Tractable XML Data Exchange via Relations

Rada Chirkova
Department of Computer Science
North Carolina State University
chirkova@csc.ncsu.edu

Leonid Libkin
School of Informatics
University of Edinburgh
libkin@inf.ed.ac.uk

Juan L. Reutter
School of Informatics
University of Edinburgh
juan.reutter@ed.ad.uk

ABSTRACT

We consider data exchange for XML documents: given source and target schemas, a mapping between them, and a document conforming to the source schema, construct a target document and answer target queries in a way that is consistent with source information. The problem has primarily been studied in the relational context, in which data-exchange systems have also been built.

Since many XML documents are stored in relations, it is natural to consider using a relational system for XML data exchange. However, there is a complexity mismatch between query answering in relational and XML data exchange, which indicates that restrictions have to be imposed on XML schemas and mappings, and on XML shredding schemes, to make the use of relational systems possible.

We isolate a set of five requirements that must be fulfilled in order to have a faithful representation of the XML data-exchange problem by a relational translation. We then demonstrate that these requirements naturally suggest the inlining technique for data-exchange tasks. Our key contribution is to provide shredding algorithms for schemas, documents, mappings and queries, and demonstrate that they enable us to correctly perform XML data-exchange tasks using a relational system.

Categories and Subject Descriptors
H.2.5 [**Heterogeneous Databases**]: Data translation

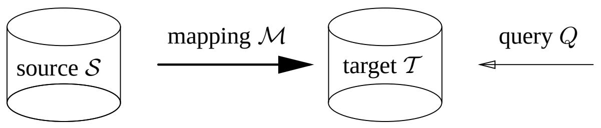
General Terms
Algorithms, Theory

Keywords
Data Exchange, XML, XML Shredding, Inlining

1. Introduction

Data exchange is the problem of finding an instance of a target schema, given an instance of a source schema and a schema mapping, that is, a specification of the relationship between the source and the target. Such a target instance should correctly represent information from the source instance under the constraints imposed by the target schema, and should allow one to evaluate queries on the target instance in a way that is semantically consistent with the source data. The problem has received much attention in the past few years, with several surveys already available [21, 9, 8].

The general setting of data exchange is this:



We have fixed source and target schemas, an instance $\mathcal{S}$ of the source schema, and a mapping $\mathcal{M}$ that specifies the relationship between the source and the target schemas. The goal is to construct an instance $\mathcal{T}$ of the target schema, based on the source and the mapping, and answer queries against the target data in a way consistent with the source data.

The mappings rarely specify the target instance completely, that is, for each source $\mathcal{S}$ and mapping $\mathcal{M}$, there could be multiple target instances $\mathcal{T}_1, \mathcal{T}_2, \ldots$ that satisfy the conditions of the mapping. Such instances are called *solutions*. The notion of query answering has to account for their non-uniqueness. Typically, one tries to compute *certain answers* $\text{CERTAIN}_{\mathcal{M}}(Q, \mathcal{S})$, i.e., answers independent of a particular solution chosen. If Q produces relations, these are usually defined as $\bigcap_i Q(\mathcal{T}_i)$. Certain answers must be produced by evaluating some query – not necessarily Q but perhaps its *rewriting* Q_{rewr} over a particular solution $\mathcal{T}$, so that $Q_{\text{rewr}}(\mathcal{T}) = \text{CERTAIN}_{\mathcal{M}}(Q, \mathcal{S})$.

Thus, the key tasks in data exchange are: (a) choosing a particular solution $\mathcal{T}$ among $\{\mathcal{T}_1, \mathcal{T}_2, \ldots\}$ to materialize, and (b) finding a way of producing query answers over that solution by running a rewritten query Q_{rewr} over it. Usually one builds a so-called *universal* solution [12, 8]; these solutions behave particularly nicely with respect to query answering.

These basics of data exchange are independent of a particular model of data. Most research on data exchange, however, occurred in the relational context [12, 13, 21, 8] or slight extensions [32, 18]; the first paper that attempted to extend relational results to the XML context was [6], and a few followups have since appeared [4, 3]. They all concentrate on the algorithmic aspects of query answering and constructing solutions, with the main goal of isolating tractable cases. The problem these papers do not address is *how XML data exchange can be implemented*?

Previous work on algorithms for XML data exchange has tacitly assumed that one uses a native XML DBMS such as [19]. However, this is not the only (and perhaps not even the most common) route: XML documents are often stored in relational DBMSs. Note that it is natural and in many cases desirable to be able to use the established relational technology to solve the considerably more recent and not as well understood XML data-exchange task. In fact, many ETL products claim that they handle XML data simply by producing relational translations (known as *shredding* [22]). This leads to a two-step approach:

- first shred XML data into relations;
- then apply a relational data-exchange engine (and publish the result back as an XML document).

The approach seems very natural, but the key question is whether it will *work correctly*. That is, are we guaranteed to have the same result as we would have gotten had we implemented a native XML data-exchange system? We answer this question in this paper.

To state more precisely the main question addressed in this paper, assume that we have a translation $\sigma(\cdot)$ that can be applied to (a) XML schemas, (b) XML documents, (c) XML schema mappings, and (d) XML queries. Then the concept of *correctness* of such a translation is shown below:

$$\begin{tikzcd}
\text{XML :} & \text{source } \mathcal{S} \arrow[r, "\text{mapping } \mathcal{M}"] \arrow[d, "\text{shred}"] & \text{target } \mathcal{T} \arrow[r, "\text{query } Q"] \arrow[d, "\text{shred}"] & \text{answer} \arrow[d, "\text{shred}"] \\
\text{Relations :} & \sigma(\mathcal{S}) \arrow[r, "\sigma(\mathcal{M})"] & \sigma(\mathcal{T}) \arrow[r, "\sigma(Q)"] & \text{answer}
\end{tikzcd}$$

That is, suppose we start with an XML document $\mathcal{S}$ and an XML schema mapping $\mathcal{M}$. In a native system, we would materialize some solution $\mathcal{T}$ over which we could answer queries Q.

But now we want a relational system to do the job. So we shred $\mathcal{S}$ into $\sigma(\mathcal{S})$ and then apply to $\sigma(\mathcal{S})$ the translation of the mapping $\sigma(\mathcal{M})$ to get a solution – which itself is a shredding of an XML solution – so that the answer to Q could be reconstructed from the result of the query $\sigma(Q)$ over that relational solution.

The idea seems simple and natural on the surface, but starts looking challenging once we look deeper into it. Before even attempting to show that the relational translation faithfully represents the XML data-exchange problem, we need to address the following.

Complexity mismatch. Without restrictions, there *cannot be a faithful representation* of XML data exchange by a relational system. Indeed, it is well known that positive relational-algebra queries can be efficiently evaluated in relational data exchange [12, 21, 8], but even for simple XML analogs of conjunctive queries finding query answers can be coNP-hard [6]. So any claim that a relational data-exchange system correctly performs XML data exchange for arbitrary documents and queries is bound to be wrong. We thus need to identify the cases that can be handled by a relational system.

Which shredding scheme to use? There are several, that can roughly be divided into two groups: those that do not take the schema information into account (e.g., the edge representation [14], interval codings [33], and other numbering schemes [30]), and those that are based on schemas for XML, such as variants of the inlining technique [28, 22]. Since in data-exchange scenarios we start with two schemas, it seems more appropriate to apply schema-based techniques.

Target constraints. In relational data exchange, constraints in target schemas are required to satisfy certain acyclicity conditions; without them, the chase procedure that constructs a target instance does not terminate [12, 21, 8]. Constraints imposed by general XML schema specifications need not in general be even definable in relational calculus, let alone be acyclic [20]. We thus need to find a shredding technique that enables us to encode targets schemas by means of constraints that guarantee chase termination.

As for the complexity issue, the work on the theory of XML data exchange has identified a class of mappings for which efficient query answering is possible [6, 4, 3]. The schemas (say, DTDs), have rules of the form $db \to book^*$, $book \to author^*\ subject$ (we shall give a formal definition later), and the mappings transform patterns satisfied over the source into patterns satisfied over targets. Observe that these mappings (just as nested-relational tgds [15, 27]) are strictly more expressive than relational tgds; see the Related-Work section for a discussion.

This restriction suggests a relational representation to use. Going with the edge representation [14] is problematic: first, each edge in an XML pattern used in a mapping will result in a join in the relational translation, making it inefficient, and second, enforcing even a simple schema structure under that representation takes us out of the class of target constraints that relational data-exchange systems can handle. Verifiably correct translations based on numerical encodings [30, 33] will necessarily involve numerical and/or ordering constraints in relational translations of mappings, and this is something that relational data exchange cannot handle at the moment [21, 8] (beyond simple ordering constraints [2]).

One translation scheme however that fits in well with restrictions identified in [6, 4, 3] is the *inlining* scheme. It works very well for DTDs of the "right" shape, and its output schemas involve only acyclic constraints, which is perfect for data-exchange scenarios.

