the semantics of part of the original content model r. Second, every regular expression r' that is "introduced" by LILO transformations is 1-unambiguous. Note that f $r = r_1, \ldots, r_j, \ldots, r_n$ is a 1-unambiguous regular expression and $w \in L(r)$ then there exist *unique* $w_1, \ldots, w_n$ such that $w_i \in L(r_i)$, $1 \leq i \leq n$, and $w = w_1 \cdots w_n$ [9]. (Note some w_i may be the empty string.) Thus, even when we mix the contents of different elements, we can always distinguish them afterward to reconstruct the original document.

Notation. In the remainder of this section, r, s denote regular expressions over $\Sigma \cup \#PCDATA$ and $a \in \Sigma$ denotes an element label. $\mu = (\sigma, \pi, S)$ and $\mu' = (\sigma', \pi', S')$ denote, respectively, the input and the output of each transformation. Recall that each rule $t_i \leftarrow r_i$ in the DTD becomes a constraint in Edge^{++} involving the simulation of the appropriate DFA on the children of elements of type t_i; we call such constraint an *edge* constraint and denote it by $t_i \overset{\text{R}}{\leftarrow} r_i$ below.

The relational schema of the input mapping scheme is of the form $S = \{E_1, \ldots, E_n, M_1, \ldots, M_k, \Gamma\}$, where $E_1, \ldots, E_n$ are *edge* relations as defined by Edge^{++}; $M_1, \ldots, M_k$ are *mapped* relations (resulting from the application of a transformation); and $\Gamma = \Gamma^E \cup \Gamma^M$, is the set of constraints in S where Γ^E contains all edge constraints, and Γ^M is a set of *mapped* constraints (resulting from the application of a transformation). Each transformation creates or modifies one or more mapped relations, and replaces one constraint in Γ^E by other constraints in Γ in a way that the resulting relational schema is equivalent to S.

4.1.1 Inlining

Inlining [6, 31] consists of using the same relation for storing an element together with one or more of its children.

Inline_Element(t, r_j, M), where $t \overset{\text{R}}{\leftarrow} r$ is an edge constraint, $r = r_1, \ldots, r_j, \ldots, r_n$ and r_j is either a or $a?$, results in using the mapped relation M to store pairs of surrogates of elements of types t and a, and applies only if $s = r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_n$ is 1-unambiguous. The schema for M is $(\underline{t}, c_1, \ldots, c_k, a)$ (the primary key is underlined), where $c_1, \ldots, c_k$, $k \geq 0$, are columns added by previous applications of inlining on type t. We replace the constraint $t \overset{\text{R}}{\leftarrow} r$ by $t \overset{\text{R}}{\leftarrow} s$, and add a mapped constraint for checking that values of a in M are unique (that is, the FD $a \rightarrow t$ holds). Since μ' stores the surrogates for a elements in M, the constraint for validating a elements is modified to use M (recall the discussion in Example 2). Finally, if a elements are mandatory (i.e., $r_j = a$), we declare a to be not null in M.

Mapping instances of S into instances of S' is done by copying the surrogates of inlined elements (of types $c_1, \ldots, c_k, a$) into M, while mapping all other children of t elements into the usual edge relations in S' (as illustrated in Example 2). The mapping in the other direction is done by copying the surrogates of s elements into the edge relations in S. Since $s = r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_n$ is 1-unambiguous, we can recover the original element ordering in an instance of S: we know that each a element must occur *after* every element whose type appears in $r_1, \ldots, r_{j-1}$ and *before* every element whose type appears in $r_{j+1}, \ldots, r_n$.

Inline_Union(t, r_j, M) is a special case of inlining that applies when $r_j = (s_1| \ldots |s_n)$, and each s_i is either a_i or $a_i?$. We add columns $a_1, \ldots, a_n$ to M, and add the following constraints. First, we enforce that at most one of $a_1, \ldots, a_n$ is not null in each tuple in M; also, if no s_i is of the form $a_i?$, then one of these columns must be not null. Both constraints can be easily written as SQL "check" clauses.

