XML Publishing: Bridging Theory and Practice

Wenfei Fan[⋆]
University of Edinburgh and Bell Laboratories

Abstract. Transforming relational data into XML, as known as *XML publishing*, is often necessary when one wants to exchange data residing in databases or to create an XML interface of a traditional database. This paper aims to provide an overview of recent advances in XML publishing. We present a notion of publishing transducers recently developed for studying the expressive power and complexity of XML publishing languages. In terms of publishing transducers we then characterize XML publishing languages being used in practice. In addition, we address dynamic aspects of XML publishing, namely, incremental maintenance and update management of XML views published from relational data.

1 Introduction

While most data is currently residing in relational databases, it is increasingly common for one to exchange the data in XML format, or to build an XML interface of the databases. This highlights the need for transforming relational data into XML, as known as *XML publishing of relational data*.

In response to the need, a variety of XML publishing languages have been developed [2, 3, 15, 26], and are rapidly being introduced into commercial products [18, 21, 24]. An XML publishing language is essentially a view definition language, for specifying XML views of relational data. Just like their relational counterparts, associated with XML publishing languages are a number of fundamental questions in connection with their complexity and expressiveness. These questions are not only of theoretical interest, but are also important in practice for both users and designers of XML publishing languages. Given a host of XML publishing languages, a user wants to decide which one to choose: is an XML view expressible in certain languages but not definable in others? Which language is "better" than others when it comes to evaluation cost? To support recursively-defined XML views in a publishing language, database vendors may want to know whether or not certain high-end DBMS features are a must: is it necessary to upgrade the DBMS to support linear recursion of SQL'99?

This paper aims to provide a synergy between theory and practice by answering these questions for XML publishing languages supported by commercial products or research prototype systems: SQL/XML of IBM DB2 XML Extender [18] and Oracle 10g XML DB [24], FOR-XML and XSD of SQL Server 2005 [21], DAD of DB2 XML Extender, DBMS_XMLGEN of XML DB, as well as XPERANTO [26], TreeQL of SilkRoute [15, 2], and ATG of PRATA [3]. We evaluate these languages in terms of their expressive power and complexity, by leveraging a notion of publishing

[⋆] Supported in part by EPSRC GR/S63205/01, GR/T27433/01, EP/E029213/1 and BBSRC BB/D006473/1.

transducers recently proposed in [13]. We characterize these languages in terms of various classes of publishing transducers, for which the complexity bounds and expressive power have been established in [13].

Another aim of the paper is to promote the study of dynamic aspects of XML publishing. Since XML publishing actually defines XML views of relational data, for all the reasons that the incremental update and view update problems are important for database views, efficient incremental maintenance and update management also deserve a full treatment for XML publishing. Unfortunately we are aware of the support of this functionality only in research prototype systems (*e.g.,* PRATA [6, 11]), but currently not in any of the commercial systems.

The remainder of the paper is organized as follows. In Section 2, we discuss various dichotomies for assessing XML publishing languages. In Section 3 we present XML publishing transducers and give an account of results about their complexity bounds and expressive power. In Section 4 we characterize the XML publishing languages mentioned above in terms of publishing transducers. In Section 5, we address the incremental update and view update problems for XML publishing. Finally, we identify open research issues in Section 6.

2 XML Publishing

An XML document is typically modeled as a node-labeled, ordered, unranked tree. Given a relational schema R, XML publishing is to define an XML view, *i.e.,* a mapping τ such that for any instance I of R, $\tau(I)$ is an XML tree.

Example 1. Consider a relational schema R_0 (with keys underlined): course($\underline{\text{cno}}$, title, type), prereq($\underline{\text{cno1, cno2}}$). A database instance D_0 of R_0 maintains course data classified into "regular" and "project" *type*, and a relation *prereq* in which a tuple *(c1, c2)* indicates that *c2* is an *immediate prerequisite* of *c1*. Note that the transitive closure of *prereq* gives the prerequisite hierarchy of the courses.

One may want to define the following XML views of the relational database:

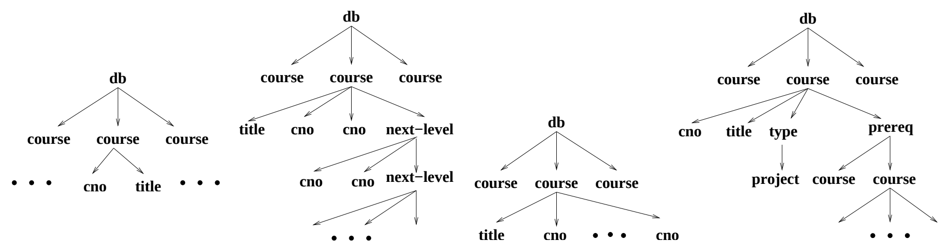
(1) As depicted in Fig. 1(a), view τ_1 is a tree of depth two, containing the list of all courses in D_0 that do not have DB as its immediate prerequisite, *i.e.,* for any such *course* c, (c, c') is not in *prereq* if the title of c' is DB.

(2) As shown in Fig. 1(b), view τ_2 contains the list of all courses in D_0. Under each course c are its *title* and the list of *cno*'s of its immediate prerequisites, followed by an element *next-level* under which are the immediate prerequisites of the *cno* children of c, and so on, until all the prerequisites of c are listed.

(3) As depicted in Fig. 1(c), view τ_3 is a tree of depth two, containing the list of all courses in D_0. Below each *course* c is its *cno*, followed by the list of all the *cno*'s that appear in the prerequisite *hierarchy* of c.