Desiderata for the translation We now formulate some basic requirements for the translation σ, in order to be able to achieve our goals described in the diagram above.[1] We need the following:

Requirement 1: translation of schemas A translation $\sigma(D)$ that, when applied to a DTD of a special form, produces a relational schema that only has acyclic constraints, which can be used in a relational data-exchange setting.

Requirement 2: translation of documents A translation $\sigma_D(\cdot)$ for a DTD D that, when applied to document T conforming to D, produces relational database $\sigma_D(T)$ of schema $\sigma(D)$.

Requirement 3: translation of queries For a DTD D, a translation $\sigma_D(Q)$ of (analogs of) conjunctive queries so that $\sigma_D(Q)\big(\sigma_D(T)\big) = Q(T)$ (that is, the result of $Q(T)$ can be computed by relational translations).

Requirement 4: translation of mappings For a mapping $\mathcal{M}$ between a source DTD D_s and a target DTD D_t, its translation $\sigma(\mathcal{M})$ is a mapping between $\sigma(D_s)$ and $\sigma(D_t)$ that preserves universal solutions. That is:

(a) Each σ_{D_t}-translation of a universal solution for T under $\mathcal{M}$ is a universal solution for $\sigma_{D_s}(T)$ under $\sigma(\mathcal{M})$; and

(b) Each universal solution for $\sigma_{D_s}(T)$ under $\sigma(\mathcal{M})$ contains[2] a σ_{D_t}-translation of a universal solution of T under $\mathcal{M}$.

Requirement 5: query answering For (analogs of) conjunctive queries over trees, computing the answer to Q under $\mathcal{M}$ over a source tree T is the same as computing a $\sigma(\mathcal{M})$-solution of $\sigma(T)$, followed by evaluation of $\sigma(Q)$ over that solution, as is normally done in a relational data-exchange system.

Satisfaction of these five requirements would guarantee that we have a *correct* relational translation of an XML data-exchange problem, which would guarantee correct evaluation of queries. The

[1]In the next sections we formalize each desideratum.

[2]We cannot require the equivalence, as relational solutions are open to adding new tuples and thus cannot always be translations of trees; we shall discuss this later.

relational approach to XML data exchange, which we propose in this paper, satisfies all the five requirements.

For the choice of the query language, one has to be careful since the definition of certain answers depends on the output of the queries. We consider two classes of conjunctive queries over trees. The first is tree patterns that output tuples of attribute values. These are the queries most commonly considered in XML data exchange [6, 4, 3] because for them we can define certain answers as the usual intersection $\text{CERTAIN}_{\mathcal{M}}(Q, \mathcal{S}) = \bigcap_i Q(\mathcal{T}_i)$. The second is a simple XML-to-XML query language from whose queries output *trees*. It is essentially the positive fragment of FLWR expressions of XQuery [31]. For outputs which are XML trees, the intersection operator is no longer meaningful for defining certain answers. Instead, we use recent results of [11] that show how to define and compute certain answers for XML-to-XML queries.

Contributions We provide a relational approach to solve two of the most important problems of XML data-exchange settings: materializing solutions and answering queries. Our specific contributions are as follows. First, we introduce an architecture for XML data exchange using relational vehicles, with a focus on correct evaluation of (analogs of) conjunctive queries on XML data. Second, we identify a class of XML schema mappings and a shredding mechanism that allows us to overcome the complexity mismatch. Third, we provide algorithms for relational translation of schemas, XML documents, schema mappings, and queries in our proposed architecture. Finally, we prove the correctness of the translations: namely, we show that they satisfy the above five requirements, and thus enable us to use relational data exchange systems for XML data-exchange tasks. Since the computational complexity of our proposed algorithms is quite low, and their correctness has been established, we believe this paper makes a case for using relational technology for provably correct XML data exchange.

Related work In recent years, significant effort has been devoted to developing high-performance XML database systems, and to building tools for data exchange. One major direction of the XML effort is the "relational approach", which uses relational DBMSs to store and query XML data. Documents could be translated into relational tuples using either a "DTD-aware" translation [29, 28] or a "schemaless" translation. The latter translations include the edge [14] and the node [33] representations of the data. Indexes could be prebuilt on the data to improve performance in relational query processing, see, e.g., [30, 33]. Constraints arising in the translation are sometimes dealt with explicitly [7, 23]. See [17] for a survey of the relational approach to answering XML queries.

The work on data exchange concentrated primarily on relations, see [8, 21] for surveys and [26, 27] for system descriptions. Mappings for the XML data exchange problem were studied in [6, 4]; it was noticed there that the complexity of many tasks in XML data exchange is higher than for their relational analogs, which suggests that restrictions must be imposed for a relational implementation. The problem of exchanging XML data was also studied in [15, 27], which give translations of documents and DTDs into nested-relational schemas, and then show how to perform XML data exchange under this translation. Most RDBMSs, however, do not provide support for nested relational schemas, and, thus, specific machinery has to be developed in order to implement this translation under a strictly relational setting. Moreover, XML mappings considered in this paper are strictly more expressive than nested-relational mappings, and every nested-relational data exchange setting can be efficiently transformed into an equivalent XML data exchange setting. Thus, the results of this paper may aid towards the development of a relational implementation for both XML and nested-relational data exchange.

Outline Key definitions are given in Section 2. Section 3 provides translations of schemas and documents and shows that they fulfill Requirements 1 and 2. Section 4 states the main concepts of relational and XML data exchange. Section 5 provides translations of mappings and queries, and shows that Requirements 3, 4, and 5 are fulfilled. Section 6 studies queries that output XML trees.

2. Preliminaries

Relational schemas and constraints. A *relational schema*, or just *schema*, is a finite set $\mathbf{R} = \{R_1, \ldots, R_k\}$ of relation symbols, possibly with a set of integrity constraints (*dependencies*). Constraints used most often in data exchange are equality- and tuple-generating dependencies [12, 21, 8], but for our purposes it will suffice to consider only *keys* and *foreign keys*. If R is a relation over attributes U, and X is a set of attributes, then X is a key of R if no two tuples of R coincide on X-attributes (that is, for all tuples $t_1, t_2 \in R$ with $t_1 \neq t_2$ we have $\pi_X(t_1) \neq \pi_X(t_2)$). If R_1 and R_2 are relations over sets of attributes U_1 and U_2, respectively, then an inclusion constraint $R_1[X] \subseteq R_2[Y]$, where $X \subseteq U_1$ and $Y \subseteq U_2$ are of the same cardinality, holds when $\pi_X(R_1) \subseteq \pi_Y(R_2)$. We further say that a foreign key on the attributes of $R_1[X] \subseteq_{FK} R_2[Y]$ holds if the inclusion constraint $R_1[X] \subseteq R_2[Y]$ holds, and Y is a key of R_2.

With each set of keys and foreign keys, we associate a graph in which we put an edge between attributes A and B if there is a constraint $R_1[X] \subseteq_{FK} R_2[Y]$ with $A \in X$ and $B \in Y$. If this graph is acyclic, we say that the set of constraints is *acyclic*. A schema is acyclic if its constraints are acyclic. In data exchange, one often uses a more technical notion of weak acyclicity: it includes some cyclic schemas for which the chase procedure still terminates. For us, however, the simple concept of acyclicity will suffice, as our translations of schemas only produce acyclic constraints.

XML documents and DTDs Assume that we have the following disjoint countably infinite sets: El of element names, Att of attribute names, and Str of possible values of string-valued attributes. All attribute names start with the symbol @.

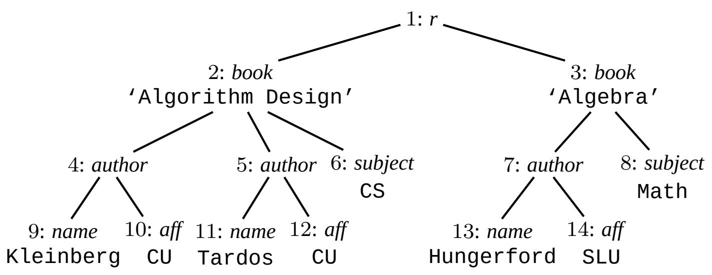
An *XML tree* is a finite rooted directed tree $T = (N, G)$, where N is the set of nodes and G is the set of edges, together with

1. a labeling function $\lambda : N \to El$;
2. attribute-value assignments, which are partial functions $\rho_{@a} : N \to Str$ for each $@a \in Att$; and
3. an ordering on the children of every node.

A DTD D over El with a distinguished symbol r (for the root) and a set of attributes Att consists of a mapping P_D from El to regular expressions over $El - \{r\}$, usually written as productions $\ell \to e$ if $P_D(\ell) = e$, and a mapping A_D from El to 2^{Att} that assigns a (possibly empty) set of attributes to each element type. For notational convenience, we always assume that attributes come in some order, just like in the relational case: attributes in tuples come in some order so we can write $R(a_1, \ldots, a_n)$. Likewise, we shall describe an ℓ labeled tree node with n attributes as $\ell(a_1, \ldots, a_n)$.

