Expressiveness and Complexity of XML Publishing Transducers

Wenfei Fan
Univ. of Edinburgh &
Bell Labs
wenfei@inf.ed.ac.uk

Floris Geerts
Univ. of Edinburgh &
Hasselt University &
Transnational Univ. of Limburg
fgeerts@inf.ed.ac.uk

Frank Neven
Hasselt University &
Transnational Univ. of Limburg
frank.neven@uhasselt.be

Abstract

A number of languages have been developed for specifying XML publishing, *i.e.,* transformations of relational data into XML trees. These languages generally describe the behaviors of a middleware controller that builds an output tree iteratively, issuing queries to a relational source and expanding the tree with the query results at each step. To study the complexity and expressive power of XML publishing languages, this paper proposes a notion of *publishing transducers*. Unlike automata for querying XML data, a publishing transducer generates a new XML tree rather than performing a query on an existing tree. We study a variety of publishing transducers based on what relational queries a transducer can issue, what temporary stores a transducer can use during tree generation, and whether or not some tree nodes are allowed to be virtual, *i.e.,* excluded from the output tree. We first show how existing XML publishing languages can be characterized by such transducers. We then study the membership, emptiness and equivalence problems for various classes of transducers and existing publishing languages. We establish lower and upper bounds, all matching except one, ranging from PTIME to undecidable. Finally, we investigate the expressive power of these transducers and existing languages. We show that when treated as relational query languages, different classes of transducers capture either complexity classes (*e.g.,* PSPACE) or fragments of datalog (*e.g.,* linear datalog). For tree generation, we establish connections between publishing transducers and logical transductions.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages – *Query Languages*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic — *Computational Logic*

General Terms: Languages, Theory, Design.

1. Introduction

To exchange data residing in relational databases, one typically needs to export the data as XML documents. This is referred to as *XML publishing* in the literature [2, 5, 11, 16, 26], and is essentially to define an XML view for relational data: given a relational schema R, it is to define a mapping τ such that for any instance I of R, $\tau(I)$ is an XML tree.

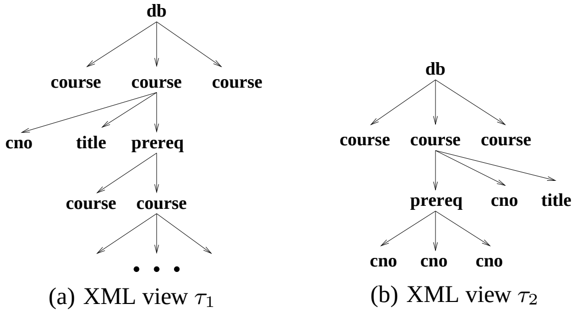

Figure 1: Example XML publishing

A number of languages have been developed for XML publishing, including commercial products such as annotated XSD of Microsoft SQL Server 2005 [19], DAD of IBM DB2 XML Extender [15], DBMS_XMLGEN of Oracle 10g XML DB [23], and research prototypes XPERANTO [26], TreeQL [11, 2] and ATG [5, 6]. These languages typically specify 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 source, generates the children of the node using the query results, and iteratively expands the subtrees of those children in the same way. It may (implicitly) store intermediate query results in registers and pass the information downward to control subtree generation [2, 5, 6, 15, 19, 23, 26]. It may also allow *virtual* tree nodes [2, 5, 6] that will be removed from the output tree to express, *e.g.,* XML entities.

Given a variety of XML publishing languages, a user may naturally ask which language should be used to define an XML view. Is the view expressible in one language but not in another? How expensive is it to compute views defined in a language? Furthermore, after the view is defined, is it possible to determine, at compile time, whether or not the view makes sense, *i.e.,* it does not always yield an empty tree? Is this view equivalent to another view, *i.e.,* they always produce the same output tree from the same relational source?

Example 1.1: Consider a *registrar* database I_0 of a relational schema R_0 consisting of *course(cno, title, dept)*, and *prereq(cno1, cno2)* (with keys underlined). The database maintains *course* data and a relation *prereq*, in which a tuple (c_1, c_2) indicates that c_2 is a prerequisite of c_1. That is, relation *prereq* gives the prerequisite hierarchy of the courses.

The registrar office wants to export two XML views:

- XML view τ_1 contains the list of all the CS *course*s extracted from the database I_0. Under each *course* are the *cno* (num-

PODS'07, June 11-13, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-685-1/07/0006...$5.00.

ber) and *title* of the course, as well as its prerequisite hierarchy. As shown in Fig. 1(a), the depth of the *course* sub-tree is determined by its prerequisite hierarchy.

- View τ_2 is a tree of depth three. As depicted in Fig. 1(b), it consists of the list of all the CS *course*s. Below each *course* c is the list of all the *cno*'s that appear in the prerequisite *hierarchy* of c, followed by the *cno* and *title* of c.

The user may ask the questions mentioned above regarding these XML views. As will be seen shortly, not all commercial languages are capable of expressing these views due to the recursive nature of the prerequisite hierarchy. □

Answering these questions calls for a full treatment of the expressive power and complexity of XML publishing languages. The increasing demand for data exchange and XML publishing highlights the need for this study. Indeed, this is not only important for the users by providing a guidance for how to choose a publishing language, but is also useful for database vendors in developing the next-generation XML publishing languages. Despite their importance, to our knowledge no previous work has investigated these issues.

Publishing transducers. To examine the complexity and expressiveness of XML publishing languages in a comparative basis, we need a uniform formalism to characterize these languages. To this end, we introduce a formalism of transducers, referred to as *publishing transducers*. A publishing transducer is a top-down transducer that simultaneously issues queries to a relational database, keeps intermediate results in its local stores (registers) associated with each node, and iteratively expands XML trees by using the extracted data. As opposed to the automata for querying XML data [21, 22], it generates a new XML tree rather than evaluating a query on an existing tree. In order to encompass existing publishing languages, we parameterize publishing transducers using the following parameters:

- $\mathcal{L}$ (**logic**): the relational query language in which queries on relational data are expressed; we consider conjunctive queries with '$=$' and '$\neq$' (CQ), first-order queries (FO), and (inflationary) fixpoint queries (FP);
- S (**store**): registers that keep intermediate results; we consider transducers in which each register stores a finite *relation* versus those that store a single *tuple*;
- O (**output**): the types of tree nodes; in addition to *normal* nodes that remain in the output tree, we may allow *virtual* nodes that will be removed from the output. We study transducers that only produce normal nodes versus those that may also allow virtual nodes.