(4) As shown in Fig. 1(d), view τ_4 is an XML tree that is required to conform to the DTD d_0 below (the definition of elements whose type is PCDATA is omitted):

```
<!ELEMENT db      (course*)>
<!ELEMENT course  (cno, title, type, prereq)>
```



(a) XML view τ_1 (b) XML view τ_2 (c) XML view τ_3 (d) XML view τ_4

Fig. 1. Example XML publishing

```
<!ELEMENT type    (regular | project)>
<!ELEMENT prereq  (course*)>
<!ELEMENT regular (empty)>   /* similarly for project */
```

We may find it difficult to express (1) τ_1 in XSD of SQL Server 2005 [21], RDB_mapping of IBM DB2 XML Extender [18] and TreeQL [2], (2) τ_2 in any language except ATG [3], (3) τ_3 in any language except DBMS_XMLGEN of Oracle 10g XML DB [24] and ATG, and (4) τ_4 in any language except ATG to guarantee the conformance to D_0. However we are not sure whether it is because we do not know the languages well enough, or due to the limitations of the languages. $\square$

To answer this question we study a variety of factors that may impact the expressive power of an XML publishing language. A publishing language typically specifies the behaviors of a middleware controller with a limited query interface to relational sources. An XML view defined in such a language builds an output tree top-down starting from the root: at each node it issues queries to a relational database I, generates the children of the node using the query results, and iteratively expands the subtrees of those children inductively. It may (implicitly) store intermediate query results in registers associated with nodes and pass the information downward to control subtree generation [2, 3, 18, 21, 24, 26]. It may also allow *virtual* tree nodes [2, 3] that will be removed from the output tree to express, *e.g.,* XML entities. In addition, it may be recursively defined, capable of generating XML trees for which the depth cannot be determined at compile time. Finally, it may encode a DTD to guarantee that the output trees conform to the predefined DTD. These motivate us to consider the following dichotomies:

– CQ, FO vs. FP: the relational query language in which queries on relational data are expressed. We consider conjunctive queries (CQ), first-order queries (FO) and (inflationary) fixpoint queries (FP). For example, view τ_1 requires an FO query and cannot be expressed in languages with CQ queries only.
– Relation vs. tuple: registers that store intermediate results. Some languages store a finite *relation* in a register while others allow a single *tuple.* View τ_2 is definable only in languages that support relation registers.
– Virtual vs. normal: the types of nodes. Languages may or may not allow *virtual* nodes that will be removed from the output tree. In a language that does not support FP (*e.g.,* SQL'99), view τ_3 is definable only if virtual nodes are allowed, by making the *next-level* nodes of τ_2 virtual.
– Recursive vs. nonrecursive: whether or not views can be recursively defined. For example, τ_2 and τ_4 are recursively defined: the depth of a *course* sub-tree

in these views is determined by its prerequisite hierarchy in D_0.

- DTD-directed or not: whether or not output XML trees are guaranteed to conform to a predefined DTD. In practice, XML publishing is often directed by a type, typically a DTD, as shown by τ_4. A community or industry agrees on a certain DTD, and subsequently all members of the community create XML views of their relational data that conform to the predefined DTD [2].

Different combinations of these parameters yield a spectrum of XML publishing languages with quite different expressive power and complexity.

3 Publishing Transducers

We now present publishing transducers introduced in [13], which allow us to characterize the complexity and expressive power of existing XML publishing languages, as well as their equivalence and separation.

3.1 Definition of Publishing Transducers

Let R be a relational schema, $\mathcal{L}$ a relational query language, and Σ a set of XML tags. A publishing transducer is a finite state machine that, given a database instance I of R, generates an XML tree with elements labeled with tags in Σ, *top-down* starting from the root. To do this, with each element labeled $a \in \Sigma$ in the XML tree, it associates a register Reg_a, storing intermediate result as a relation of a fixed arity. At each a-node v, the transducer extracts data from the underlying database I and the intermediate result in Reg_a, via a query in $\mathcal{L}$, and spawns the children of v using the data. This is directed by a transduction rule, which is uniquely determined by the tag a and the current state of the machine. In contrast to tree recognizers (see [16]) and the automata for querying XML [22, 23], which operate on an *existing tree* and either accept the tree or select a set of nodes from the tree, a publishing transducer does not take a tree as input; instead, it builds a *new XML tree* based on the data from a relational source.

Definition 1. *A* publishing transducer *for a relational schema R is defined to be $\tau = (Q, \Sigma, q_0, \delta)$, where Q is a finite set of* states*; Σ is a finite alphabet of* tags*; q_0 is the* start state *associated with the root tag $r \in \Sigma$; and δ is a finite set of* transduction rules*: for each $(q, a) \in Q \times \Sigma$, there is a unique rule*

$$(q, a) \quad \rightarrow \quad (q_1, a_1, \phi_1(\bar{x}_1; \bar{y}_1)), \ldots, (q_k, a_k, \phi_k(\bar{x}_k; \bar{y}_k)).$$

Here $k \geq 0$, $a_1, \ldots, a_k$ are distinct tags in Σ, $(q_i, a_i) \in Q \times \Sigma$ for $i \in [1, k]$, and $\phi_i \in \mathcal{L}$ is a relational query from R and Reg_a to Reg_{a_i}. $\square$

We next give basic properties and the semantics of publishing transducers.

Deterministic. A publishing transducer is deterministic: for each $(q, a) \in Q \times \Sigma$, there is a unique transduction rule, except that for the start state q_0, for which only the rule for (q_0, r) is defined. Furthermore, (a) q_0 and r do not appear in the right-hand side of any rule; (b) *text* is a special "tag" in Σ, and the right-hand side of the rule for (q, text) is empty, *i.e., text* nodes do not have any children.

Tuple vs. relation register. The $\mathcal{L}$ query $\phi_i(\bar{x}_i; \bar{y}_i)$ extracts data from I and Reg_a. The result of the query is grouped by attributes $\bar{x}_i$, yielding sets of tuples. For each set S_j, a distinct a_i child is created, carrying S_j as the content of its register Reg_{a_i}. These a_i children are ordered based on an implicit ordering on the domain of data. When $|\bar{y}_i| = 0$, *i.e.,* when the result is grouped by the entire tuple, each register Reg_{a_i} holds a *single* tuple and is thus a *tuple register.*

Transduction. Initially, τ constructs a tree t consisting of a single node labeled (q_0, r) with an empty register. At each step, τ expands t by *simultaneously* operating on the leaf nodes of t. At each leaf u labeled (q, a), τ generates new nodes by finding the rule for (q, a) from δ, issuing queries $\phi_1(\bar{x}_1; \bar{y}_1), \ldots, \phi_k(\bar{x}_k; \bar{y}_k)$ embedded in the rule to the database I and the register $\mathsf{Reg}_a(u)$ associated with u. For each $i \in [1, k]$, the a_i children and their associated register Reg_{a_i} are produced as described above. These yield the children of u characterized by a regular expression $a_1^* \ldots a_k^*$. The transformation proceeds until a stop condition is satisfied at all the leaf nodes. A stop condition is one of the following.