A tree T conforms to a DTD D (written as $T \models D$) if its root is labeled r, the set of attributes for a node labeled ℓ is $A_D(\ell)$, and the labels of the children of such a node, read from left to right, form a string in the language of $P_D(\ell)$.

A class of DTDs In this paper we consider a restriction on DTDs called *nested-relational DTDs* [1, 6], a class of DTDs that naturally represent nested relational schemas such as the ones used by the Clio data-exchange system [26]. The reason for using them is that outside of this class, it is very easy to construct instances of



(a) Tree T

r	$\rightarrow$	*book**
book	$\rightarrow$	$author^*\ subject$
$author$	$\rightarrow$	$name\ aff$
$A_D(book)$	$=$	$@title$
$A_D(subject)$	$=$	$@sub$
$A_D(name)$	$=$	$@nam$
$A_D(aff)$	$=$	$@aff$

(b) DTD D

Figure 1: The XML tree T conforms to D

XML data-exchange problems that will exhibit coNP-hardness of answering conjunctive queries (which are known to be tractable in practically all instances of relational data exchange), see [6].

A DTD D is *non-recursive* if the graph $G(D)$ defined as $\{(\ell, \ell') \mid \ell'$ is mentioned in $P(\ell)\}$ is acyclic. A non-recursive DTD D is *nested-relational* if all rules of D are of the form $l \rightarrow \tilde{l}_0 \ldots \tilde{l}_m$ where all the l_i's are distinct, and each $\tilde{l}_i$ is one of l_i and l_i^*. From now on, unless otherwise noted, all DTDs are assumed to be nested-relational. We also assume, without loss of generality, that the graph $G(D)$ is not a directed acyclic graph (DAG) but a tree. (One can always unfold a DAG into a tree by tagging occurrences of element types with the types of their predecessors.)

EXAMPLE 2.1. Figure 1(a) shows an example of an XML tree. In the Figure, the node identifiers precede the corresponding labels of each node in T; we omit the attribute names and only show the attribute values of each node. In addition, Figure 1(b) shows an example of a nested relational DTD. Moreover, it is easy to see that the tree T of Figure 1(a) conforms to D. $\square$

3. Translations of schemas and documents

We now review the *inlining* technique [28], provide a precise definition of the translation, and show that it satisfies our **Requirements 1** and **2**. The main idea of inlining is that separate relations are created for the root and each element type that appears under a star, and other element types are inlined in the relations corresponding to their "nearest appropriate ancestor". Each relation for an element type has an ID attribute that is a key, as well as (for non-root) a "parent-ID" attribute that is a foreign key pointing to the "nearest appropriate ancestor" of that element in the document. All the attributes of a given element type in the DTD become attributes in the relation corresponding to that element type when such a relation exists, or otherwise become attributes in the relation for the "nearest appropriate ancestor" of the given element type.

We begin with a formal definition of the *nearest appropriate ancestor* for the element types used in D. Given a nested-relational DTD $D = (P_D, A_D, r)$, we "mark" in $G(D)$ each element type that occurs under a star in P_D. In addition, we mark the root element type in $G(D)$. Then, for a given element type ℓ, we define the *nearest appropriate ancestor* of ℓ, denoted by $\mu(\ell)$, as the closest marked element type ℓ' in the path from the root element to ℓ in the graph $G(D)$. The inlining schema generation is formally captured by means of the procedure INLSCHEMA below.

EXAMPLE 3.1. Consider again DTD D in Figure 1(b). The relational schema INLSCHEMA(D) is as follows:

$R_r(\underline{\texttt{rID}})$
$R_{book}(\underline{\texttt{bookID}},\texttt{@title},\texttt{rID},\texttt{subID},\texttt{@sub})$
$R_{author}(\underline{\texttt{authID}},\texttt{bookID},\texttt{nameID},\texttt{afID},\texttt{@nam},\texttt{@aff})$

Keys are underlined; we also have the following foreign keys: $R_{book}(\texttt{rID}) \subseteq_{FK} R_r(\texttt{rID})$ and $R_{author}(\texttt{bookID}) \subseteq_{FK} R_{book}(\texttt{bookID})$. $\square$

```
Procedure INLSCHEMA($D$)
  Input : A nested relational DTD $D$.
  Output: A relational schema $\mathbf{S}_D$ and a set of integrity
          constraints $\Delta_D$
  Set $\mathbf{S}_D = \emptyset$ and $\Delta_D = \emptyset$
  for each marked element type $\ell$ of $D$:
     add to $\mathbf{S}_D$ a relation $R_\ell$, with attributes:
        $attr(R_\ell) = \begin{cases} id_\ell \\ A_D(\ell) \\ id_{\mu(\ell)} & \mid \text{ if } \ell \neq r. \\ id_{\ell'} & \mid \mu(\ell') = \ell,\ \ell' \text{ is not marked,} \\ A_D(\ell') & \mid \mu(\ell') = \ell,\ \ell' \text{ is not marked.} \end{cases}$
  endfor
  for each relation $R_\ell$ in $\mathbf{S}_D$:
     add to $\Delta_D$ the constraint stating that $id_\ell$ is key of $R_\ell$ and,
     if $\ell \neq r$, the foreign key
        $R_\ell[id_{\mu(\ell)}] \subseteq_{FK} R_{\mu(\ell)}[id_{\mu(\ell)}]$.
  endfor
  add to $\Delta_D$ the dependency (stating the uniqueness of the root)
        $\forall \bar{y} \forall \bar{z} R_r(x, \bar{y}) \wedge R_r(x', \bar{z}) \rightarrow x = x'$.
  return $(\mathbf{S}_D, \Delta_D)$
```

The following shows that our **Requirement 1** is satisfied.

PROPOSITION 3.2. *For every nested relational DTD D, the output of* INLSCHEMA*(D) is an acyclic relational schema.*

Shredding of XML documents. We now move to the shredding procedure. Given the inlining INLSCHEMA(D) $= (\mathbf{S}_D, \Delta_D)$ of a DTD D, and an XML tree T conforming to D, we use the algorithm INLDOC to *shred* T into an instance of the relational schema $\mathbf{S}_D$ that satisfies the constraints in Δ_D. Let us first explain this translation by means of an example.

EXAMPLE 3.3. Recall tree T from Figure 1(a) and DTD D from Figure 1(b). Figure 2 shows relations R_{book} and R_{author} in the shredding of T. $\square$

To present the algorithm, we define the *nearest appropriate ancestor* $\mu(n)$ of a node n of an XML document T that conforms to a DTD D, as follows. Mark each node n of T such that $\lambda(n)$ is starred in D, as well as the root of T. Then $\mu(n)$ is the closest marked node n' that belongs to the path from the root to n. In the following algorithm, and for the remainder of the paper, we denote by id_n the relational element representing the node n of a tree T.

bookID	@title	rID	subID	@sub
id_2	'Algorithm Design'	id_1	id_6	CS
id_3	'Algebra'	id_1	id_8	Math

(a) Relation R_{book} in $\textsc{InlDoc}(T, D)$

authID	bookID	nameID	afID	@nam	@af
id_4	id_2	id_9	id_{10}	'Kleinberg'	CU
id_5	id_2	id_{11}	id_{12}	'Tardos'	CU
id_7	id_3	id_{13}	id_{14}	'Hungerford'	SLU

(b) Relation R_{author} in $\textsc{InlDoc}(T, D)$

Figure 2: Shredding of T into $\textsc{InlSchema}(D)$

```
Procedure INLDOC($T, D$)
  Input  : A nested relational DTD $D$ and an XML tree $T$ that
           conforms to $D$.
  Output: A relational instance of the schema INLSCHEMA($D$).
  for each marked node $n$ of $T$:
      Let $\ell$ be the label of $n$; Add to the relation $R_\ell$ of $I$ a tuple
      that contains elements
      $\begin{cases} id_n & \\ \rho_{@a}(n) & \mid \quad @a \in A_D(\ell) \\ id_{\mu(n)} & \mid \quad \text{if } \ell \neq r \\ id_{n'} & \mid \quad \mu(n') = n, n' \text{ is not marked.} \\ \rho_{@a}(n') & \mid \quad \mu(n') = n \text{ , } @a \in A_D(\lambda(n')) \text{ and} \\ & \quad\;\; n' \text{ is not marked} \end{cases}$
      where the identifiers and attributes values for each of the
      elements $id_{n'}$, $id_{\mu(n)}$ and $\rho_{@a}(n')$ coincide with the
      position of the attributes for $id_{\lambda(n')}$, $id_{\mu(\ell)}$ and
      $A_D(\lambda(n'))$ of $R_\ell$.
  endfor
  return $I$
```

The following proposition shows our **Requirement 2** is satisfied.

PROPOSITION 3.4. *Let D be a DTD, and T an XML tree such that $T \models D$. Then $\textsc{InlDoc}(T, D)$ is an instance of the schema computed by $\textsc{InlSchema}(D)$.*

4. Relational and XML Data Exchange

We now quickly review the basics of relational data exchange and introduce XML schema mappings that guarantee tractable query answering.