Note that the update anomaly discussed in Example 1 is eliminated by defining the constraints discussed above in that relational schema.

Inlining Textual Content. After inlining elements whose content model is #PCDATA, the mapped relation can be used to store their content directly:

Inline_Cdata(a, M), where a is the type of an inlined element in the mapped relation M, replaces the column for storing surrogates of a elements by a column for storing their textual content. Also, this transformation eliminates the constraint $a \overset{\text{R}}{\leftarrow}$ #PCDATA in S'. As illustrated in Example 2, the mapping between instances of S and S' amounts to copying the tuples in **Value** corresponding to a elements into M, ignoring their surrogates. The reverse mapping invents new surrogates for each inlined element.

4.1.2 Nesting Elements

This transformation eliminates some elements in the initial database by nesting their content within their parents.

Nest(t, a) applies when $t \overset{\text{R}}{\leftarrow} r$ and $a \overset{\text{R}}{\leftarrow} r'$ are such that $r = r_1, \ldots, a, \ldots, r_n$, and $s = r_1, \ldots, r', \ldots, r_n$ is 1-unambiguous. Nesting does not add new mapped relations to S'; instead, it replaces the constraints $t \overset{\text{R}}{\leftarrow} r$ and $a \overset{\text{R}}{\leftarrow} r'$ in S by $t \overset{\text{R}}{\leftarrow} s$ in S'. The wrec-ILOG$^\neg$ programs for mapping instances of S and S' are straightforward, and since s is 1-unambiguous, we can distinguish the children of a elements among the children of t elements in an instance of S'.

4.1.3 Outlining Elements

This transformation is the opposite of inlining and results in using separate mapped relations for storing

```
Input : a DTD  $X = \langle \Sigma, r, \mathbf{R} \rangle$
Output : a mapping scheme $\mu = (\sigma, \pi, S)$
 1. Let $\mu$ be the Edge$^{++}$ mapping scheme for $X$
 2. For each $t \in \Sigma$, create a mapped relation $M_t$ in $S$
 3. Let $t = r$
 4. Let $r = r_1 \ldots, r_n$be the content model associated
    with $t$ in $\mathbf{R}$
 5. For each $r_i, 1 \leq i \leq n$, apply $\mathbf{Nest}(t, r_i)$,
    $\mathbf{Inline\_Element}(t, r_i, M_t)$, $\mathbf{Inline\_Cdata}(r_i, M_t)$,
    $\mathbf{Inline\_Union}(t, r_i, M_t)$, $\mathbf{Outline}(t, r_i)$,
    $\mathbf{Outline\_Union}(t, r_i)$ whenever possible
    and in this order
 6. If a transformation creates a new type $t'$,
    add a mapped relation $M_{t'}$ to $S$
 7. For each element type $t_i$ occurring in $r$,
    let $t = t_i$ and repeat from step 3 until no changes
    are made to $S$
 8. Remove from $S$ all relations are not used by $\sigma$ or $\pi$
 9. Return the resulting mapping scheme
```

Figure 6: LILO algorithm for designing information-preserving mapping schemes.

the content of elements of the same type. For instance, one could use this transformation to store city officials separately from the other children of city elements. Intuitively, this can be seen as replacing the constraint $city \overset{\text{R}}{\leftarrow} name, (province|state), official+$ by $city^1 \overset{\text{R}}{\leftarrow} name, (province|state)$ and $city^2 \overset{\text{R}}{\leftarrow} official+$, which could be further transformed independently.

$\mathbf{Outline}(t, r_j)$ applies when $t \overset{\text{R}}{\leftarrow} r_1, \ldots, r_j, \ldots, r_m$ is an edge constraint, the participation of r_j is either $*$ or $+$, and $s = r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_n$ is 1-unambiguous. In general, outlining does not introduce new mapped relations; instead it replaces the original edge constraint by $t \overset{\text{R}}{\leftarrow} s$ and $t' \overset{\text{R}}{\leftarrow} r_j$. The wrec-ILOG$^\neg$ programs for outlining map part of the content of each t element into the content of a new t' element (in the forward direction) and merge these pieces of content back (in the backward direction).