We denote by PT($\mathcal{L}, S, O$) various classes of publishing transducers, where $\mathcal{L}, S, O$ are logic, store and output parameters as specified above. As we will see later, different combinations of these parameters yield a spectrum of transducers with quite different expressive power and complexity.

Main results. We present a comprehensive picture of the complexity and expressiveness for all classes PT($\mathcal{L}, S, O$) as well as for existing XML publishing languages.

Characterization of existing XML publishing languages. We examine several commercial languages and research proposals, and show that each of these languages can be characterized as a special case of publishing transducers. For example, annotated XSD of Microsoft [19] is a class of "nonrecursive" PT(CQ, tuple, normal), DBMS_XMLGEN of Oracle [23] can be expressed in PT(FP, tuple, normal), and SQL/XML of IBM [15] is a class of nonrecursive PT(FO, tuple, normal). Moreover, relation stores and virtual nodes are needed to characterize TreeQL [11, 2] and ATG [5, 6]. Conversely, for most classes PT($\mathcal{L}, S, O$) there are existing publishing languages corresponding to them. For the few that do not find a corresponding commercial system, we explain why it is the case. For example, no commercial language corresponds to PT(FP, relation, virtual) because it does not increase the expressive power over PT(FO, relation, virtual), and for the latter a running prototype system [5] has already been being used.

Static analysis. We investigate classical decision problems associated with transducers: the membership, emptiness and equivalence problems. 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 from a publishing transducer (membership), and whether a more efficient publishing transducer can in fact generate the same set of XML trees as a more expensive transducer (equivalence). We establish complexity bounds for these problems, ranging from PTIME to undecidable, for all the classes PT($\mathcal{L}, S, O$) and for the special cases that characterize existing publishing languages. All these upper and lower bounds match, except one with a Π_3^p upper bound and a Π_2^p lower bound. We also provide data complexity for *evaluating* various publishing transducers.

Expressive power. We characterize the expressiveness of publishing transducers in terms of both relational query languages and logical transducers for tree generation.

We first treat a publishing transducer as a relational query that, on an input relational database, evaluates to a relation which is the union of the registers associated to nodes of the output tree with a designated label. We show that each class PT($\mathcal{L}, S, O$) captures either a complexity class or a fragment of a well-studied relational query language, except one for which we only show that it contains a fragment of datalog. For example, the largest class PT(FP, relation, virtual) captures PSPACE and the smallest PT(CQ, tuple, normal) captures linear datalog (see, *e.g.*, [14]). Along the same lines we characterize the existing publishing languages. For example, we show that SQL/XML of IBM [15] is in FO and annotated XSD of Microsoft [19] is in union of CQ queries.

For tree generation, we establish connections between certain fragments of PT($\mathcal{L}, S, O$) and logical interpretations [12] or transductions [8]. For example, we show that PT($\mathcal{L}$, tuple, virtual) contain the $\mathcal{L}$-transducers for $\mathcal{L}$ ranging over CQ, FO and FP, and that regular unranked tree languages are contained in PT(FO, tuple, normal) but not in PT(CQ, relation, virtual).

In both settings we also provide separation and equivalence results for various classes of publishing transducers. For example, we show that PT(FP, relation, normal) and PT(FO, relation, normal) are equivalent in the relational setting, whereas for tree generation, PT(FO, relation, normal) is properly contained in PT(FP, relation, normal) but in contrast, PT(FO, relation, virtual) and PT(FP, relation, virtual) have the same expressive power.

To our knowledge, this work is the first to provide a general theoretical framework to study the expressive power and complexity of XML publishing languages. A variety of techniques are used to prove the results, including finite model constructions and a wide range of simulations and reductions.

Related work. As remarked earlier, a number of XML publishing languages have been proposed (see [16] for a survey). However, the complexity and expressiveness of these languages have not been studied. There has also been recent work on data exchange, *e.g.*, [3, 10]. This work differs from [3, 10] in that we focus on (a) transformations from relational data to XML defined in terms of transducers with embedded relational queries, rather than relation-to-relation [10] or XML-to-XML [3] mappings derived from source-to-target constraints, and (b) complexity and expressiveness analyses instead of consistent query answering.

A variety of tree automata and transducers have been developed (see [13] for a survey), some particularly for XML (*e.g.,* [18, 20, 21, 22]). As remarked earlier, tree recognizers [13] and the automata for querying XML [21, 22] operate on an existing tree, and either accept the tree or select a set of nodes from the tree. In contrast, a publishing transducer does not take a tree as input; instead, it builds a new tree by extracting data from a relational source. While the k-pebble transducers of [20] return an XML tree as output, they also operate on an input XML tree rather than a relational database, and cannot handle data values. Similarly, an XSM of [18] takes XML data streams as input and produces one or more XML streams. Furthermore, the expressive power and complexity of these XML transducers have not been studied.

There has been a host of work on the expressive power and complexity of relational query languages (see [1, 9] for surveys). While those results are not directly applicable to publishing transducers, some of our results are proved by capitalizing on related results on relational query languages.

Logical interpretations or transductions define a mapping from structures to structures through a collection of formulas (see *e.g.,* [8] for a survey of graph transductions). Recently logical tree-to-tree interpretations are used in [4] to characterize XQuery. We employ transductions to characterize the tree generating power of publishing transducers.

Organization. Section 2 reviews XML trees. Section 3 defines publishing transducers. Section 4 characterizes existing XML publishing languages in terms of these transducers. Section 5 studies decision problems for a variety of publishing transducers and existing languages, and Section 6 investigates their expressive power. Section 7 summarizes the main results of the paper.

2. XML Trees with Local Storage

We first review XML trees and define trees with registers.

XML trees. An XML document is typically modeled as a node-labeled tree. Assume a finite alphabet Σ of *tags*. A *tree domain dom* is a subset of $\mathbb{N}^*$ such that for any $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, if $v.i$ is in *dom* then so is v, and in addition, if $i > 1$ then $v.(i-1)$ is also in *dom*. A Σ*-tree* t is defined to be $(dom(t), lab)$, where $dom(t)$ is a tree domain, and lab is a function from $dom(t)$ to Σ.