Relational Data Exchange A schema mapping $\mathcal{M}$ is a triple $(\mathbb{S}, \mathbb{T}, \Sigma)$, where $\mathbb{S}$ is a source schema, $\mathbb{T} = (\mathbf{T}, \Delta_{\mathbf{T}})$ is a target schema with a set of constraints Δ_T, and Σ is a set of *source-to-target dependencies* that specify how the source and the target are related. Most commonly these are given as source-to-target tuple generating dependencies (st-tgds):

$$\varphi(\bar{x}) \to \exists \bar{z}\, \psi(\bar{x}, \bar{z}), \tag{1}$$

where φ and ψ are conjunctions of relational atoms over $\mathbb{S}$ and $\mathbb{T}$, respectively.

In data-exchange literature, one normally considers instances with two types of values: constants and nulls. Instances $\mathcal{S}$ of the source schema $\mathbb{S}$ consist only of constant values, and nulls are used to populate target instances $\mathcal{T}$ when some values are unknown.

An instance $\mathcal{T}$ of $\mathbb{T}$ (which may contain both constants and nulls) is called a *solution* for an instance $\mathcal{S}$ of $\mathbb{S}$ under $\mathcal{M}$, or an $\mathcal{M}$-*solution*, if every st-tgd (1) from Σ is satisfied by $(\mathcal{S}, \mathcal{T})$ (that is, for each tuple $\bar{a}$ such that $\varphi(\bar{a})$ is true in $\mathcal{S}$, there is a tuple $\bar{b}$ such that $\psi(\bar{a}, \bar{b})$ is true in $\mathcal{T}$).The set of all $\mathcal{M}$-solutions for $\mathcal{S}$ is denoted by $\textsc{Sol}_{\mathcal{M}}(\mathcal{S})$ (or $\textsc{Sol}(\mathcal{S})$ if $\mathcal{M}$ is understood).

Certain answers and canonical universal solution The main difficulty in answering a query Q against the target schema is that there could be many possible solutions for a given source. Thus, for query answering in data exchange one normally uses the notion of certain answers, that is, answers that do not depend on a particular solution. Formally, for a source $\mathcal{S}$ and a mapping $\mathcal{M}$, we define $\textsc{certain}_{\mathcal{M}}(Q, \mathcal{S})$ as $\bigcap\{Q(\mathcal{T}) \mid \mathcal{T} \in \textsc{Sol}_{\mathcal{M}}(\mathcal{S})\}$.

Building all solutions is impractical (or even impossible), so it is important to find a particular solution $\mathcal{T}_0 \in \textsc{Sol}_{\mathcal{M}}(\mathcal{S})$, and a rewriting Q_{rewr} of Q, so that $\textsc{certain}_{\mathcal{M}}(Q, \mathcal{S}) = Q_{\text{rewr}}(\mathcal{T}_0)$.

Universal solutions were identified in [12] as the preferred solutions in data exchange. Over them, every positive query can be answered, with a particularly simple rewriting: after Q is evaluated on a universal solution $\mathcal{T}_0$, tuples containing null values are discarded. Even among universal solutions there are ones that are most commonly materialized in data-exchange systems, such as the *canonical solution* $\textsc{CanSol}_{\mathcal{M}}(\mathcal{S})$, computed by applying the chase procedure with constraints Σ and $\Delta_{\mathbf{T}}$ to the source instance $\mathcal{S}$. If all the constraints in $\Delta_{\mathbf{T}}$ are acyclic (in fact, even a weaker notion suffices), such a chase terminates and computes $\textsc{CanSol}_{\mathcal{M}}(\mathcal{S})$ in polynomial time [12].

Note that our **Requirement 4** relates universal solutions in relational and XML data exchange; in particular, we do not insist on working with the canonical solutions, and others, such as the core [13] or the algorithmic constructions of [25] can be used as well.

Towards XML schema mappings: patterns To define XML schema mappings, we need the notions of schemas and source-to-target dependencies. The notion of schema is well understood in the XML context. Our dependencies, as in [6, 4, 3] will be based on *tree patterns*. Patterns are defined inductively as follows:

- $\ell(\bar{x})$ is a pattern, where ℓ is a label, and $\bar{x}$ is a (possibly empty) tuple of variables (listing attributes of a node);
- $\ell(\bar{x})[\pi_1, \ldots, \pi_k]$ is a pattern, where $\pi_1, \ldots, \pi_k$ are patterns, and ℓ and $\bar{x}$ are as above.

We write $\pi(\bar{x})$ to indicate that $\bar{x}$ is the tuple of all the variables used in a pattern. The semantics is defined with respect to a node of a tree and to a valuation of all the variables of a pattern as attribute values. Formally, $(T, v) \models \pi(\bar{a})$ means that π is satisfied in node v when $\bar{x}$ is interpreted as $\bar{a}$. It is defined as follows:

- $(T, v) \models \ell(\bar{a})$ if v is labeled ℓ and its tuple of attributes is $\bar{a}$;
- $(T, v) \models \ell(\bar{a})[\pi_1(\bar{a}_1), \ldots, \pi_k(\bar{a}_k)]$ if
  1. $(T, v) \models \ell(\bar{a})$ and
  2. there exist children $v_1, \ldots, v_k$ of v (not necessarily distinct) so that $(T, v_i) \models \pi_i(\bar{a}_i)$ for every $i \leq k$.

We write $T \models \pi(\bar{a})$ if $(T, r) \models \pi(\bar{a})$, that is, the pattern is witnessed at the root.

EXAMPLE 4.1. Consider tree T from Figure 1(a), and the tree pattern $\pi(x, y) = r[book(x)[author[name(y)]]]$, which finds books together with the names of their authors. Then it is easy to see that $T \models \pi($`'Algorithm Design', Tardos`$)$. In fact, evaluation of $\pi(x, y)$ over T returns the tuples (`'Algorithm Design', Tardos`), (`'Algorithm Design', Kleinberg`), and (`'Algebra',Hungerford`). □

Given a DTD D and a tree pattern π, we say that π is *compatible* with D if there exists a tree T that conforms to D and a tuple of attribute values $\bar{a}$ such that $T \models \pi(\bar{a})$. In general, checking compatibility of patterns with DTDs is NP-complete [10], but for the DTDs we consider here it can be easily done in polynomial time.

EXAMPLE 4.2.[Example 4.1 continued] The pattern $\pi(x, y)$ is compatible with the DTD D of Figure 1(b). On the other hand, the pattern $\pi'(x) = r[author(x)]$ is not, because no tree consistent with D can have a child of r labeled as *author*, or an *author*-labeled node with an attribute. $\square$

Remark More general patterns have been considered in the literature [5, 24, 10, 4, 3]; in particular, they may involve descendant navigation, wild cards for labels, and sibling order. However, [6, 4, 3] showed that with these features added, query answering in data exchange becomes intractable even for very simple queries. In fact, the restrictions we use in our definition were identified in [6] as essential for tractability of query answering. Note that the same restriction was imposed to queries when transforming XML data into nested-relational schemas [15, 27].

XML schema mappings As our descriptions of XML schemas we shall use DTDs (since for complex schemas, query answering in data exchange is known to be intractable [6], and DTDs will suffice to capture all the known tractable cases). Source-to-target constraints will be given via patterns.

Formally, an *XML schema mapping* is a triple $\mathcal{M} = (D_S, D_T, \Sigma)$, where D_S is the source (nested relational) DTD, D_T is the target (nested relational) DTD, and Σ is a set of *XML source-to-target dependencies* [6], or XML stds, of form

$$\pi(\bar{x}) \rightarrow \pi'(\bar{x}, \bar{z}), \qquad (2)$$

where π and π' are tree patterns compatible with D_S and D_T, respectively.

As in the relational case, target trees may contain nulls to account for values not specified by mappings. Given a tree T that conforms to D_S, a tree T' (over constants and nulls) is an $\mathcal{M}$-solution for T if T' conforms to D_T, and the pair (T, T') satisfies all the dependencies (2) from Σ. The latter means that for every tuple $\bar{a}$ of attribute values from T, if T satisfies $\pi(\bar{a})$, then there exists a tuple $\bar{b}$ of attribute values from T' such that T' satisfies $\pi'(\bar{a}, \bar{b})$. The set of all $\mathcal{M}$-solutions for T is denoted by $\textsc{Sol}_{\mathcal{M}}(T)$.

EXAMPLE 4.3. Consider the data-exchange scenario $(D, D_T, \mathcal{M})$ given by the DTDs D and D_T of Figures 1(b) and 3(b), respectively, and where $\mathcal{M}$ is specified by the dependency

$$r[book(x)[author[name(y)]]] \rightarrow r[writer[name(y), work(x)]],$$

that restructures book-author pairs as writer-work. It can be shown that the XML tree T' in Figure 3(a) is an $\mathcal{M}$-solution for T. $\square$

5. XML data exchange using relations

We now provide algorithms for implementing XML data exchange via relational translations. Since we have already shown how to translate DTDs and documents, we need to present translations of stds of mappings and queries. Both of them are based on translating patterns into relational conjunctive queries. We first concentrate on that translation. Then we show how to extend it easily to mappings and queries, and prove the correctness of the translations. This will complete our program of using a relational system for XML data exchange in a semantically correct way.