When r_j is either $a*$ or $a+$ we can store just the parent-child relationship, ignoring labels (since they are all identical); if $r_j = a*$, we can also drop $t' \overset{\text{R}}{\leftarrow} a*$. The ordering of the outlined elements, however, must still be preserved (using the **FLC** and **ILS** relations).

$\mathbf{Outline\_Union}(t, r_j)$ is a special case that applies when $r_j = (u_1| \ldots |u_k)$ and results in replacing the original edge constraint $t \overset{\text{R}}{\leftarrow} r_j$ by the following:

$$\begin{aligned} t_1 &\overset{\text{R}}{\leftarrow} r_1, \ldots, r_{j-1}, u_1, r_{j+1}, \ldots, r_n \\ &\vdots \\ t_k &\overset{\text{R}}{\leftarrow} r_1, \ldots, r_{j-1}, u_k, r_{j+1}, \ldots, r_n \end{aligned}$$

It is easy to see that all regular expressions above are 1-unambiguous. One issue with this transformation is that the type of an element may change after an update. Thus, whenever an update on element e of type t_i violates the edge constraint of t_i, we must check whether the new content of e satisfies the constraint of some other type t_j, and if so, update the type of e.

Operation	512KB	4MB	32MB	256MB	2GB
Edge^{++} mapping scheme					
Insertion	7.72	15.87	17.70	20.48	20.45
Deletion	4.45	8.78	10.43	11.60	11.89
Q3	1.12	5.74	334.21	—	—
Q17	0.20	0.23	0.35	1.5	8
LILO mapping scheme					
Insertion	4.73	8.55	9.58	9.99	11.38
Deletion	3.53	6.41	7.48	8.16	8.76
Q3	0.09	0.12	0.18	1.47	50.41
Q17	0.03	0.03	0.03	0.09	0.59

Table 1: Experimental results; all times shown in milliseconds. For insertion and deletions, the time includes both modifying the database and checking the constraints.

4.2 The LILO Algorithm

The LILO algorithm is given in Figure 6. LILO visits each type t once, and after t is visited, its validating constraint is either mapped by some transformation or left unchanged. The order in which the transformations are attempted reflect the criteria discussed in the beginning of Section 4.1: reducing the number of joins required for navigating the document (achieved mostly by inlining), and simplifying the validation constraints introduced by Edge^{++}.

Theorem 1. *Given a DTD X, LILO always terminates and produces a mapping scheme that is information-preserving with respect to X.*

Proof. LILO always terminates because each rule in the DTD is visited only once, and the total length (i.e., number of symbols) of all rules in Γ^E after each iteration of LILO is strictly smaller than at the beginning of that iteration. The resulting mapping scheme is information-preserving since Edge^{++} is information-preserving, and all transformation used by LILO are equivalence preserving (Section 3.1). □

5 Experimental Evaluation

Table 1 presents experimental results comparing the behavior of Edge^{++} and LILO mapping schemes for processing updates and queries, using XMark [30] documents ranging from 512KB to 2GB.

We used DB2 V8.1 on a Pentium-4 2.5GHz (1.5GB of RAM) running Linux 2.4; we limited DB2's buffer to 45MB, to avoid having large portions of the document always in main memory. This is the only parameter we tune; furthermore, the only indices in each database are those created automatically for the primary keys of each relation. In our implementation of Edge^{++}, we use horizontal partition in the **Edge** relation based on the type of the parent element, which eliminates some joins in the definition of the validation constraints. For processing updates, we note that incremental computation techniques [18] can be used to improve the checking of the Datalog constraints; we adapted the simple incremental validation algorithm from [4] in our implementation, thus avoiding the recomputation of the recursive constraints from scratch after every update. Finally, it is worth noting that, in this experiment, all transactions were executed at the user level, thus incurring overheads (e.g., query compilation and optimization) that can be avoided if these methods are implemented inside the database engine.