Intuitively, $dom(t)$ is the set of the *nodes* in t, while the empty string ε represents the root of t, denoted by *root*(t). Each node $v \in dom(t)$ is labeled by the function lab with a tag a of Σ, called an a*-element*. Moreover, v has a (possibly empty) list of elements as its children, denoted by *children*(v). Here $v.i \in dom(t)$ is the i-th child of v, and v is called the parent of $v.i$. Note that t is *unranked*, *i.e.,* there is no fixed bound on the number of children of a node in t.

In particular we assume that Σ contains a special *root tag* r, such that $lab(\varepsilon) = r$ and moreover, for any $v \in dom(t)$, $lab(v) \neq r$ if $v \neq \varepsilon$. To simplify the discussion we also assume a special tag, *text*, in Σ. Any node labeled *text* carries a string (PCDATA) and is referred to as a *text* node.

Trees with local storage. We study Σ-trees generated from relational data. To construct a tree in a context-dependent fashion, one needs to pass information from a node to its children. To do this, we store data values in a local store at each node. We assume a recursively enumerable infinite domain $\mathbf{D}$ of *data values* which serves both as the domain of the relational databases and of the local stores at nodes of the generated output tree.

A Σ*-tree with local storage*, or simply a tree if it is clear from the context, is a pair (t, Reg), where t is a Σ-tree, and *Reg* is a function that associates each node $v \in dom(t)$ with a finite relation over $\mathbf{D}$. We refer to *Reg*(v) as the *local store* or the *register* of v, and use Tree$_\Sigma$ to denote the set of all Σ-trees with local storage.

We consider two classes of trees: for all $v \in dom(t)$, (a) either *Reg*(v) stores a *finite relation* over $\mathbf{D}$, (b) or *Reg*(v) is a *single tuple* over $\mathbf{D}$. These are referred to as Σ-trees with *relation* stores and *tuple* stores, respectively. Note that trees with tuple stores are a *special case* of trees with relation stores. As will be seen shortly, the content of *Reg*(v) is computed via a relational query on a database over $\mathbf{D}$, and it is used to control how the children of v will be generated.

3. Publishing Transducers

We now define publishing transducers. Intuitively, a publishing transducer is a finite-state machine that creates a tree from a relational database in a *top-down way*. It starts from an initial state that corresponds to the root node of the tree, and then follows *deterministically* a transition based on the current state of the transducer and the tag of the current node in the tree created so far. The transition directs how the children of a node are generated based on the underlying database, by providing the tags of the children as well as relational queries that extract data from the database. More specifically, for each child tag a, a relational query of the form $\phi(\bar{x};\bar{y})$ is specified, which generates a list of children labeled a. The result of the query is partitioned using the *group-by* attributes $\bar{x}$, yielding sets of tuples. For each set, a child labeled a is spawned, carrying the set in its local register, which will be used in queries in successive transitions. As a result, the structure of the tree is dependent on the underlying database instance.

We next define publishing transducers more formally. In the following, a relational schema is a finite collection of relation names and associated arities.

Definition 3.1: Let R be a relational schema and $\mathcal{L}$ a relational query language. A *publishing transducer* for R is defined to be $\tau = (Q, \Sigma, \Theta, q_0, \delta)$, where Q is a finite set of *states*; Σ is a finite alphabet of *tags*; Θ is a function from Σ to $\mathbb{N}$ associating the *arity* of registers *Reg*$_a$ to each Σ tag a; q_0 is the *start state*; and δ is a finite set of *transduction rules* such that for each $(q, a) \in Q \times \Sigma$, if $q \neq q_0$ and a is not the root tag r, then there is a unique rule of the form:

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

Here $k \geq 0$, and for $i \in [1, k]$, $(q_i, a_i) \in Q \times \Sigma$, and $\phi_i \in \mathcal{L}$ is a query from R and *Reg*$_a$ to *Reg*$_{a_i}$, where *Reg*$_a$ and *Reg*$_{a_i}$ are a $\Theta(a)$- and a $\Theta(a_i)$-ary relation, respectively. To simplify the discussion we assume that $a_i \neq a_j$ if $i \neq j$. As mentioned above, tuples in the result of $\phi_i(\bar{x}_i;\bar{y}_i)$ are grouped by $\bar{x}_i$ and are distributed among different children labeled a_i as the content of *Reg*$_{a_i}$ of each child. This will be explained in more detail below.

There are two special cases: (a) q_0 and r do not appear in the right-hand side of any rule, and there is exactly one rule for q_0, namely, the rule for (q_0, r), referred to as the *start rule*; (b) if a is *text*, then $k = 0$ in the rule for $(q, \textit{text})$, *i.e.,* the right-hand side of the rule is empty. □

Example 3.1: The view shown in Fig. 1(a) can be defined by a publishing transducer $\tau_1 = (Q_1, \Sigma_1, \Theta_1, q_0, \delta_1)$, where $Q_1 = \{q_0, q\}$, Σ_1 = {*db*, *course*, *prereq*, *cno*, *title*, *text*}, and the root tag is *db*; we associate four sets of registers Reg_c, Reg_p, $Reg_\#$ and Reg_t with *course*, *prereq*, *cno* and *title* nodes, to which the arity-function Θ_1 assigns $2, 1, 1, 1$, respectively; finally, δ_1 is defined as follows:

$$\begin{array}{l}
\delta_1(q_0, db) \quad = (q, course,\ \phi_1(cno, title; \emptyset)),\ \text{where} \\
\quad \phi_1(cno, title) = \exists\ dept\ (course(cno, title, dept) \wedge dept = \text{‘}CS\text{’})
\end{array}$$

$$\begin{array}{l}
\delta_1(q, course) \ = (q, cno,\ \phi_2^1(cno; \emptyset)),\ \ (q, title,\ \phi_2^2(title; \emptyset)), \\
\qquad\qquad\qquad (q, prereq,\ \phi_2^1(cno; \emptyset)),\ \text{where} \\
\quad \phi_2^1(cno) = \exists\ title\ Reg_c(cno, title),\ \text{and} \\
\quad \phi_2^2(title) = \exists\ cno\ Reg_c(cno, title),
\end{array}$$

$$\begin{array}{l}
\delta_1(q, prereq) \ = (q, course,\ \phi_3(cno, title; \emptyset)),\ \text{where} \\
\quad \phi_3(c, t) = \exists\ c'\ d\ (Reg_p(c') \wedge prereq(c', c) \wedge course(c, t, d))
\end{array}$$

$$\delta_1(q, cno) = (q, text,\ \phi_4(cno; \emptyset)),\ \text{where}\ \phi_4(c) = Reg_\#(c)$$

$$\delta_1(q, title) = (q, text,\ \phi_5(title; \emptyset)),\ \text{where}\ \phi_5(t) = Reg_\#(t)$$

Note that in each query $\phi(\bar{x}; \bar{y})$ in the rules, $|\bar{y}| = 0$, *i.e.*, $\bar{y}$ is $\emptyset$. The semantics of τ_1 will be given in Example 3.2. □

A publishing transducer τ can be recursive. To illustrate this we define the *dependency graph* G_τ of τ. 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.