Inlining tree patterns. The key ingredient in our algorithms is a translation of patterns π compatible with a DTD D into a *conjunctive query* $\textsc{InlPattern}(\pi, D)$ over the relational schema $\textsc{InlSchema}(D)$. Very roughly, it can be viewed as this:

1. View a pattern $\pi(\bar{x})$ as a tree T_π in which some attribute values could be variables;
2. Compute the relational database $\textsc{InlDoc}(T_\pi, D)$ (which may have variables as attribute values);
3. View $\textsc{InlDoc}(T_\pi, D)$ as a tableau of a conjunctive query; the resulting query is $\textsc{InlPattern}(\pi, D)$.

The algorithm is actually more complicated because $\textsc{InlDoc}$ cannot be used in Step 2; we shall explain shortly why.

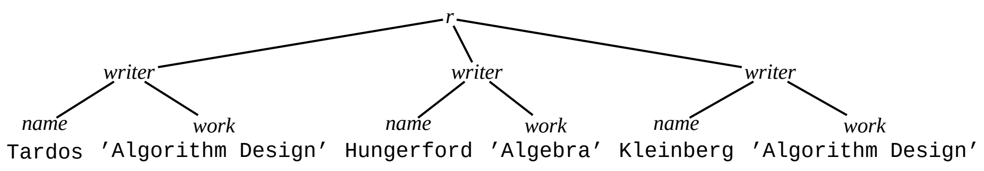
Towards defining $\textsc{InlPattern}$, observe that each tree pattern $\pi(\bar{x})$ can be viewed as an XML document $T_{\pi(\bar{x})}$, in which both values and variables can be used as attribute values. It is defined inductively as follows: $T_{\ell(\bar{x})}$ is a single-node tree labeled ℓ, with $\bar{x}$ as attribute values, and if π is $\ell(\bar{x})[\pi_1(\bar{x}_1), \ldots, \pi_k(\bar{x}_k)]$, then the root of T_π is labeled ℓ and has $\bar{x}$ as attribute values. It also has k children, with the subtrees rooted at them being $T_{\pi_1(\bar{x}_1)}, \ldots, T_{\pi_k(\bar{x}_k)}$.

However, even for a pattern $\pi(\bar{x})$ compatible with a DTD D, we may not be able to define its inlining as the inlining of $T_{\pi(\bar{x})}$, because $T_{\pi(\bar{x})}$ need not conform to D. For example, if a DTD has a rule $r \rightarrow ab$ and we have a pattern $r[a]$, it is compatible with D, but $T_{r[a]}$ does not conform to D, as it is missing a b-node. Hence, the procedure $\textsc{InlDoc}$ cannot be used 'as-is' in our algorithm.

Nevertheless, we can still mark the nodes of $T_{\pi(\bar{x})}$ with respect to D and define the nearest appropriate ancestor exactly as it has been done previously. Intuitively, the procedure $\textsc{InlPattern}$ shreds each node of $T_{\pi(\bar{x})}$ into a different predicate, and then joins these predicates using the nearest appropriate ancestor.

```
Procedure InlPattern($\pi$, $D$)
  Input  : A DTD $D$, a tree pattern $\pi(\bar{x})$ compatible with $D$.
  Output: Conjunctive query over InlSchema($D$).
  for each node $v$ of $T_{\pi(\bar{x})}$ of form $\ell(\bar{x}_v)$:
     Construct a query $Q_v(\bar{x}_v)$ as follows:
     if $v$ is marked then
          $Q_v(\bar{x}_v) := \exists id_v \exists id_{\mu(v)} \exists \bar{z} R_\ell(id_v, \bar{x}_v, id_{\mu(v)}, \bar{z}),$
        where $\bar{z}$ is a tuple of fresh variables, and the positions
        of variables $id_v$, $\bar{x}_v$ and $id_{\mu(v)}$ are consistent with the
        attributes $id_\ell$, $A_D(\ell)$ and $id_{\mu(\ell)}$ respectively in
        $attr(R_\ell)$.
        If $\ell = r$, then $Q_v$ does not use $id_{\mu(v)}$.
     else ($v$ is not marked):
        set $v' := \mu(v)$,  $\ell' := \lambda(v')$, and let $Q_v(\bar{x}_v)$ be
          $\exists id_{v'} \exists id_{\mu(v')} \exists id_v \exists \bar{z} R_{\ell'}(id_{v'}, id_{\mu(v')}, id_v, \bar{x}_v, \bar{z}),$
        where $\bar{z}$ is a tuple of fresh variables, and the positions
        of the variables $id_{v'}$, $id_{\mu(v')}$, $id_v$ and $\bar{x}_v$ are consistent
        with the attributes $id_{\ell'}$, $id_{\mu(\ell')}$, $id_\ell$ and $A_D(\ell)$
        respectively in $attr(R_{\ell'})$. If $\ell' = r$, then $Q_v$ does not
        use $id_{\mu(v')}$.
  endfor
  return $\bigwedge_{v \in T_{\pi(\bar{x})}} Q_v(\bar{x}_v)$.
```

Note that the compatibility of π with D ensures that $\textsc{InlPattern}$ is well defined. That is, (1) every attribute formula of the form $\ell(\bar{x})$ only mentions attributes in $A_D(\ell)$, and (2) for all nodes $v, v' \in T_{\pi(\bar{x})}$, if v' is a child of v, then $\lambda(v') \in P_D(\lambda(v))$.



(a) Target Tree T'

r	$\rightarrow$	$writer^*$
$writer$	$\rightarrow$	$name\ work^*$
$A_D(name)$	$=$	$@nam$
$A_D(work)$	$=$	$@title$

(b) Target DTD D_T

Figure 3: Tree T' is an $\mathcal{M}$-solution for T

Correctness. Given a pattern $\pi(\bar{x})$, the evaluation of π on a tree T is $\pi(T) = \{\bar{a} \mid T \models \pi(\bar{a})\}$. The following proposition shows the correctness of InlPattern.

Proposition 5.1. *Given a nested relational DTD D, a pattern π compatible with D, and a tree T that conforms to D, we have $\pi(T) = \text{InlPattern}(\pi, D)\big(\text{InlDoc}(T, D)\big)$.*

That is, the inlining of π, applied to the inlining of T, returns $\pi(T)$.

Conjunctive queries over trees. We use the language that is essentially conjunctive queries over trees [6, 16, 10] with navigation along the child axis.

The language $\mathcal{CTQ}$ is obtained by closing patterns under conjunction and existential quantification:

$$Q \ := \ \pi \mid Q \wedge Q \mid \exists x\, Q,$$

where π is a fully specified tree-pattern formula. The semantics is straightforward, given the semantics of patterns defined above: $Q(\bar{a}) \wedge Q'(\bar{b})$ is true iff both $Q(\bar{a})$ and $Q'(\bar{b})$ are true, and $\exists x\, Q(\bar{a}, x)$ is true iff $Q(\bar{a}, c)$ is true for some value c. The output of Q on a tree T is denoted by $Q(T)$.

We say that a query Q is compatible with the DTD D if every pattern used in it is compatible with D.

The inlining of queries Q compatible with D is given by the recursive algorithm InlQuery below.

```
Procedure InlQuery($Q$, $D$)
  Input  : A DTD $D$, a query $Q$ compatible with $D$.
  Output: A conjunctive query over InlSchema($D$).
  if $Q = \pi$ then
      return InlPattern($\pi, D$)
  else if $Q = Q_1 \wedge Q_2$ then
      return InlQuery($Q_1, D$) $\wedge$ InlQuery($Q_2, D$)
  else if $Q = \exists x Q_1$ then
      return $\exists x$ InlQuery($Q_1, D$)
```

Now we show that every query Q in $\mathcal{CTQ}$ can be computed by its inlining on the inlining of its input (assuming, of course, compatibility with a DTD). In other words, **Requirement 3** is satisfied.

Theorem 5.2. *Given a DTD D, a tree T that conforms to it, and a compatible query Q, we have*

$$Q(T) \ = \ \text{InlQuery}(Q, D)\big(\text{InlDoc}(T, D)\big).$$

Inlining XML schema mappings We use our transformation of tree patterns to define the procedure InlMap, that, given source and target DTDs D_S and D_T, transforms an XML mapping $\mathcal{M}$ into a relational mapping InlMap($\mathcal{M}$,D_S,D_T) specified with a set of source-to-target tuple generating dependencies.