Updates. The update workload 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 deletion removes one of the "new" items inserted. Table 1 shows that while both mapping approaches scale very well with document size, LILO is up two times faster for insertions and 45% faster for deletions when compare to Edge^{++}; on average, LILO is 83% faster for insertions and 36% faster for deletions.

Queries. While our focus in this paper is on information preservation, we compared Edge^{++} and LILO for query processing as well. We used XMark queries whose focus was on navigating the documents, and here we show the results for two of them: Q3 and Q17; Q3 takes order into account and performs a join, while Q17 is among the simplest queries in the benchmark. For practical reasons, we set a timeout limit of 10 minutes for running each query. Table 1 shows that LILO is vastly superior to Edge^{++} for query processing: Q3 times out on the 256MB Edge^{++} mapping, while it takes less than a second on the LILO mapping for the 2GB document. The improved performance is due mostly to the considerably fewer joins required for querying LILO mappings compared to Edge^{++}.

6 Discussion and Related Work

We proposed a novel and sound framework for generating information-preserving XML-to-relational mapping schemes from an XML schema, which consists of applying equivalence-preserving transformations to schemes that are known to be information-preserving. As discussed, this process results in a mapping scheme whose relational schema is *equivalent* to the original XML schema. Our framework is extensible and can handle arbitrary transformations that can be written in wrec-ILOG$^{\neg}$, which captures all transformations of interest. We also introduced LILO: a mapping scheme design algorithm for $\mathcal{XDS}$ that uses transformations to derive new mapping schemes that preserve the *validity* of XML documents.

Using relational constraints to ensure validity as in LILO has several advantages compared to the alternative of materializing and re-validating the portion of the document that is updated. Notably, our approach leverages the constraint-checking infrastructure already available in the DBMS, and does not require the development and maintenance of a separate validation tool. Moreover, our experiments indicate that even an implementation that incurs all the overheads of user-level transactions leads to good performance.

While checking and incremental checking of constraints in the relational setting is well studied (see, e.g., [18] for a survey), researchers are only beginning to consider updating XML [7, 24, 32, 33], and the problems of validation and incremental validation after updates [4, 21, 28]. In our setting, document validation amounts to checking Datalog constraints, while incremental validation amounts to incremental checking of such constraints.

There are well known notions of *information preservation* in the context of the relational and hierarchical data models [19, 26]; recently, information preservation of XML-to-XML mappings has been studied as well [5]. However, to the best of our knowledge, there has been no work on information preservation in the context of XML-to-relational mapping schemes. In [3], we discussed *losslessness* and *validation*, and in this paper we showed how these notions can be used for establishing the *relative information capacity* of XML and relational schemas. We note that other authors have also identified the need for information preservation in mapping schemes and have informally discussed losslessness [15, 33].

The literature on mapping schemes is already vast, although the focus to date has been mostly on the performance of queries (see e.g., [22] for a survey) and updates [32, 33], rather than on information preservation. Nevertheless, several existing methods (possibly with straightforward extensions) can guarantee losslessness. For instance, numbering schemes that capture both element identity and ordering [33, 34], can be used to fully preserve the structure of the documents [6, 15, 17, 31]. While some mapping schemes are oblivious to document schemas [17], others exploit (and *require*) document schemas [6, 29, 31]. The latter approach has been shown to lead to better query performance, and is more closely related to our framework and LILO; a similar gain in query performance is noticed when comparing LILO to Edge^{++}. Some of the transformations used in LILO can be viewed as extending those in [6, 29, 31] to guarantee information preservation. LegoDB [6] uses a cost-based model for designing mapping schemes whose goal is minimizing the *estimated* cost of executing an input query workload on an input XML document. Such estimates are obtained by using features in modern RDBMSs that allow the query optimizer to use statistical information describing a *hypothetical* database instance. While it would be interesting to extend LILO with a cost-based model, strong empirical evidence suggests that there can be considerable discrepancies between the *estimated* and the *actual* execution costs of queries, even for simple SQL workloads [12]. Thus, other models may have to be considered.