1. There is a node v on the path from the root to u such that v and u have the *same* state q, tag a, and $Reg_a(v) = Reg_a(u)$. Since the subtree rooted at u is uniquely determined by $q, a, Reg_a(u)$ and I, this asserts that the tree t will not expand at u if the expansion *adds no new information* to the tree.
2. The query $\phi_j(\bar{x}_j; \bar{y}_j)$ evaluates to empty for all $i \in [1, k]$.
3. The right-hand side of the rule for (q, a) is empty. This is particularly the case for $a = text$, for which u carries a string representation of $Reg_a(u)$.

These conditions ensure the termination of the transformation.

When the tree cannot be expanded further, *i.e.,* all leaf nodes satisfy a stop condition, an XML tree is generated by removing all registers and states from t. It is the output of the transducer τ, denoted by $\tau(I)$. We use $\tau(R)$ to denote the set of all XML trees generated by τ when I ranges over all instances of R.

Example 2. We define a publishing transducer $\tau_1 = (Q_1, \Sigma_1, q_0, \delta_1)$ to generate the view of Fig. 1(a), where $Q_1 = \{q_0, q\}$, $\Sigma_1 = \{db, course, cno, title, text\}$, and the root tag is db. The transduction δ_1 is defined as follows:

$\delta_1(q_0, db) \quad = (q, course,\ \phi_1(n, t;\ \emptyset))$, where
$\quad \phi_1(n, t) = \exists t_p\ (course(n, t, t_p) \wedge \neg\exists n_1, t_1, t_{p1}\ (prereq(n, n_1) \wedge course(n_1, t_1, t_{p1}) \wedge t_1 = \text{‘DB’}))$

$\delta_1(q, course) = (q, cno,\ \phi_2(n; \emptyset)),\quad (q, title,\ \phi_3(t; \emptyset)),$ where
$\quad \phi_2(n) = \exists t\ \mathsf{Reg}_c(n, t)$, and $\phi_3(t) = \exists n\ \mathsf{Reg}_c(n, t)$,

$\delta_1(q, cno) = (q, text, \phi_4(n))$, where $\phi_4(n) = \mathsf{Reg}_n(n)$ (similarly for $\delta_1(q, title)$)

$\delta_1(q, text) = .$ (empty right-hand side.)

Here registers Reg_c and Reg_n are associated with *course* and *cno* nodes, respectively. These are tuple registers: in each query $\phi(\bar{x}; \bar{y})$ in δ_1, $|\bar{y}| = 0$.

Given a database I_0 of schema R_0, τ_1 generates an XML tree as follows. First, it creates the root of a tree t labeled with (q_0, db). It then evaluates query ϕ_1 on I, and for each tuple (*cno, title*) in the result of the query, it expands the tree by spawning a *course* child of the root, carrying the tuple in its register Reg_c. For each *course* node, it generates its *cno* and *title* children by extracting relevant attribute from the tuple in Reg_c via queries ϕ_2 and ϕ_3, respectively, which in turn have a single *text* child carrying the attribute as PCDATA. The transformation stops at the *text* nodes (stop condition 3 above). Finally, it outputs an XML tree by striking out states and registers associated with the nodes in t. $\square$

Recursive transducers. Define the *dependency graph* G_τ of τ as follows. For each $(q, a) \in Q \times \Sigma$ there is a unique node $v(q, a)$ in G_τ, and there is an edge from $v(q, a)$ to $v(q', a')$ iff (q', a') is on the right-hand side of the rule for (q, a). We say that the transducer τ is *recursive* iff there is a cycle in G_τ.

Example 3. To generate the XML view of Fig. 1(b), we define a publishing transducer $\tau_2 = (Q_2, \Sigma_2, q_0, \delta_2)$, where the transduction δ_2 is defined as follows:

$\delta_2(q_0,\ db) \quad = (q, course,\ \psi_1(n, t;\ \emptyset))$, where $\psi_1(n, t) = \exists t_p\ course(n, t, t_p)$

$\delta_2(q,\ course) \ = (q, title,\ \psi_2(t; \emptyset)),\ (q, cno,\ \psi_3(n; \emptyset)),\ (q, next\text{-}level,\ \psi_4(\emptyset;\ n))$, where $\psi_2(t) = \exists n\ \mathsf{Reg}_c(n, t)$, $\psi_3(n_1) = \exists n, t(\mathsf{Reg}_c(n, t) \wedge prereq(n, n_1))$

$\delta_2(q, next\text{-}level) = (q, cno,\ \psi_5(n; \emptyset)),\ (q, next\text{-}level,\ \psi_5(\emptyset;\ n))$

where ψ_4 is identical to ψ_3 except that its result is put in a single relation ($|\bar{x}| = 0$ in ψ_4) as the content of register Reg_{nl} of the *next-level* node; in other words, Reg_{nl} is a *relation register* while Reg_n associated with *cno* is a *tuple register*; ψ_5 is the same as ψ_3 except that Reg_{nl} is used instead of Reg_c. In contrast to τ_1, τ_2 is *recursively defined*: in its dependency graph there is an edge from $v(q, next\text{-}level)$ to itself. On an instance I_0 of R_0 the transformation of τ_2 terminates due to stop condition 2, in any practical setting where no course is a prerequisite of itself. $\square$

Virtual vs. normal nodes. To incorporate virtual nodes we generalize transducers to be of the form $\tau = (Q, \Sigma, q_0, \delta, \Sigma_e)$, where Σ_e is a designated subset of Σ and $r \notin \Sigma_e$, referred to as the *virtual tags* of τ; and Q, Σ, q_0, δ are the same as in Definition 1. On a relational database I the transducer τ behaves the same as a normal transducer, except that the XML tree $\tau(I)$ is obtained as follows. For each node v in t, if v is labeled with a tag in Σ_e, we *shortcut* v by replacing v with the children of v, *i.e.,* treating these children nodes as children of the parent of v, and removing v from the tree. The process continues until no node in the tree is labeled with a tag in Σ_e.