Transformations. In a nutshell, τ generates a tree from a database I of schema R in a top-down fashion. Initially, τ constructs a tree t consisting of a single node labeled (q_0, r) with an empty storage. 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 embedded in the rule to the relational database I and the register $Reg_a(u)$ associated with u, and spawning the children of u based on the query results. The query results are kept in the registers of these children nodes. The transformation proceeds until a stop condition is satisfied at all the leaf nodes (to be presented shortly). At the end, all registers and states are removed from the tree t to obtain a Σ-tree, which is the output of τ.

We now formally define the transformation induced by τ from a database I. As in [2], we assume an implicit ordering $\leq$ on $\mathbf{D}$, which is just used to order the nodes in the output tree and, hence, get a unique output. We do *not* assume that the ordering is available to the query language $\mathcal{L}$.

We extend Σ-trees with local storage by allowing nodes to be labeled with symbols from $\Sigma \cup Q \times \Sigma$. We use $\text{Tree}_{Q \times \Sigma}$ to denote the set of all such extended Σ-trees. Then, every step in the transformation rewrites a tree in $\text{Tree}_{Q \times \Sigma}$, starting with the single-node tree (q_0, r).

More specifically, for two trees $\xi, \xi' \in \text{Tree}_{Q \times \Sigma}$, we define the step-relation $\Rightarrow_{\tau,I}$ as follows: $\xi \Rightarrow_{\tau,I} \xi'$ iff there is a leaf u of ξ labeled (q, a) and one of the following conditions holds:

(1) if there is an ancestor v of u such that u, v are labeled with *the same state and tag*, and $Reg_a(v) = Reg_a(u)$, then ξ' is obtained from ξ by changing $lab(u)$ to a. Otherwise,

(2) assume that the rule for (q, a) is

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

If $k > 0$, then ξ' is obtained from ξ by rooting the forest $f_1 \cdots f_k$ under u. For each $j \in [1, k]$, f_j is constructed as follows. Let $\{\bar{d}_1, \ldots, \bar{d}_n\} = \{\bar{d} \mid I \cup Reg_a(u) \models \exists \bar{y}_j \phi_j(\bar{d}; \bar{y}_j)\}$ and $\bar{d}_1 \leq \cdots \leq \bar{d}_n$ with $\leq$ extended to tuples in the canonical way. Then f_j is a list of nodes $[v_1, \cdots, v_n]$, where v_i is labeled with (q_j, a_j) and its register $Reg_{a_j}(v_i)$ stores the relation $\{\bar{d}_i\} \times \{\bar{e} \mid I \cup Reg_a(u) \models \phi_j(\bar{d}_i; \bar{e})\}$; here we use Reg_a and Reg_{a_j} to denote the registers associated with the a-node u and the a_j-node v_i, respectively. If all f_i's are empty, ξ' is obtained from ξ by labeling u with a.

If $k = 0$, *i.e.*, the right-hand side of the rule is empty, then ξ' is obtained from ξ by changing the label of u to a. In particular, if the tag a is *text*, then in ξ', u carries a string representation of $Reg_a(u)$ (assuming a function that maps relations over $\mathbf{D}$ to strings, based on the order $\leq$).

The first condition, referred to as the *stop-condition*, states that the transformation stops at the leaf u if there is a node v on the path from the root to u such that u *repeats* the state q, tag a, and the content of $Reg_a(v)$ of v. Since the subtree rooted at u is uniquely determined by $q, a, Reg_a(u)$ and I, this asserts that the tree will not expand at u if the expansion *does not add new information* to the tree. This stop condition is the same as the one used in ATGs [6]. As will be seen in the next section, most commercial systems support only *nonrecurisve* publishing transducers and thus do not necessarily need a stop condition.

The second condition states how to generate the children of the leaf u via a transduction rule. Observe that the children spawned from u can be characterized by a regular expression $a_1^* \ldots a_k^*$. For each $j \in [1, k]$, the a_j children are *grouped by* the values $\bar{d}$ of the parameter $\bar{x}$ in the query $\exists \bar{y}_j \phi_j(\bar{x}_j; \bar{y}_j)$. That is, for each distinct $\bar{d}$ such that $\exists \bar{y}_j \phi_j(\bar{d}; \bar{y}_j)$ is nonempty, an a_j child w is spawned from u, carrying the result of $\phi_j(\bar{d}; \bar{y}_j)$ in its local store $Reg_{a_j}(w)$.

The transformation *stops* at the leaf u, *i.e.*, no children are spawned at u, if (a) the stop condition given above is satisfied; or (b) the query $\phi_j(\bar{x}_j; \bar{y}_j)$ turns out to be empty for all $i \in [1, k]$ when it is evaluated on I and $Reg_a(u)$; in this case all the forests f_j are empty; or (c) the right-hand side of the rule for (q, a) is empty, *i.e.*, $k = 0$ in condition (2) above; this is particularly the case for $a = text$, as text nodes have no children. These conditions ensure the termination of the computation. Note that transduction at other leaf nodes may proceed after the transformation stops at u.