Several techniques have been proposed for translating specific XML Schema constraints into relational ones in mapping schemes; for instance, techniques for translating keys [14], foreign-keys [11], cardinality constraints [6, 23], ID/IDREF attributes [4], and type specialization [3] have been proposed. However, to the best of our knowledge, no work has addressed the problem of mapping the element validity constraint, requiring the content of valid elements to be words in regular languages defined in the document schemas [8], which is achieved by LILO.

Finally, an interesting observation made in [2] is that some of the strategies defined in the literature are orthogonal, and new mapping schemes can be derived by mixing them. The framework we propose is extensible can accommodate *any* transformation that can be described in wrec-ILOG$^\neg$. Thus, while we considered transformations for preserving element validity only, our framework is not tied to specific kinds of constraints: different transformations that preserve additional constraints can be easily incorporated.

Acknowledgments. This work was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada, the IRIS Network of Centers of Excellence, the National Science Foundation, and an IBM Ph.D. Fellowship.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[2] S. Amer-Yahia, F. Du, and J. Freire. A Comprehensive Solution to the XML-to-Relational Mapping Problem. In *WIDM*, 2004.

[3] D. Barbosa, J. Freire, and A. O. Mendelzon. Information Preservation in XML-to-Relational Mappings. In *XSym*, 2004.

[4] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient Incremental Validation of XML Documents. In *ICDE*, 2004.

[5] P. Bohannon, W. Fan, M. Flaster, and P. Narayan. Information Preserving XML Schema Embedding. In *VLDB*, 2005.

[6] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *ICDE*, 2002.

[7] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML View Updates to Relational View Updates: old solutions to a new problem. In *VLDB*, 2004.

[8] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. M. (Editors). *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, third edition, 2004.

[9] A. Brüggemann-Klein and D. Wood. One-Unambiguous Regular Languages. *Information and Computation*, 142, 1998.

[10] 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 *VLDB*, 2000.

[11] Y. Chen, S. Davidson, C. S. Hara, and Y. Zheng. RRXF: Redundancy reducing XML storage in relations. In *VLDB*, 2003.

[12] M. Consens, D. Barbosa, A. Teisanu, and L. Mignet. Goals and Benchmarks for Autonomic Configuration Recommenders. In *SIGMOD*, 2005.

[13] C. J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison Wesley, 4th edition, 1997.

[14] S. Davidson, W. Fan, C. Hara, and J. Qin. Propagating XML Constraints to Relations. In *ICDE*, 2003.

[15] A. Deutsch, M. Fernández, and D. Suciu. Storing Semistructured Data with STORED. In *SIGMOD*, 1999.

[16] M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A Framework for Publishing Relational Data in XML. *TODS*, 27(4), 2002.

[17] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[18] A. Gupta and I. S. Mumick, editors. *Materialized Views - Techniques, Implementations and Applications*. MIT Press, 1998.

[19] R. Hull. Relative Information Capacity of Simple Relational Database Schemata. *SIAM Journal of Computing*, 15(3), 1986.

[20] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *VLDB*, 1990.

[21] B. Kane, H. Su, and E. A. Rundensteiner. Consistently Updating XML Documents Using Incremental Constraint Check Queries. In *CIKM*, 2002.

[22] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL Query Translation Literature: the State of the Art and Open Problems. In *XSym*, 2003.

[23] D. Lee and W. W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *ER*, 2000.

[24] P. Lehti. Design and implementation of a data manipulation processor for an xml query language. Master's thesis, Universität Darmstadt, 2001.

[25] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[26] R. Miller, Y. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *VLDB*, 1993.

[27] M. Nicola and J. John. XML Parsing: a Threat to Database Performance. In *CIKM*, 2003.

[28] Y. Papakonstantinou and V. Vianu. Incremental Validation of XML Documents. In *ICDT*, 2003.

[29] M. Ramanath, J. Freire, J. R. Haritsa, and P. Roy. Searching for Efficient XML-to-Relational Mappings. In *XSym*, 2003.

[30] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, 2002.

[31] 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 *VLDB*, 1999.

[32] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, 2001.

[33] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD*, 2002.

[34] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD*, 2001.