Correctness While one could be tempted to ask for a translation that preserves all solutions, such a result need not hold. The relational mapping InlMap uses null values to represent the shredded

```
Procedure InlMap($\mathcal{M}$, $D_S$, $D_T$)
  Input  : An XML mapping $\mathcal{M}$ from a source DTD $D_S$ to a
           target DTD $D_T$.
  Output: A relational mapping from InlSchema($D_S$) to
          InlSchema($D_T$).
  Set InlMap($\mathcal{M}, D_S, D_T$) := $\emptyset$
  for dependency $\pi(\bar{x}) \rightarrow \exists\bar{z}\pi'(\bar{x}, \bar{z})$ in $\mathcal{M}$ do
      InlMap($\mathcal{M}, D_S, D_T$) := InlMap($\mathcal{M}, D_S, D_T$) $\bigcup$
      $\{$InlQuery($\pi, D_S$)($\bar{x}$) $\rightarrow \exists\bar{z}$ InlQuery($\pi', D_T$)($\bar{x}, \bar{z}$)$\}$
  end
  return InlMap($\mathcal{M}, D_S, D_T$)
```

nodes of XML trees, and thus we should only consider solutions whose null values have not been renamed. However, relational solutions are open to renaming of nulls. This intuition can be formalized by means of the universal solutions, which are the most general among all solutions, and thus do not permit null renaming. Furthermore, one typically materializes a universal solution, as these solutions contain all the information needed to compute certain answers of conjunctive queries. This motivates the restriction of our **Requirement 4** to universal solutions.

The theorem below shows that parts (a) and (b) of **Requirement 4** hold. Note that in part (b), relational universal solutions are only required to contain a shredding of an XML universal solution. This is because relational solutions are also open to adding arbitrary tuples, which need not reflect a tree structure of an XML document.

Theorem 5.3. *a) Let $\mathcal{M} = (D_S, D_T, \Sigma)$ be an XML schema mapping and T an XML document that conforms to D_S. If T' is an $\mathcal{M}$-universal solution for T, then its inlining* InlDoc(T', D_T) *is an* InlMap($\mathcal{M}, D_S, D_T$)*-universal solution for* InlDoc(T, D_S).
b) Let $\mathcal{M} = (D_S, D_T, \Sigma)$ be an XML schema mapping, and T an XML document that conforms to D_S. Then for every InlMap($\mathcal{M}, D_S, D_T$)*-universal solution R for* InlDoc(T, D_S) *there exists an $\mathcal{M}$-universal solution T' such that* InlDoc(T', D_T) *is contained in R.*

Answering XML queries using relational data exchange. The semantics of query answering in data exchange, both relational and XML [12, 21, 8, 6, 4], is defined by means of certain answers. That is, given a schema mapping $\mathcal{M} = (D_S, D_T, \Sigma)$, a tree T that conforms to D_S, and a conjunctive tree query Q that is compatible with D_T, the *certain answers of Q for T under $\mathcal{M}$,* denoted by certain$_{\mathcal{M}}(Q, T)$, is the set of tuples that belong to the evaluation of Q over every possible $\mathcal{M}$-solution for T, that is, $\bigcap\{Q(T') \mid T'$ is an $\mathcal{M}$-solution for $T\}$. Note that our queries return sets of tuples, so we can talk about the intersection operator.

It was shown in [6, 4] that, for conjunctive tree queries and mappings using nested-relational DTDs, computing certain answers for a given source tree T is solvable in polynomial time. Thus, for the classes of mappings and queries we consider, there is no complexity mismatch between relational and XML data exchange. The next theorem shows that our translation is correct with respect to query answering, that is, our **Requirement 5** is satisfied.

THEOREM 5.4. *Let $\mathcal{M} = (D_S, D_T, \Sigma)$ be an XML schema mapping. Then, for every XML tree T that satisfies D_S and for every conjunctive tree query Q, the certain answers of Q for T under $\mathcal{M}$ and the certain answers of* INLQUERY(Q, D_T) *for* INLDOC(T, D_S) *over* INLMAP$(\mathcal{M}, D_S, D_T)$ *coincide:*

$$\text{CERTAIN}_{\mathcal{M}}(Q,T) = \text{CERTAIN}_{\text{INLMAP}(M)}(\text{INLQUERY}(Q, D_T), \text{INLDOC}(T, D_S)).$$

This result, combined with the standard procedure of evaluating conjunctive queries in relational data exchange, also gives us an algorithm for computing certain answers.

COROLLARY 5.5. *Under the conditions of Theorem 5.4,* CERTAIN$_{\mathcal{M}}(Q,T)$ *can be obtained by the following procedure:*

1. *run* INLQUERY(Q, D_T) *on an* INLMAP$(\mathcal{M}, D_S, D_T)$*-universal solution for* INLDOC(T, D_S)*;*
2. *discard all tuples that contain null values.*

6. XML-to-XML Queries

Up to now, we have only considered queries that output tuples of attribute values. In this section, we shall focus on proper XML-to-XML query languages, that is, queries that output XML trees.

Some immediate questions arise when dealing with these formalisms in a data exchange context. Let $\mathcal{M} = (D_S, D_T, \Sigma)$ be an XML schema mapping, T be a tree conforming to D_S, and $\mathcal{Q}$ be an XML-to-XML query. Since the evaluation of $\mathcal{Q}$ over T returns an XML tree, we cannot define certain answers as $\bigcap\{\mathcal{Q}(T')) \mid T'$ is a solution for $T\}$, since the meaning of the intersection operator for XML documents is not clear.

To overcome this problem, we use recent results from [11] which showed how to define certain answers for queries returning XML trees, and how to use them in data exchange context. The key idea of [11] is to use tree patterns to define information contained in documents, and to use them to represent compactly the certain knowledge from the collection $\{\mathcal{Q}(T')) \mid T'$ is a solution for $T\}$. More precisely, if Π is a set of tree patterns which are matched by every tree $Q(T')$, we look for a small set Π_0 of patterns that is equivalent to Π as a description of certain answers. By equivalence we mean that a tree matches every pattern in Π iff it matches every pattern in Π_0. If the set Π_0 is finite, then its patterns can be put together to create a tree with nulls, which we then view as the certain answer.

We shall not need additional details of the construction; instead we shall use a result from [11] that tells us how certain answers can be computed for a specific XML-to-XML query language. The language, called TQL (to be defined shortly), is inspired by XQuery's FLWR expressions, and is restricted to positive features (i.e., no negation). The key result from [11] is the following:

PROPOSITION 6.1 ([11]). *Let $\mathcal{M} = (D_S, D_T, \Sigma)$ be an XML schema mapping, Q a TQL query, and T a tree that conforms to D_S. If T' is an $\mathcal{M}$-universal solution for T, then* CERTAIN$_{\mathcal{M}}(Q,T) = Q(T')$.

Given this result, we now do the following. We provide a formal definition of the TQL language of [11], which can express XML-to-XML analogs of relational conjunctive queries. We then show how to adapt the machinery we have previously developed for evaluating certain asnwers over a universal solution. Note that for this new translation, a TQL query Q returning trees needs to be translated into a *set* of relational queries generating views that define the shredding of the tree $Q(T)$.

6.1 TQL queries

TQL queries [11] are inspired by the FLWR (for-let-where-return) expressions of XQuery [31], but they only use positive features. The key construct is for $\pi(\bar{x})$ return $q(\bar{x})$, where $\pi(\bar{x})$ is a pattern and $q(\bar{x})$ is a query that defines a forest expression. Formally, the syntax of forest expressions is

$$\begin{array}{rcl} q(\bar{x}) & ::= & \epsilon \\ & | & \ell(\bar{a}, \bar{x}')[q'(\bar{x}'')] \\ & | & q'(\bar{x}'), q''(\bar{x}'') \\ & | & \textsf{for } \pi(\bar{a}, \bar{x}, \bar{y}) \textsf{ return } q'(\bar{x}, \bar{y}) \end{array}$$

where ℓ ranges over node labels, $\bar{a}$ over constant attribute values, and $\bar{x}$ etc are tuples of variables.

A TQL query Q is an expression of the form $r[q]$, where q is a forest expression without variables. To define the semantics of this language, we first define inductively the forest $[\![q(\bar{x})]\!]_{T,v}$, for a valuation v of all variables in $\bar{x}$ as attribute values. We use the notation $\ell(\bar{a})[f]$ for a tree whose root is labeled ℓ and carries a tuple of attributes $\bar{a}$, and f is the forest of subtrees below the root.

$$\begin{array}{rcl} [\![\epsilon]\!]_{T,v} & = & \epsilon \text{ (empty forest)} \\ [\![\ell(\bar{a}, \bar{x}')[q'(\bar{x}'')]]\!]_{T,v} & = & \ell(\bar{a}, v(\bar{x}'))\left[[\![q']\!]_{T,v}\right] \\ [\![q'(\bar{x}), q''(\bar{x}'')]\!]_{T,v} & = & [\![q']\!]_{T,v} \cup [\![q'']\!]_{T,v} \end{array}$$

$$[\![\textsf{for } \pi(\bar{a}, \bar{x}, \bar{y}) \textsf{ return } q'(\bar{x}, \bar{y})]\!]_{T,v} =$$

$$\bigcup\left\{[\![q']\!]_{T,v'} \mid v' \text{ extends } v \text{ and } T \models \pi(\bar{a}, v'(\bar{x}), v'(\bar{y}))\right\}$$

For a tree T and a query $Q = r[q]$, the evaluation $Q(T)$ of Q over T is defined as the tree $r[[\![q]\!]_T]$, i.e., the forest $[\![q]\!]_T$ under root r.