Example 4. The XML view of Fig. 1(c) can be generated by a publishing transducer $\tau_3 = (Q_2, \Sigma_2, q_0, \delta_2, \{next\text{-}level\})$, which is identical to τ_2 given in Example 3 except that here *next-level* is treated as a virtual tag. $\square$

Different classes. We denote by $\mathrm{PT}(\mathcal{L}, S, O)$ various classes of publishing transducers. Here $\mathcal{L}$ indicates the relational query language in which queries embedded in the transducers are defined, ranging over CQ, FO and FP, all with equality '=' and inequality $\neq$. *Store S* is either *relation* or *tuple*, indicating that the trees induced by the transducers are with relation or tuple registers, respectively. As mentioned earlier, transducers with tuple registers are a special case of those with relation registers, *i.e.,* when $|\bar{y}_i| = 0$ in each query $\phi_i(\bar{x}_i; \bar{y}_i)$. *Output O* is either *normal* or *virtual*, indicating whether a transducer allows virtual nodes or not. We denote by $\mathrm{PT}_{nr}(\mathcal{L}, S, O)$ the subclass of $\mathrm{PT}(\mathcal{L}, S, O)$ consisting of all *nonrecursive* transducers. For instance, the transducers τ_1, τ_2 and τ_3 given in Examples 2, 3 and 4 are in PT_{nr}(FO, tuple, normal), PT(CQ, relation, normal) and PT(FO, relation, virtual), respectively (τ_3 is also in PT_{nr}(FP, tuple, normal)).

Classes	Equivalence	Emptiness	Membership
PT(FP, S, O)	undecidable	undecidable	undecidable
PT(FO, S, O)	undecidable	undecidable	undecidable
PT(CQ, tuple, normal)	undecidable	PTIME	Σ_2^p-complete
PT(CQ, relation, normal)	undecidable	PTIME	undecidable
PT(CQ, S, virtual)	undecidable	NP-complete	undecidable
PT$_{nr}$(FO, O, normal)	undecidable	undecidable	undecidable
PT$_{nr}$(CQ, tuple, normal)	Π_3^p-complete	PTIME	Σ_2^p-complete
PT$_{nr}$(CQ, tuple, virtual)	Π_3^p-complete	NP-complete	Σ_2^p-complete

Table 1. Complexity of decision problems (S: relation or tuple; O: normal or virtual)

3.2 Complexity and Expressiveness of Publishing Transducers

Complexity. A natural question is: does a publishing transducer for a relational schema R always terminate on all instances of R? This is answered in [13]: For any publishing transducer τ defined for schema R and for any database I of R, the transformation of τ on I always terminates, and its worst-case data-complexity is (a) EXPTIME if τ is in PT($\mathcal{L}$, S, O) and S is tuple, (b) 2EXPTIME if τ is in PT($\mathcal{L}$, S, O) and S is relation, (c) PTIME if τ is in PT$_{nr}$($\mathcal{L}$, S, O) no matter whether S is tuple or relation, where $\mathcal{L}$ ranges over CQ, FO and FP, and O is either normal or virtual. This tells us that while the presence of relation registers and recursion may complicate the transformation, relational query language $\mathcal{L}$ and virtual nodes have no impact on the worse-case data complexity.

Classical decision problems associated with transducers include the following. For a class PT($\mathcal{L}$, S, O) of publishing transducers,

(i) the *membership problem* is to determine, given an XML tree t and a transducer τ in this class, whether or not there is a database I such that $t = \tau(I)$;

(ii) the *emptiness problem* is to decide, given τ in this class, whether there is an instance I such that $\tau(I)$ is a nontrivial tree with more than one node;

(iii) the *equivalence problem* is to determine, given two transducers τ_1 and τ_2 in the class defined for *the same schema* R, whether or not $\tau_1(I) = \tau_2(I)$ for all instances I of R, *i.e.,* they produce the same trees on all the instances of R.

The analyses of these problems may tell a user, at compile time, whether or not a publishing transducer makes sense (emptiness), whether an XML tree of particular interest can be generated by a transducer (membership), and whether a transducer can replaced by a more efficient one (equivalence).

Matching upper and lower bounds are established in [13] for various classes of publishing transducers, and are summarized in Table 1.

Expressive power. A class PT$(\mathcal{L}_1, S_1, O_1)$ is *contained in* PT$(\mathcal{L}_2, S_2, O_2)$, denoted by PT$(\mathcal{L}_1, S_1, O_1) \subseteq$ PT$(\mathcal{L}_2, S_2, O_2)$, if for any τ_1 in PT$(\mathcal{L}_1, S_1, O_1)$ defined for a relational schema R, there exists τ_2 in PT$(\mathcal{L}_2, S_2, O_2)$ for the same R such that $\tau_1(I) = \tau_2(I)$ for all instances I of R. The two classes are *equivalent* in expressive power, denoted by PT$(\mathcal{L}_1, S_1, O_1) =$ PT$(\mathcal{L}_2, S_2, O_2)$, if PT$(\mathcal{L}_1, S_1, O_1)$ $\subseteq$ PT$(\mathcal{L}_2, S_2, O_2)$ and PT$(\mathcal{L}_2, S_2, O_2) \subseteq$ PT$(\mathcal{L}_1, S_1, O_1)$. A class PT$(\mathcal{L}_1, S_1, O_1)$ is *properly contained in* PT$(\mathcal{L}_2, S_2, O_2)$ if PT$(\mathcal{L}_1, S_1, O_1) \subseteq$ PT$(\mathcal{L}_2, S_2, O_2)$ but PT$(\mathcal{L}_1, S_1, O_1) \neq$ PT$(\mathcal{L}_2, S_2, O_2)$.