Example 3.2: Given an instance I_0 of the schema R_0 described in Example 1.1, the publishing transducer τ_1 given in Example 3.1 works as follows. It first generates the root of the tree t, labeled with $(q_0,\ db)$. It then evaluates the query ϕ_1 on I_0, and for each distinct tuple in the result, it spawns a *course* child v carrying the tuple in its register $Reg_c(v)$. At node v it issues queries ϕ_2^1 and ϕ_2^2 on $Reg_c(x)$, and spawns its *cno*, *title* and *prereq* children carrying the corresponding tuple in their registers. At the *cno* child, it simply extracts the string value of *cno* and the transformation stops; similarly for *title*. At the *prereq* child u, it issues query ϕ_3 against both I_0 and $Reg_p(u)$; *i.e.*, it extracts all (immediate) prerequisites of the course v, for which the cno is stored in $Reg_p(u)$. In other words, the *cno* information passed down from node v is used to determine the children of u. For each distinct tuple in the result of ϕ_3, it generates a course child of u. The transformation continues until either it reaches some course for which there is no prerequisite, *i.e.*, ϕ_3 returns empty at its *prereq* child; or when a course requires itself as a prerequisite (which does not happen in practice), and at this point the stop condition terminates the transformation. The final tree, after the local registers and states are stripped from it, is a Σ-tree of the form depicted in Fig. 1(a).

Note that the transformation is *data-driven*: the number of children of a node and the depth of the XML tree are determined by the relational database I. □

We denote by $\Rightarrow_{\tau,I}^*$ the reflexive and transitive closure of $\Rightarrow_{\tau,I}$. The *result* of the τ-*transformation* on I *w.r.t.* $\leq$ is the tree ξ such that $(q_0, r) \Rightarrow^* \xi$ and all leaf nodes of ξ carry a label from Σ. This means that ξ is final and cannot be expanded anymore. We use $\tau(I)$ to denote the Σ-tree obtained from ξ by striking out the local storage and states from ξ. We denote by $\tau(R)$ the set $\{\tau(I) \mid I \text{ is an instance of } R\}$, *i.e.,* the set of trees induced by τ-transformations on I when I ranges over all instances of the relational schema R. Note that for any order on the input instance, a transducer always terminates and produces a unique output tree.

Virtual nodes. To cope with XML entities we also consider a class of publishing transducers with *virtual nodes*. Such a transducer is of the form $\tau = (Q, \Sigma, \Theta, q_0, \delta, \Sigma_e)$, where Σ_e is a designated subset of Σ, referred to as the *virtual tags* of τ; and $Q, \Sigma, \Theta, q_0, \delta$ are the same as described in Definition 3.1. We require that Σ_e does not contain the root tag. On a relational database I the transducer τ behaves the same as a normal transducer, except that the Σ-tree $\tau(I)$ is obtained from the result ξ of the τ-transformation on I as follows. First, the local registers and states are removed from ξ. Second, for each node v in $dom(\xi)$, if v is labeled with a tag in Σ_e, we *shortcut* v by replacing v with *children*(v), *i.e.,* treating *children*(v) 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 3.3: Suppose that we want to define a publishing transducer for the XML view shown in Fig. 1(b), and that the query language $\mathcal{L}$ is FO. One can show, via a simple argument using Ehrenfeucht-Fraïssé (EF)-style game, that this is not expressible as a normal transducer of Definition 3.1 (see, *e.g.,* [17] for a discussion of EF games). In contrast, this can be defined as a publishing transducer τ_2 with virtual nodes. Indeed, capitalizing on a virtual tag l, we give some of the transduction rules δ_2 of τ_2 as follows:

$\delta_2(q_0, db)$ and $\delta_2(q, course)$ are as in Example 3.1

$\delta_2(q, prereq) = (q, l,\ \varphi_1(\emptyset; cno)),\ (q, cno,\ \varphi_2(cno; \emptyset))$
$\quad\varphi_1(c) = Reg_p(c) \vee \exists\, c'\ (Reg_p(c') \wedge prereq(c', c))$
$\quad\varphi_2(c) = \varphi_1(c) \wedge \forall c' (Reg_p(c') \leftrightarrow \varphi_1(c'))$,

$\delta_2(q, l)$ is as $\delta_2(q, prereq)$ with $precreq = l$ and $Reg_p = Reg_l$.

In φ_1, $|\bar{x}| = 0$ and thus the result of φ_1 is put in a *single relation*, stored in the register $Reg_l(v)$ of the l child v. In contrast, $|\bar{y}| = 0$ in φ_2 and thus its query result is *grouped* by each distinct tuple. Hence, if the query result is nonempty, then for each tuple in it, a distinct *cno* child is generated.

Intuitively, for each course c the transducer τ_2 recursively finds *cno*'s in the prerequisite hierarchy of c and adds these *cno*'s to the relation $Reg_l(v)$ until it reaches a fixpoint, where v is labeled with the virtual tag l. Only at this point, the query $\varphi_2(c)$ returns a nonempty set $Reg_l(v)$. For each *cno* in the set, a distinct *cno* node is created. Then, all the nodes labeled l are removed and those *cno* nodes become the children of c. Thus τ_2 induces the XML view of Fig. 1(b). □

Fragments. We denote by 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. We consider $\mathcal{L}$ ranging over conjunctive queries with '$\neq$' (CQ), first-order logic (FO) and (inflationary) fixpoint logic (FP), all with equality '$=$'. *Store S* is either *relation* or *tuple*, indicating that the Σ-trees induced by the transducers are with relation or tuple stores, respectively. Observe that transducers with tuple stores are a special case of those with relation stores. For any transducer τ with *tuple* stores, $|\bar{y}_i| = 0$ in each query $\phi_i(\bar{x}_i; \bar{y}_i)$ in τ, as illustrated in Example 3.1. *Output O* is either *normal* or *virtual*, indicating whether a transducers allow virtual nodes or not. Thus PT(FP, relation, virtual) is the largest class considered in this paper, which consists of transducers that are defined with fixpoint-logic queries and generate trees with relation stores and virtual nodes. In contrast, PT(CQ, tuple, normal) is the smallest.

For each class PT$(\mathcal{L}, S, O)$, we denote by PT$_{nr}(\mathcal{L}, S, O)$ its subclass consisting of all *nonrecursive* transducers in it.

For instance, the transducers τ_1 and τ_2 given in Examples 3.1 an 3.3 are in PT(CQ, tuple, normal) and PT(FO, relation, virtual), respectively (τ_2 is also definable in PT$_{nr}$(FP, tuple, normal); we omit this definition for the lack of space).

4. Characterization of XML Publishing Languages

We examine publishing languages that are either supported by commercial products or are representative research proposals (see [16] for a survey). We classify these languages in terms of publishing transducers with certain restrictions.