EXAMPLE 6.2. Recall the tree T from figure 1(a). The tree T' from figure 3(a) can also be obtained as the transformation $Q(T)$ resulting from the evaluation of a TQL query Q over T, where $Q = r[q]$, and q is defined as

$$\textsf{for } r/\mathit{book}(x)/\mathit{author}/\mathit{name}(y) \textsf{ return } \mathit{writer}[\mathit{name}(y), \mathit{work}(x)] \quad (3)$$

For the sake of readability, we use the / operator to denote the child axis in tree patterns. □

6.2 Inlining TQL queries

If Q is a TQL query, then, to be able to define its inlining translation, we need to specify a DTD for trees $Q(T)$. Note that TQL queries define the shape of their outputs, and at the same time do not put restrictions on the number of appearances of labels. Hence it is natural to define the DTD for outputs of Q as a *starred* DTD D_Q, whose shape is determined by Q, and where each element type except the root occurs under the Kleene star.

More precisely, for a forest expression q, we define a forest F_q inductively as follows: F_ε is the empty forest; $F_{\ell[q']}$ is $\ell[F_{q'}]$; $F_{q' \cup q''} = F_{q'} \cup F_{q''}$, and $F_{\textsf{for } \pi \textsf{ return } q'} = F_{q'}$. For $Q = r[q]$ we let $T_Q = r[F_q]$.

Then D_Q is a non-recursive DTD that has a rule $p \to c_1^* \cdots c_n^*$ for each node p in T_Q with children labelled $c_1, \ldots, c_n$. As usual, we require that D_Q be acyclic and we assume without loss of generality that $G(D_Q)$ is a tree.

EXAMPLE 6.3.[Example 6.2 continued] Recall query $Q = r[q]$. Then, T_Q is the XML tree given by $r[\mathit{writer}[\mathit{name}, \mathit{work}]]$, and thus D_Q contains productions $r \rightarrow writer^*$, $writer \rightarrow name^* work^*$, $name \rightarrow \epsilon$ and $work \rightarrow \epsilon$. □

Before showing the algorithm INLTQL, we need to introduce some features that will be used in the algorithm. Consider again query (3) and DTD D_Q in examples 6.2 and 6.3. For each pair of attributes that satisfy $r/book(x)/author/name(y)$, the query Q creates a subtree $writer[name(y), work(x)]$ in the tree $Q(T)$. Thus, the relational translation would need to create one tuple in the relations corresponding to *writer*, *name* and *work* for each pair of attributes x, y that satisfy the relational translation of the pattern $r/book(x)/author/name(y)$ in the instance INLDOC(T).

Thus, in the relational translation we need a way to associate each particular *writer* wih a particular *name* and *work*. One possible way of doing this is by creating a (Skolem) function f that associates with each pair $(name, work)$ a unique identifier for the corresponding *writer*. Thus, the function f must be defined in such a way that $f(book, name)$ is different for each different pair $(name, work)$. We enforce this requirement by letting each term $f(\bar{a})$ represent a distinct constant $c_{f(\bar{a})}$.

We will define our translation algorithm inductively. The key procedure TQLSTEP for the inductive step is described below. Its inputs, in addition to a query and a DTD, include a conjunctive query corresponding to the conjunction of patterns in the query, and a function term corresponding to the parent in the tree $Q(T)$ (for example, when creating views for relation R_{work}, we would input the identifier $f(x, y)$ of the parent node labelled *writer*). This is illustrated by the example below.

EXAMPLE 6.4.[Example 6.3 continued] Assume that query $Q = r[q]$ of examples 6.2 and 6.3 is posed over T under schema D. The following views define the translation for Q:

$R_r(f_r)$:= true
$R_{writer}(f_{writer}(x, y), f_r)$:=
  INLQUERY$(r/book(x)/author/name(y), D)$
$R_{name}(f_{name}(x, y), f_{writer}(x, y), y)$:=
  INLQUERY$(r/book(x)/author/name(y), D)$
$R_{work}(f_{work}(x, y), f_{writer}(x, y), x)$:=
  INLQUERY$(r/book(x)/author/name(y), D)$

Notice how each tuple in relations R_{name} and R_{work} is set to reference the correct tuple in relation R_{writer}. □

```
Procedure TQLSTEP($Q$, $D$, $\varphi$, $t$)
  Input  : A forest expression $q(\bar{x})$, a DTD $D$, a conjunctive
           query $\varphi(\bar{x})$ and a skolem term $t$.
  Output: A set of views over INLSCHEMA$(D_Q)$.

  if $q(\bar{x}) ::= \epsilon$ then
      return $\emptyset$
  else if $q(\bar{x}) ::= q'(\bar{x}'), q''(\bar{x}'')$ then
      return TQLSTEP$(q', D, \varphi, t) \cup$ TQLSTEP$(q'', D, \varphi, t)$
  else if $q(\bar{x}) ::= \ell(\bar{a}, \bar{x}')[q'(\bar{x}'')]$ then
      Let $f$ be a fresh skolem function. Define view $V$ as
      $R_\ell(f(\bar{x}), t, \bar{a}, \bar{x}')$ := INLQUERY$(\varphi, D)$, or just
      $R_\ell(f(), t, \bar{a})$ := true if $\varphi = \emptyset$.
      return $\{V\} \cup$ TQLSTEP$(q', D, \varphi, f(\bar{x}))$
  else if $q(\bar{x}) ::=$ for $\pi(\bar{a}, \bar{x}, \bar{y})$ return $q'(\bar{x}, \bar{y})$ then
      Let $\varphi'(\bar{a}, \bar{x}, \bar{y}) = \varphi(\bar{x}) \wedge \pi(\bar{a}, \bar{x}, \bar{y})$.
      return TQLSTEP$(q', D, \varphi', t)$
```

To define the inlining translation INLTQL, we simply need a Skolem term for the root of the tree, as the basis for the inductive procedure TQLSTEP.

```
Procedure INLTQL($Q$, $D$)
  Input  : A TQL query $Q = r[q]$ and a DTD $D$.
  Output: A set of views over INLSCHEMA$(D_Q)$.

  Create a 0-ary function $f_r$.
  return TQLSTEP$(Q, D, \emptyset, f_r())$
```

A TQL query Q is compatible with a DTD D if all the patterns used in Q are compatible with D. The following proposition shows that INLTQL satisfies an analog of **requirement 3** for queries that outputs trees.

PROPOSITION 6.5. *Given a DTD D, a TQL query Q compatible with D, and and a tree T that conforms to D, we have that* INLDOC$(Q(T), D_Q) =$ INLTQL$(Q, D)($INLDOC$(T))$, *up to renaming of nulls.*

That is, the set of views INLTQL(Q, D) applied to the inlining of T yields the same answer as the inlining of $Q(T)$.

Translating relations back into XML

To complete the translation, we need an algorithm to publish back the relational data as an XML document. This is done by means of the algorithm PUBREL. We say that an instance I of INLSCHEMA(D) *D-represents* a tree T that conforms to D if $I =$ INLDOC(T, D).

```
Procedure PUBREL($D$,$I$)
  Input  : A DTD $D$ and an instance $I$ that $D$-represents some
           tree.
  Output: An XML tree $T$ that is $D$-represented by $I$.

  for each node $\ell$ of $G(D)$, traversed as Depth-first-search do
      for each tuple $t$ of $R_\ell$ in $I$ with elements $n$, $\bar{a}$ and $n'$
      corresponding to attributes $id_n$, $A_D(\ell)$ and $id_{\mu(n)}$. do
          for every non-starred node $\ell'$ of $G(D)$ such that
          $\mu(\ell') = \ell$, and elements $n''$ and $\bar{b}$ in $t$ corresponding
          to attributes $id_{\ell'}$ and $A_D(\ell')$ do
              Create a node $n''$ in $T$ labelled $\ell'$, with attributes $\bar{b}$,
              in a parent-child scheme that resembles $G(D)$.
          endfor
          Add to $T$ a node $n$ labelled $\ell$, with attributes $\bar{a}$, with $n'$
          as ancestor, according to the parent-child sequence
          defined by $G(D$ (no parent if $\ell = r$).
      endfor
  endfor
  return $T$
```

This algorithm will only work for relational instances that represent shredded documents. The following proposition shows its correctness.

PROPOSITION 6.6. *Given a DTD D and a relational instance I of* INLSCHEMA(D), *it is the case that* INLDOC(PUBREL$(D, I)) = I$.