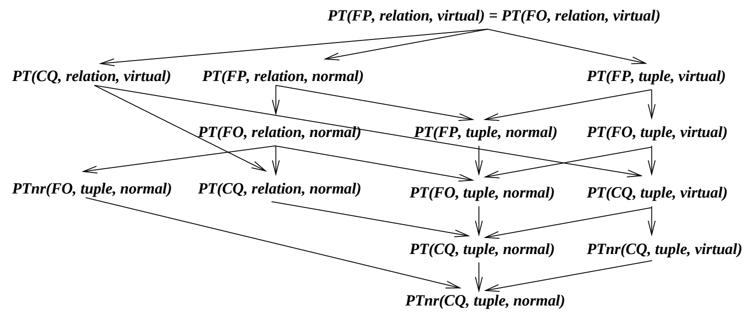


Fig. 2. Containment of various classes of XML publishing transducers

Containment. A containment hierarchy on various classes of publishing transducers is developed in [13], and is shown in Fig. 2. The containment is proper except that PT(FO, tuple, virtual) = PT(FP, tuple, virtual) if PTIME = NLOGSPACE. Figure 2 tells us that SQL'99 does not increase expressive power over SQL to a publishing language that supports virtual nodes, recursion and relation registers.

DTD conformance. It is known [23] that a set of unranked trees is regular iff it is MSO definable, and that a set of trees is MSO definable iff it is the set of trees recognized by a specialized DTD [25].

A DTD d' over Σ is defined by a set of rules of the form $a \to \alpha$, where a is a tag in Σ and α is a regular expression over Σ. An XML tree t conforms to d' iff for each a-element v in t, the list of the labels of the children of v is a word in α. A *normalized* DTD is a DTD in which for every rule $a \to \alpha$, α is defined as either $a_1, \ldots, a_k$ (concatenation), or $a_1| \ldots |a_k$ (disjunction) or a_1^* (Kleene closure), where a_i is an element type. It is known that every DTD can be converted to an equivalent normalized DTD in linear time [5].

A *specialized* DTD d over Σ is a triple (Σ', d', g), where $\Sigma \subseteq \Sigma'$, g is a mapping $\Sigma' \mapsto \Sigma$, and d' is a DTD over Σ'. A tree t conforms to d if there exists a Σ'-tree t' that satisfies d' and moreover, $t = g(t')$. We denote by $L(d)$ the set of all Σ-trees conforming to d. A (specialized or normal) DTD d is said to be definable in PT($\mathcal{L}$, S, O) if there exists a publishing transducer τ in the class defined for some relational schema R such that $L(d) = \tau(R)$.

It is shown [13] that every specialized DTD over Σ is definable in PT(FO, tuple, virtual), and every normalized DTD is definable in PT(FO, tuple, normal). However, there exist normal DTDs that are not definable in PT(CQ, relation, virtual). This tells us that when $\mathcal{L}$ is FO or FP, PT($\mathcal{L}$, S, virtual) is capable of defining all specialized DTDs, and thus all regular unranked trees and MSO definable trees. In addition, PT($\mathcal{L}$, tuple, normal) can define normalized DTDs. In contrast, PT(CQ, S, O) does not have sufficient power to express even DTDs.

Example 5. Recall DTD D_0 from Example 1, which is normalized. One can define τ_4 in PT(FO, tuple, normal) to generate views of the form shown in Fig. 1(d) that are guaranteed to conform to D_0. While it is trivial to enforce rules defined with concatenation and Kleene closure in D_0, *e.g., db, course, prereq*, to enforce the rule *type* $\to$ *regular* | *project*, we need to make sure that each *type* node has either a *regular* or a *project* child, but not both. This can be checked by a Boolean FO query φ. If φ is true τ_4 uses the transduction below for *type* nodes:

$\delta_4(q,\ type) = (q, regular,\ \varphi_1(t_p;\emptyset)),\ (q, project,\ \varphi_2(t_p;\emptyset))$
where $\varphi_1(t_p) = \exists n, t\ course(n, t, t_p) \wedge \mathsf{Reg}_c(n) \wedge t_p$=‘regular’; /*similarly for φ_2*/

Otherwise (φ is false) τ_4 produces a default XML tree that conforms to D_0. □

4 XML Publishing Languages in Practice

We are now ready to assess the expressive power of XML publishing languages being used in practice, in terms of various classes of publishing transducers.

SQL/XML is an SQL extension for XML, by incorporating XML publishing functions: XMLELEMENT, XMLATTRIBUTE, XMLFOREST, XMLCONCAT, XMLAGG and XMLGEN. For instance, τ_1 of Example 2 can be defined in SQL/XML as follows:

```
SELECT XMLELEMENT {NAME “course”, XMLFOREST {c.cno AS “cno”, c.title AS “title”}}
FROM course c
WHERE NOT EXISTS (SELECT c’.cno   FROM course c’, prereq p
                  WHERE p.cno1 = c.cno AND p.cno2 = c’.cno AND c’.title = ‘DB’)
```

SQL/XML has essentially the same expressive power as PT_{nr}(FO, tuple, normal). It cannot express XML views τ_2, τ_3 and τ_4 given earlier. It has been introduced into commercial products, including IBM XML Extender [18] and Oracle 10g XML DB [24]. The publishing language of XPERANTO [26] has the same expressive power as SQL/XML.

DBMS_XMLGEN is a PL/SQL package supported by Oracle 10g XML DB [24]. It extends SQL/XML by supporting SQL’99 and a function newContextFormHierarchy, which, in combination of CONNECT BY PRIOR of SQL’99, is capable of expressing recursive XML views. For example, the following defines a recursive XML view that contains the list of all courses; under each course c are the cno and title of c followed by the hierarchy of the prerequisite courses of c.

```
DBMS_XMLGEN.newContextFormHierarchy{
  SELECT XMLELEMENT {NAME “course”, XMLFOREST {c.cno AS “cno”, c.title AS “title”}},
  FROM course c
  CONNECT BY PRIOR course.cno = prereq.cno1}
```

DBMS_XMLGEN allows neither virtual nodes nor relation registers. It cannot define τ_2 or guarantee specialized DTD conformance. Furthermore, it does not have stop condition and thus cannot guarantee termination. If the stop condition is imposed, XML views definable in DBMS_XMLGEN are in PT(FP, tuple, normal).