Microsoft SQL Server 2005 [19]. Two main XML publishing methods are supported by Microsoft, namely, FOR-XML expressions and annotated XSD schema.

The first method extracts data from a relational source via SQL queries, and organizes the extracted data into XML elements using a FOR-XML construct. Hierarchical XML trees can be built top-down by nested FOR-XML expressions. While no explicit registers are used, during tree generation information can be passed from a node to its children along the same lines as the use of tuple variables in nested SQL queries (*i.e.,* correlation). The depth of a generated tree is bounded by the nesting level of FOR-XML expressions (although user-defined functions can be recursive, Microsoft imposes a maximum recursive depth, and thus a bounded tree depth). No virtual nodes are allowed. Thus FOR-XML expressions are definable in PT$_{nr}$(FO, tuple, normal).

The second method specifies an XML view by annotating a (nonrecursive) XSD schema, which associates elements and attributes with relations and table columns, respectively. Given a relational source, the annotated XSD constructs an XML tree by populating elements with tuples from their corresponding tables, and instantiating attributes with values from the corresponding columns. Information is passed via parent-child key-based joins, specified in terms of a relationship annotation. It only supports simple condition tests and does not allow virtual nodes. The depth of the tree is bounded by the fixed "tree template" (XSD). Thus annotated XSD can be expressed in PT$_{nr}$(CQ, tuple, normal).

IBM DB2 XML Extender [15]. IBM also supports two main methods: SQL/XML and document access definition (DAD).

The first method extends SQL by incorporating XML constructs (*e.g.,* XMLAGG, XMLELEMENT). It extracts relational data in parallel with XML-element creation. Nested queries are used to generate a hierarchical XML tree, during which a node can pass information to its children via correlation. The tree has a fixed depth bounded by the level of query nesting, and has no virtual nodes. Thus SQL/XML is essentially PT$_{nr}$(FO, tuple, normal).

The second method in turn has two flavors, namely, SQL_MAPPING and RDB_MAPPING. The former extracts relational data with a single SQL query, and organizes the extracted tuples into a hierarchical XML tree by using a sequence of group_by, one for each tuple column and following a fixed order on the columns. The depth of the tree is bounded by the arity of the tuples returned by the query. The latter embeds nested RDB_NODE expressions in a DAD. The DAD is basically a tree template with a fixed depth, and those embedded expressions are essentially CQ queries for populating elements and attributes specified in the DAD. Neither of these two allows virtual nodes. One can express DAD with SQL_MAPPING in PT$_{nr}$(FO, tuple, normal), and RDB_MAPPING in PT$_{nr}$(CQ, tuple, normal).

Microsoft SQL Server 2005		IBM DB2 XML Extender		Oracle 10g XML DB		XPERANTO	TreeQL	ATG
FOR XML	annotated XSD	SQL/XML	DAD (SQL/RDB)	SQL/XML	DBMS_XMLGEN			
PT$_{nr}$(FO, t, n)	PT$_{nr}$(CQ, t, n)	PT$_{nr}$(FO, t, n)	PT$_{nr}$(FO, t, n) (SQL) PT$_{nr}$(CQ, t, n) (RDB)	PT$_{nr}$(FO, t, n)	PT(FP, t, n)	PT$_{nr}$(FO, t, n)	PT$_{nr}$(CQ, t, v)	PT(FO, t, v) [5] PT(CQ, r, v) [6]

Table 1: Characterization of existing XML publishing languages (t: tuple; r: relation; n: normal; v: virtual)

Oracle 10g XML DB [23]. Oracle supports SQL/XML as described above, and a PL/SQL package DBMS_XMLGEN. DBMS_XMLGEN extends SQL/XML by supporting the linear recursion construct *connect-by* (SQL'99), and is thus capable of defining recursive XML views. Given a relational source, an XML tree of an unbounded depth is generated top-down, along the same lines as nested SQL/XML queries. Information is passed from a node to its children via *connect-by* joins. For each tuple resulted from the joins, a child node is created, whose children are in turn created in the next iteration of the recursive computation. Neither virtual nodes are allowed, nor an explicit stop condition is given. If the stop condition given in Section 3 is imposed, XML views defined in DBMS_XMLGEN are expressible in PT(FP, tuple, normal).

XPERANTO [26]. It supports essentially the same XML views as SQL/XML, and thus in PT$_{nr}$(FO, tuple, normal).

TreeQL [11, 2]. TreeQL was proposed for the XML publishing middleware SilkRoute [11]. Here we consider its abstraction developed in [2]. It defines an XML view by annotating the nodes of a tree template (of a fixed depth) with CQ queries. It supports virtual tree nodes and tuple-based information passing via free-variable binding (*i.e.,* the free variables of the query for a node v are a subset of the free variables of each query for a child of v). Thus TreeQL views are expressible in PT$_{nr}$(CQ, tuple, virtual).

ATG [5, 6]. Attribute transformation grammars (ATG) were proposed in [5] and revised in [6], for XML publishing middleware PRATA. An ATG defines an XML view based on a DTD, by associating each element type with an inherited attribute (register), and annotating each production $a \rightarrow \alpha$ in the DTD with a set of relational queries, one for each sub-element type b in the regular expression α, specifying how to populate the b sub-elements of an a element. It supports recursive DTDs and thus recursive XML views, as well as virtual nodes to cope with XML entities. While the early version of [5] employs FO queries and tuple registers, the revised ATGs [6] adopt CQ queries, relation registers and the stop condition of Section 3. ATGs of [5, 6] are basically PT(FO, tuple, virtual) and PT(CQ,relation,virtual), respectively.

The characterization is summarized in Table 1. Except DBMS_XMLGEN and ATGs, these languages do not support recursive XML views exported from relational data. Indeed, one can verify, via a simple EF-game argument, that the XML views of Example 3.1 and 3.3 are expressible in DBMS_XMLGEN and ATGs, but not in the other languages.