6.3 TQL queries in XML data exchange

Combining the previously mentioned result in [11] with the correctness of the algorithms we presented we conclude that **requirements 1-5** are satisfied for data exchange with XML-to-XML queries:

THEOREM 6.7. *Let $\mathcal{M} = (D_S, D_T, \Sigma)$ be an XML schema mapping. Then, for every XML tree T that satisfies D_S and for every TQL query Q, the certain answers of Q for T under $\mathcal{M}$ and the certain answers of* $\textsc{InlTQL}(Q, D_T)$ *for* $\textsc{InlDoc}(T, D_S)$ *over* $\textsc{InlMap}(\mathcal{M}, D_S, D_T)$ *coincide:*

$$\textsc{InlDoc}(\textsc{certain}_{\mathcal{M}}(Q,T), D_Q) \;=\; \textsc{certain}_{\textsc{InlMap}(M)}(\textsc{InlTQL}(Q, D_T), \textsc{InlDoc}(T, D_S)).$$

Remark: The notion of certain answers naturally (component-wise) extends to queries computing multiple relations.

Theorem 6.7 and Proposition 6.6 give us a way of computing $\textsc{certain}_{\mathcal{M}}(Q,T)$. First, compute $\textsc{certain}_{\textsc{InlMap}(M)}(\textsc{InlTQL}(Q, D_T), \textsc{InlDoc}(T, D_S))$ by materializing views $\textsc{InlTQL}(Q, D_T)$ over the canonical solution for $\textsc{InlDoc}(T, D_S)$ and then use the procedure PubRel to output it as the tree $\textsc{certain}_{\mathcal{M}}(Q,T)$.

7. Concluding Remarks

Our technique provides a relational approach to solve two of the most important problems of XML data-exchange settings: materializing solutions and answering queries. The diagram below summarizes this. In a pure XML setting, we can start with a document T and use a mapping $\mathcal{M}$ to find a (universal) solution T'_{univ}, over which we can then answer a query Q to produce certain answers.

$$\begin{tikzcd}
T \arrow[r, "\mathcal{M}"] \arrow[d, "\textsc{InlDoc}"] & T'_{\text{univ}} \arrow[r, "Q"] \arrow[d, "\textsc{InlDoc}"] & \text{certain answer} \arrow[d, equal] \\
R \arrow[r, "\textsc{InlMap}(\mathcal{M})"] & R'_{\text{univ}} \arrow[r, "\textsc{InlQuery}(Q)"] & \text{certain answer}
\end{tikzcd}$$

Using the translation InlDoc of documents, we generate a relational instance R, on which the translation of the mapping $\textsc{InlMap}(\mathcal{M})$ generates a universal solution R'_{univ}. This solution is a shredding, via InlDoc, of a universal XML solution, and also conforms to the shredding of source DTD. Finally, we apply the standard technique [12] for evaluating queries in relational data exchange to the query translation $\textsc{InlQuery}(Q)$ or $\textsc{InlTQL}(Q)$ to produce the correct answers, in the latter case with the possibility of using PubRel to publish back the results into XML.

Implementing our proposed algorithms for use in practical systems would be straightforward using their specifications given in this paper. A natural next step is to evaluate XML data-exchange systems using relational data storage and implementations of our algorithms. We are currently working on this direction.

We finish with a remark about the possibility of allowing operators ? and $+$ in DTDs, as well as a choice operator for representing multiple choices. We say that a non-recursive DTD D is an *extended nested relational* DTD if all rules of D are of the form $\ell \rightarrow \tilde{\ell}_0 \ldots \tilde{\ell}_m$, or $\ell \rightarrow \ell_0 + \ldots + \ell_m$, where all the ℓ_i's and $\tilde{\ell}_i$'s are distinct, and each $\tilde{\ell}_i$ is one of ℓ_i, $\ell_i?$, ℓ_i^* or ℓ_i^+ (as usual $\ell?$ stands for $\ell|\epsilon$ and ℓ^+ for $\ell\ell^*$).

The procedure InlSchema can be extended to these DTDs. For each element ℓ that is under the operator ?, the transformation creates a special relation ℓ that references the relation of the nearest appropriate ancestor of ℓ. Furthermore, the transformation for a rule of the form $\ell_1 \rightarrow \ell_2^+$ can be defined by including a dependency that ensures that there is at least one tuple in the relation R_{ℓ_2} for each tuple in R_{ℓ_1}. Finally, for the choice operator $\ell \rightarrow \ell_0 + \ldots + \ell_m$ the transformation would create one relation R_ℓ for each possible choice of $\ell_0, \ldots, \ell_m$. Then, it is possible to extend all the procedures in a way that still satisfies **requirements 1-5** under extended nested relational DTDs.

Acknowledgments Partial support provided by EPSRC grant G049165 and FET-Open Project FoX, grant agreement 233599.

8. References

[1] S. Abiteboul, L. Segoufin and V. Vianu. Representing and querying XML with incomplete information. *TODS*, 31(1) (2006), 208-254

[2] F. Afrati, C. Li, V. Pavlaki. Data exchange in the presence of arithmetic comparisons. In *EDBT 2008*, pages 487-498.

[3] S. Amano, C. David, L. Libkin, F. Murlak. On the tradeoff between mapping and querying power in XML data exchange. In *ICDT 2010*.

[4] S. Amano, L. Libkin, F. Murlak. XML schema mappings. In *PODS 2009*, pages 33-42.

[5] S. Amer-Yahia, S. Cho, L. Lakshmanan, D. Srivastava. Tree pattern query minimization. *VLDB J.* 11 (2002), 315–331.

[6] M. Arenas, L. Libkin. XML data exchange: consistency and query answering. *J. ACM* 55(2): (2008).

[7] A. Balmin and Y. Papakonstantinou. Storing and querying XML data using denormalized relational databases. *VLDB J.*, 14:30–49, 2005.

[8] P. Barceló. Logical foundations of relational data exchange. *SIGMOD Record* 38(1): 49–58 (2009).

[9] P. A. Bernstein, S. Melnik. Model management 2.0: manipulating richer mappings. *SIGMOD'07*, pages 1-12

[10] H. Björklund, W. Martens, T. Schwentick. Conjunctive query containment over trees. In *DBPL 2007*, pages 66-80.

[11] C. David, L. Libkin, F. Murlak. Certain answers for XML queries. In *PODS 2010*, pages 191-202.

[12] R. Fagin, P. G. Kolaitis, R. Miller, L. Popa. Data exchange: semantics and query answering. *TCS* 336(1): 89–124 (2005).

[13] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM TODS* 30(1):174–210, 2005.

[14] D. Florescu, D. Kossman. Storing and querying XML data using a RDBMS *IEEE Data Engineering Bulletin* 22(3): 27–34, 1999.

[15] A. Fuxman, M. Hernández, H. Ho, R. Miller, P. Papotti, L. Popa. Nested mappings: schema mapping reloaded. *VLDB'06*, pages 67-78.

[16] G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *JACM* 53(2): 238-272, 2006.

[17] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE TKDE*, 19:1381–1403, 2007.

[18] M. Hernández, H. Ho, L. Popa, A. Fuxman, R. Miller, T. Fukuda, P. Papotti. Creating nested mappings with Clio. In *ICDE 2007*.

[19] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, C. Yu. TIMBER: A native XML database. *VLDB Journal* 11(4): 274-291, 2002.

[20] N. Klarlund, T. Schwentick, D. Suciu. XML: model, schemas, types, logics, and queries. In *Logics for Emerging Appl. of Databases 2003*.

[21] Ph. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS 2005*, pages 61-75.

[22] R. Krishnamurthy, R. Kaushik, J. Naughton. XML-to-SQL query translation literature: state of the art and open problems. In *XSym'03*.

[23] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML views as integrity constraints and their use in query translation. In *ICDE'05*.

[24] L. Lakshmanan, G. Ramesh, H. Wang, Z. Zhao. On testing satisfiability of tree pattern queries. *VLDB 2004*, pages 120–131.

[25] G. Mecca, P. Papotti, S. Raunich. Core schema mappings. In *SIGMOD 2009*, pages 655-668.

[26] R. Miller, M. Hernández, L. Haas, L. Yan, H. Ho, R. Fagin, L. Popa. The Clio project: managing heterogeneity. *SIGMOD Record*. 30 (2001).

[27] L. Popa, Y. Velegrakis, R. Miller, M. Hernández, R. Fagin. Translating Web data. In *VLDB 2002*, pages 598–609.

[28] J. Shanmugasundaram, et al. Relational databases for querying XML documents: limitations and opportunities. *VLDB'99*, pages 302-314.

[29] J. Shanmugasundaram, et al. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30:20–26, 2001.

[30] I. Tatarinov, et al. Storing and querying ordered XML using a relational database system. In *SIGMOD'02*, pages 204–215.

[31] XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery.

[32] C. Yu, L. Popa. Constraint-based XML query rewriting for data integration. In *SIGMOD 2004*, pages 371-382.

[33] C. Zhang, et al. On supporting containment queries in relational database management systems. In *SIGMOD'01*, pages 425–436.