FOR-XML and XSD are supported by SQL Server 2005 [21]. FOR-XML extends SQL as follows: it extracts data from a relational source via an SQL query, and organizes the data into a hierarchical XML view with nested FOR-XML constructs. For example, τ_1 can be defined with FOR-XML as follows:

```
SELECT c.cno AS “cno”, c.title AS “title”
FROM course c
WHERE NOT EXISTS (SELECT c’.cno   FROM course c’, prereq p
                  WHERE p.cno1 = c.cno AND p.cno2 = c’.cno AND c’.title = ‘DB’)
FOR XML PATH(‘course’), ROOT(‘db’)
```

FOR-XML supports neither recursive XML views, virtual nodes nor relation registers. It has essentially the same expressive power as PT_{nr}(FO, tuple, normal).

XSD specifies an XML view by annotating a (nonrecursive) schema, which associates elements and attributes with tables and table columns, respectively. Given a relational source, the annotated XSD constructs an XML tree by populating elements and attributes with tuples and values from their corresponding tables and columns, respectively. Information is passed via parent-child key-based joins. For example, the annotated XSD below generates a view that contains the list of all courses; under each course are its *cno*, *title* followed by a list of *prereq* elements, which consists of the *cno*'s of all immediate prerequisite courses:

```
<xsd:annotation>
  <xsd:appinfo>
    <sql:relationship name="prereq" parent="course" parent-key="course.cno"
                                     child="prereq" child-key="prereq.cno1"/>
  </xsd:appinfo>
<xsd:element name="course" sql:relation = "course">
    <xsd:complexType> <xsd:sequence>
      <xsd:element name="cno" sql:relation = "course.cno"> </xsd:element>
      <xsd:element name="title" sql:relation = "course.title"> </xsd:element>
      <xsd:element name="prereq" sql:relationship="prereq" maxOccurs="unbounded"/>
  </xsd:sequence> </xsd:complexType> </xsd:element>
</xsd:annotation>
```

All XSD views are nonrecursive and are expressible in PT_{nr}(CQ, tuple, normal).

DAD (Document Access Definition) of IBM DB2 XML Extender [18] supports SQL_MAPPING and RDB_MAPPING, which are similar to FOR-XML and XSD despite different syntax, and are contained in PT_{nr}(FO,tuple,normal) and PT_{nr}(CQ,tuple,normal), respectively. For example, the FOR-XML and XSD views given above can be expressed in SQL_MAPPING and RDB_MAPPING, respectively:

SQL_*mapping*:

```
<SQL_stmt> SELECT c.cno AS "cno", c.title AS "title"
           FROM course c
           WHERE NOT EXISTS (SELECT c'.cno  FROM course c', prereq p
                  WHERE p.cno1 = c.cno AND p.cno2 = c'.cno AND c'.title = 'DB')
</SQL_stmt>
<element_node name="course" multi_occurrence="yes">
  <element_node name="cno"> <text_node> <column name="cno"/></text_node>
  </element_node>              ... /*similarly for <element_node name="title">*/
</element_node>
```

RDB_*mapping*:

```
<element_node name="db">
  <rdb_node> <table name="course"/> <table name="prereq"/>
             <condition>course.cno=prereq.cno1</condition> </rdb_node>
  <element_node name="course" multi_occurrence="yes">
   ... /* <element_node name="cno"> and <element_node name="title">*/
   <element_node name="prereq" multi_occurrence="yes"> <text_node> <rdb_node>
      <table name="prereq"/><column name="cno2"/> </rdb_node> </text_node>
    </element_node> </element_node> </element_node>
```

TreeQL was proposed for publishing middleware SilkRoute [15]. Its abstraction [2] is precisely PT_{nr}(CQ, tuple, virtual): it defines an XML view by annotating the nodes of a tree template of a fixed depth with CQ queries, and supports virtual tree nodes and tuple-based information passing (tuple registers).

ATG (Attribute Transformation Grammar) was proposed in [3] and revised in [6], as the language of XML publishing middleware PRATA. An ATG defines an XML view based on a normalized DTD, by associating each element type a with an inherited attribute (register) $\$a$, and annotating its DTD production $a \to \alpha$ with a set of relational queries, one for each sub-element type b in the regular expression α, specifying how to compute $\$b$ and populate the b children of an a element. It supports recursive DTDs and thus recursive XML views, as well as virtual nodes to cope with XML entities. It provides a DTD-directed method to define XML views, such that the views are guaranteed to conform to a predefined DTD. For example, view τ_4 given earlier is defined in ATG as follows, which guarantees the view to conforms to DTD D_0 of Example 1:

```
db → course*
    $course = SELECT cno, title, type  FROM course
course → cno, title, type, prereq
    $prereq = SELECT cno   FROM $course;     similarly for $cno, $title and $type
type → regular | project
    $regular = SELECT cno   FROM $type   WHERE type='regular';
    $project = SELECT cno   FROM $type   WHERE type='project';
prereq → course*
    $course = SELECT c.cno, c.title, c.type   FROM prereq p, $prereq cp, course c
                WHERE cp.cno = p.cno1 AND p.cno2 = c.cno;
```

ATG of [3] is essentially PT(FO,relation,virtual) and can express views τ_1–τ_4.

XQuery [9] is a Turing-complete XML query language and can express arbitrary XML views. In practice one typically does not need the expressive power of XQuery and thus should not be penalized by the evaluation cost of full-fledged XQuery. In contrast to ATG, XQuery neither guarantees DTD conformance, nor provides any guidance on how to define an XML view that typechecks.