5. Decision Problems and Complexity

In this section we first provide tight worst-case complexity for evaluating various publishing transducers. We then focus on central decision problems associated with these transducers. Consider a class PT($\mathcal{L}$, S, O) of publishing transducers. (*i*) The *membership problem* for PT($\mathcal{L}$, S, O) is to determine, given a Σ-tree t and a transducer τ in this class, whether there is an instance I with $t = \tau(I)$, *i.e.,* τ on I computes the tree t. (*ii*) The *emptiness problem* for PT($\mathcal{L}$, S, O) is to determine, given τ in this class, whether there is an instance I with $\tau(I) \neq r$, *i.e.,* the tree with the root only. So, it is to decide whether τ can induce nontrivial trees. (*iii*) The *equivalence problem* for PT($\mathcal{L}$, S, O) is to determine, given two transducers τ_1 and τ_2 in the class defined for relational databases of *the same schema* R, whether or not $\tau_1(I) = \tau_2(I)$ for all instances I of R, *i.e.,* the two transducers produce the same Σ-trees on all the instances of R.

We first establish upper and lower bounds for these problems, all matching except one, for all classes of transducers defined in Section 3. We then revisit these issues for nonrecursive transducers that characterize the existing publishing languages studied in Section 4. Our main conclusion for this section is that most of these problems are beyond reach in practice for general publishing transducers, but some problems become simpler for certain existing languages.

5.1 Decision Problems for Publishing Transducers

We first discuss the data complexity of computing the output of a publishing transducer.

Proposition 5.1: For any τ in PT($\mathcal{L}$, S, O), where $\mathcal{L}$ is CQ, FO or FP, and O is normal or virtual, and for any database I, the size of $\tau(I)$ is at most exponential and double exponential in the size of I when S is tuple and relation, respectively. There are instances for which this maximal size is reached when $\mathcal{L}$ is CQ. Worst-case data-complexity is EXPTIME and 2EXPTIME when S is tuple and relational, respectively. □

PROOF. It suffices to remark that the rank of $\tau(I)$ is bounded by a polynomial in the size $|I|$ of I, its depth by a polynomial in $|I|$ if S is tuple, and by an exponential if S is relation. To see that the bounds are tight, for transducers with tuple stores consider a database I_1 encoding a DAG of a certain shape (*e.g.,* a chain of diamonds), and a recursive transducer τ_1 in PT(CQ, tuple, normal) expanding the DAG into a tree. Then the size of the output $\tau_1(I_1)$ is exponential in $|I_1|$. For relation stores, consider I_2 encoding a n-digit binary counter, and τ_2 in PT(CQ, relation, normal) that at each node creates two branches, each incrementing the counter by 1. Then the size of $\tau_2(I_2)$ is 2^{2^n}. □

We now turn to the classical decision problems associated with transducers.

Proposition 5.2: Membership, emptiness and equivalence are undecidable for PT($\mathcal{L}, S, O$) when $\mathcal{L}$ is FO or FP, S is relation or tuple, and O is virtual or normal. □

PROOF. It suffices to show that these problems are undecidable for PT(FO, tuple, normal). This is verified by a reduction from the satisfiability problem for relational FO queries, which is known to be undecidable (see, *e.g.,* [1]). □

For $\mathcal{L}$ equal to CQ, the situation gets slightly better.

Theorem 5.3: For PT(CQ, S, O),

- the emptiness problem is decidable in PTIME for PT(CQ, S, normal), but becomes NP-complete for PT(CQ, S, virtual);
- the equivalence problem is undecidable;
- the membership problem is Σ_2^p-complete for PT(CQ, tuple, normal), but becomes undecidable when either S is relation or O is virtual. □

PROOF. For the emptiness problem for τ in PT(CQ, S, normal), it is sufficient to test emptiness of the CQ queries in the start rule of τ. The satisfiability of these queries can be checked in PTIME in the size of the queries. The NP lower bound for the emptiness problem for PT(CQ, tuple, virtual) is by a reduction from 3SAT [24]. The upper bound for the emptiness problem for τ in PT(CQ, relation, virtual) is proved by providing an NP algorithm that (1) guesses a path from the root of the dependency graph G_τ of τ to a node labelled with a non-virtual tag; (2) checks the satisfiability of the composition of the CQ queries along that path. The latter can be checked in PTIME in the size of the original CQ queries.

For PT(CQ, tuple, normal) the undecidability of the equivalence problem is by a reduction from the halting problem for 2-register machines (see, *e.g.,* [7]) which leads to the undecidability of the problem for PT(CQ, S, O). We note that it remains undecidable for PT(CQ, relation, O) without '$\neq$'.

The Σ_2^p lower bound for the membership problem for PT(CQ, tuple, normal) *in the absence of* '$\neq$', is by a reduction from $\exists^*\forall^*$-3SAT [24]. The upper bound is proved by (1) establishing a small model property: for any Σ-tree t and τ in the class, if $t \in \tau(R)$, then there exists an I such that $\tau(I) = t$ and $|I|$ is linear in $|t|$; (2) providing an algorithm for checking the existence of I by using a nondeterministic PTIME Turing machine with a NP oracle. For PT(CQ, tuple, virtual), the undecidability is by a reduction from the emptiness problem for deterministic finite 2-head automata (see, *e.g.,* [27]). For PT(CQ, relation, O), the undecidability is by a reduction from the satisfiability problem for FO queries: given a FO query q on databases of schema R_1, we define a new schema R_2 that subsumes R_1 to encode the result of each sub-query of q, and a transducer τ in the class that checks whether q on an instance I of R_1 yields the result coded in the corresponding instance of R_2. Capitalizing on virtual nodes, we show that $\tau(R_2)$ contains a fixed tree iff q is not satisfiable. The proof does not make use of '$\neq$'. □

5.2 Complexity of Existing Publishing Languages

The results of the previous section carry over immediately to the existing publishing languages adopting recursion, which are PT(FP, tuple, normal) (DBMS_XMLGEN), PT(FO, tuple, virtual) (ATG [5]) and PT(CQ, relation, virtual) (ATG [6]). Table 1 shows that, in contrast, many of them are non-recursive: PT$_{nr}$(FO, tuple, normal) (FOR-XML, SQL_mapping, SQL/XML), PT$_{nr}$(CQ, tuple, normal) (annotated XSD, RDB_mapping), and PT$_{nr}$(CQ, tuple, virtual) (TreeQL). Each of these nonrecursive classes is treated below.

We show that the absence of recursion for these publishing languages simplifies the analyses. Indeed, for any τ in one of these nonrecursive classes, the Σ-tree induced by τ on any database is bounded by τ. From this it follows:

Corollary 5.4: For publishing transducers τ in PT$_{nr}$(FO, tuple, normal) (or PT$_{nr}$(CQ, tuple,O)), the worst-case data complexity for τ-transformations is in PTIME (both for O normal or virtual). □

The decision problems also become simpler, to an extent.