5 Dynamic Aspects

In many applications including mediation, archiving and Web site management, large XML documents may need to be *exported* from relational sources and *maintained*, rather than being "disposed". Just like their relational counterparts, there are two practical problems associated with XML views published from relational data: *the incremental update problem* and *the view update problem.*

Incremental publishing. Given a publishing transducer τ defined on a relational schema R, an instance I of R, the XML view $t = \tau(I)$, and changes ΔI to I, *incremental XML publishing* is to compute XML changes Δt to t such that $t \oplus \Delta t = \tau(I \oplus \Delta I)$. As an example, recall the XML view of Fig. 1(d) published from database I_0 by the transducer τ_4 of Example 5. When I_0 is updated by, *e.g.,* ΔI consisting of the insertion of tuples into *prereq* followed by deletion of tuples from *course*, certain *course* subtrees of Fig. 1(d) have to be changed. Incremental publishing is to compute the XML change Δt in response to I_0.

In contrast to recomputing the new view $\tau(I \oplus \Delta I)$ from scratch, incremental publishing can, in principle, improve performance substantially by applying only the changes Δt to the old view t. The need for this is evident in practice: as PRATA experienced with applications of European Bioinformatics Institute [10], recomputing the entire new view may be quite costly: it may take hours for large XML views. In contrast, relational changes ΔI are often small, and a small ΔI typically incurs only small XML changes Δt. Incremental XML publishing is to efficiently compute Δt by minimizing unnecessary recomputation.

One approach for incremental XML publishing is to push incremental computation to the underlying relational DBMS, along the same lines as implementations of XML publishing middleware [3, 15, 26]. However, several practical issues hamper the applicability of this *reduction* approach. For example, for recursive XML views the reduction approach depends on incremental update of materialized views defined using SQL'99 recursion. However, few DBMS's support SQL'99, and none supports incremental maintenance of SQL'99 views. In addition, if the queries embedded in a transducer are even mildly complex, the combined queries to be pushed down to DBMS may become extremely complex. They may not be effectively optimized by all DBMS, even for non-recursive publishing mappings.

In light of this, we outline an incremental publishing technique developed for PT(CQ,tuple,virtual) in [6]. It requires the lowest common denominator of DBMS functionality: neither SQL'99 nor incremental maintenance. The technique can be extended to other publishing transducers and languages mentioned earlier.

(1) *External storage.* For any instance I of schema R, a publishing transducer τ on R induces a function ST that, given a tag a, a state q and a value v of Reg_a, $\mathsf{ST}(a, q, v)$ returns a *unique* subtree in $t = \tau(I)$, rooted at a node tagged a and carrying v in its register. Leveraging this *subtree property*, we can store t using (i) a hash index H in which each entry $(a, q, \mathsf{id}(v), p)$ identifies a node in t, along with a pointer p to its subtree in S to be given below, where $\mathsf{id}(v)$ is the unique and compact representation of v, computed by a Skolem function; (ii) a subtree pool S consisting of entries $(a, q, \mathsf{id}(v), L)$, where L is a list of H entries to all the children of the node identified by $(a, q, \mathsf{id}(v))$. This allows us to store an XML view as a DAG, which may take exponentially less space than t.

(2) *Algorithm.* Given ΔI, one can compute XML updates Δt as follows.

(Step 1) Compute E^+ and E^-, the set of edges to be inserted into and deleted from t, respectively. Here each edge is identified by a pair of H-entries. This can be done as follows. (i) For each pair (q, a, q', b) of (state, tag) pairs, define an SQL query $Q_{(q,a,q',b)}$ as the union of queries ψ that appear in a τ rule $(q, a) \rightarrow \ldots (q', b, \psi)$. (ii) Compute the incremental version $\Delta Q_{(q,a,q',b)}$ of $Q_{(q,a,q',b)}$ in response to ΔI, by capitalizing on incremental techniques for SQL queries such as the counting method of [17]. (iii) Evaluate $\Delta Q_{(q,a,q',b)}$ for all involved tags (a, b) to find E^+ and E^-. Note that *neither* $Q_{(q,a,q',b)}$ *nor* $\Delta Q_{(q,a,q',b)}$ is recursive.

(Step 2) Update the hash index H and the subtree pool S with E^+ and E^-, by modifying the L filed of those relevant S entries.

(Step 3) For each newly inserted node in E^+, generate its subtree if its subtree does not have an entry in S. Only this phase involves recursive computation.

(Step 4) Clean up H and S by removing "dangling" entries, by a garbage collection procedure that runs in the background.

This algorithm avoids unnecessary recomputation by reusing subtrees in S at various levels of granularity. It guarantees that each distinct subtree of the new view is computed *at most once*.

View updates. As opposed to incremental publishing, *the view update problem* in connection with XML publishing can be stated as follows. Given a publishing transducer τ defined on a relational schema R, an instance I of R, the XML view $t = \tau(I)$, and XML updates Δt on t, it is to compute *relational updates* ΔI such that $t \oplus \Delta t = \tau(I \oplus \Delta I)$. That is, the relational changes ΔI, when propagated to XML via τ, yield the desired XML updates Δt on the view t. XML updates can be expressed in terms of XPath. For example, one may want to pose $U =$ *insert* CS240 *into course[cno='*CS650*']//course[cno='*CS450*']/prereq* on the XML view of Fig. 1(d); in response to this we want to find tuples ΔI_0 to insert into the underlying database I_0 such that $I_0 \oplus \Delta I_0$ yields the updated XML view.

The update problem is already hard for relational views. Indeed, given view updates, it is likely that there may not exist updates on the underlying source without introducing side effects, or there must exist multiple source updates (see, *e.g.,* [1]). Commercial relational DBMS's do not provide sophisticated view-update functionality. In particular, few complexity bounds are known even for relational view updates (see [8, 12] for recent work in this line of research).

When it comes to XML views published from relational data, the update problem is far more intriguing. In addition to the complications encountered in relational views, it introduces a number of new challenges. First, these XML views may be recursively defined, and are required to conform to a predefined schema. Second, XML updates may be recursive themselves (*e.g.,* descendant-or-self axis in XPath). Third, the semantics of XML view updates has to revised. For example, the XML update U given above attempts to add CS240 as a prerequisite of only those CS450 nodes below CS650. However, CS450 may appear elsewhere in the tree. The subtree property given above tells us that there is a *unique* CS450 subtree. Thus either all occurrences of CS450 should take CS240 as a prerequisite or none of them does. Commercial XML publishing products, *e.g.,* Microsoft SQL Server 2005 [21], provide at best extremely limited view update functionality.

Algorithm. An approach to tackling the view update problem has been developed in [11] for PT(CQ, tuple, virtual). It allows both XML views and updates to be recursively defined, and adopts a revised notion of side effects of XML view updates based on the subtree property. It compresses the XML view t into a DAG, using the same external storage as in the setting of incremental publishing, and derives a set V of relational views defined as edge queries $Q_{(q,a,q',b)}$ given above.

Note that V is a set of union of CQ queries. Given an XML update expression U_X posed on t, one can compute the relational updates ΔI as follows.

(Step 1) Validate $U_X(t)$ *w.r.t.* a predefined DTD of the XML view (if any), and reject U_X if $U_X(t)$ violates the DTD.

(Step 2) Translate U_X to an update expression U_V on the relational view V.

(Step 3) Translate U_V to an update expression U_R on the source I, if it exists.

(Step 4) Update the underlying database I using U_R and the relational views V using U_V, if U_R exists, otherwise report side effects to the user and reject U_X.

Step 3 may fail, *i.e.,* the XML view t is found *not updatable* by U_X, as for its relational counterparts. Furthermore, heuristic algorithms are necessary for Step 3 as the view update problem is intractable for the relational views V.

6 Concluding Remarks

The primary goal of the paper is to provide an overview of recent advances in XML publishing and a synergy between theory and practice. It is by no means a comprehensive survey: a number of related articles are not referenced due to the space constraint (see [20] for a recent survey). It is worth mentioning that XML publishing differs from recent work on data exchange (see [19] for a survey) in that XML publishing focuses on transformations from relations to XML defined in terms of mappings with embedded relational queries, rather than relation-to-relation or XML-to-XML mappings derived from source-to-target constraints.

Below we highlight some open research issues. One topic concerns XML integration: in contrast to XML publishing, it is to define a mapping from multiple distributed relational sources to XML documents. Along the same lines as XML publishing, the expressive power and complexity of XML integration languages deserve a full treatment. These are, however, more intriguing than their counterparts for XML publishing. In particular, to cope with dependencies on various sources, *two-way* transducers are required in contrast to *top-down* XML publishing transducers, as indicated by the language proposed in [4] for XML integration. In addition, other issues that arise in practice, such as information preservation required by data migration [7], make the study more complicated.

Another issue is about XML shredding, *i.e.,* for storing XML data in relations. A publishing transducer τ can be treated as a relational query [13]: fixing a designated output tag a_o, we can define the output of $\tau(I)$ on a database I as a *relation*: the union of all $\mathsf{Reg}_{a_o}(v)$ for all nodes v labeled a_o in the tree. Similarly one can define an *XML shredding automaton* that has XML queries embedded in it, operates on *existing XML trees* and returns *a set of tuples* to add to a database. In contrast to publishing transducers, an XML shredding automaton outputs a *relation* instead of an XML tree. Based on this an XML shredding language has been given in [14]. XML shredding automata can be used to characterize the expressive power and complexity of XML shredding languages found in practice.

Much more needs to be done for the view update problem associated with XML views published from relational data. To our knowledge no previous work has considered view update management for publishing transducers defined with relational queries beyond CQ, *e.g.,* PT(FO, relation, virtual).

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. *TOCL*, 4, 2003.
3. M. Benedikt, C. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-directed publishing with attribute translation grammars. In *VLDB*, 2002.
4. M. Benedikt, C. Y. Chan, W. Fan, J. Freine, and R. Rastogi. Capturing both type and integrity constraints in data integration. In *SIGMOD*, 2003.
5. M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, 2005.
6. P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, 2004.
7. P. Bohannon, W. Fan, M. Flaster, and P. Narayan. Information preserving XML schema embedding. In *VLDB*, 2005.
8. P. Buneman, S. Khanna, and W. Tan. On propagation of deletions and annotations through views. In *PODS*, 2002.
9. D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001. `http://www.w3.org/TR/xquery`.
10. B. Choi, W. Fan, X. Jia, and A. Kasprzyk. A uniform system for publishing and maintaining XML data. In *VLDB*, 2004. Demo.
11. B. Choi, C. Gao, W. Fan, and S. Viglas. Updating recursive XML views. In *ICDE*, 2007.
12. G. Cong, W. Fan, and F. Geerts. Annotation propagation revisited for key preserving views. In *CIKM*, 2006.
13. W. Fan, F. Geerts, and F. Neven. Expressiveness and complexity of XML publishing transducers. In *PODS*, 2007.
14. W. Fan and L. Ma. Selectively storing XML data in relations. In *DEXA*, 2006.
15. M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. SilkRoute: A framework for publishing relational data in XML. *TODS*, 27(4):438–493, 2002.
16. F. Gécseg and M. Steinby. Tree languages. In *Handbook of Formal Languages*, volume 3. Springer, 1996.
17. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.
18. IBM. DB2 XML Extender. *www-3.ibm.com/software/data/db2/extended/xmlext/*.
19. P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, 2005.
20. R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Xsym*, 2003.
21. Microsoft. XML support in microsoft SQL server 2005, 2005. *msdn.microsoft.com/library/en-us/dnsql90/html/sql2k5xml.asp/*.
22. F. Neven. On the power of walking for querying tree-structured data. In *PODS*, 2002.
23. F. Neven and T. Schwentick. Query automata over finite trees. *TCS*, 275(1-2):633–674, 2002.
24. Oracle. Oracle Database 10g Release 2 XML DB Whitepaper. *http://www.oracle.com/technology/tech/xml/xmldb/index.html*.
25. Y. Papakonstantinou and V. Vianu. Type inference for views of semistructured data. In *PODS*, 2000.
26. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. L. H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB J.*, 10(2-3):133–154, 2001.