Theorem 5.5: The emptiness, membership and equivalence problems are undecidable for PT$_{nr}$(FO, tuple, normal). The emptiness problem for PT$_{nr}$(CQ, tuple, normal) is in PTIME; it is NP-complete for PT$_{nr}$(CQ, tuple, virtual). The membership and equivalence problems for PT$_{nr}$(CQ, tuple, O) are Σ_2^p-complete, and in Π_3^p and Π_2^p-hard, respectively. □

PROOF. The proof of Proposition 5.2 remains intact for PT$_{nr}$(FO, tuple, normal). Similarly, the PTIME upper bound for emptiness of Theorem 5.3 trivially holds for PT$_{nr}$(CQ, tuple, normal). Since the NP lower bound proof of Theorem 5.3 for emptiness uses a non-recursive transducer, the lower bound extends to PT$_{nr}$(CQ, tuple, virtual). The NP upper bound of Theorem 5.3 trivially holds for PT$_{nr}$(CQ, tuple, virtual). Similarly, for PT$_{nr}$(CQ, tuple, O), the Σ_2^p upper-bound proof of Theorem 5.3 for membership extends to PT$_{nr}$(CQ, tuple, virtual). For equivalence for PT$_{nr}$(CQ, tuple, normal), we prove the Π_2^p lower bound by reduction from the equivalence problem for CQ queries with '$\neq$' [28]. We give a Π_3^p-time checking algorithm for PT$_{nr}$(CQ, tuple, virtual), by characterizing transducer equivalence in terms of (a) isomorphism between the dependency graphs of transducers (DAGs), and (b) a form of equivalence on CQ queries along the paths in the two DAGs starting from the root. □

6. Expressiveness of Publishing Transducers

In this section, we characterize the expressive power of publishing transducers in terms of relations-to-tree mappings (i.e., tree generation) and relations-to-relation mappings (i.e, relational query languages).

6.1 Tree Generation versus Relational Languages

Although publishing transducers define mappings from relational databases to trees, they can also be considered as a relational query language mapping relational databases to relations. To this end, we fix a designated output label a_o. For any instance I of R, the τ-transformation on I yields a final tree ξ with local storage in Tree$_{Q\times\Sigma}$, from which the output Σ-tree $\tau(I)$ is obtained by removing local stores and transducer states (recall from Section 3). The *relation* induced by τ on I is then defined to be the union of all the stores $Reg_{a_o}(v)$ for all nodes v labeled a_o in ξ. Therefore, we refer to τ as a relational query when τ is viewed as a mapping from instances I to the relation induced by τ on I. When τ is viewed as a relation-to-tree mapping, we refer to τ as a tree generating mapping.

We want to compare the expressive power of one class PT($\mathcal{L}_1, S_1, O_1$) with that of another class PT($\mathcal{L}_2, S_2, O_2$) both as a tree generation and a relational query language. We say that PT($\mathcal{L}_1, S_1, O_1$) is *contained in* PT($\mathcal{L}_2, S_2, O_2$) as a tree/relational query language, 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 they define the same tree/relational query.

The two classes are said to be *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($\mathcal{L}_1, S_1, O_1$). We say that PT($\mathcal{L}_1, S_1, O_1$) is *properly contained in* PT($\mathcal{L}_2, S_2, O_2$), denoted by PT($\mathcal{L}_1, S_1, O_1$) $\subset$ 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($\mathcal{L}_2, S_2, O_2$). These notions extend to comparing PT($\mathcal{L}, S, O$) vs. other tree generating formalisms, and to comparing PT($\mathcal{L}, S, O$) vs. relational query languages.

We also characterize PT($\mathcal{L}, S, O$) with respect to complexity classes. Treating PT($\mathcal{L}, S, O$) as a relational query language, for example, we consider the *recognition problem* for its transducers τ: given a tuple u and an instance I of the schema for which τ is defined, it is to determine whether u is in the relation $R_o(a_o)$ induced by τ on I. We say that PT($\mathcal{L}$, S, O) *captures* a complexity class $\mathcal{C}$ if the recognition problem for all transducers in PT($\mathcal{L}, S, O$) is in $\mathcal{C}$ and moreover, for any query q whose recognition problem is in $\mathcal{C}$, there exists τ in PT($\mathcal{L}, S, O$) defined on the same schema R as q, such that q and τ return the same output relation on all instances of R.

Outline. We study the expressive power of all the classes PT($\mathcal{L}, S, O$) defined in Section 3 with respect to relational query and tree generation languages, in Sections 6.2 and 6.3, respectively.

We then investigate the expressive power of existing XML publishing languages in Section 6.4. The results in this section hold irrespectively of whether the queries in $\mathcal{L}$ have explicit access to the order $\leq$ on the domain $\mathbf{D}$, unless explicitly stated otherwise.

6.2 Expressiveness in Terms of Relational Queries

We start by treating PT($\mathcal{L}, S, O$) as a relational query language. We first review two fragments of datalog. One fragment is *linear datalog* (see *e.g.,* [1]), denoted by LINDATALOG. It consists of datalog programs in which each rule is of form: $p(\bar{x}) \leftarrow p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)$, and moreover, at most one p_i is an IDB predicate. We allow some p_j to be $\neq$. The other, referred to as *deterministic datalog* and denoted by DDATALOG, is the class of programs in which each IDB predicate has only one rule of the form above (its body may contain more than one IDB predicate).

We use $\mathsf{TC}_0[\mathcal{L}]$ to denote a fragment of transitive closure logic: the set of all formulas $[\mathsf{TC}_{\bar{x},\bar{y}}\ \varphi](\bar{a}, \bar{b})$, where $\varphi \in \mathcal{L}$. Following [14] one can verify that TC_0[CQ] = LINDATALOG.

The main result of Section 6.2 is given as follows.

Theorem 6.1: When treated as relational query languages,

(1) PT($\mathcal{L}$, S, virtual) $=$ PT($\mathcal{L}$, S, normal),

(2) PT(CQ, tuple, O) $\subset$ PT(FO, tuple, O)
(3) $\subseteq$ PT(FP, tuple, O)
(4) $\subset$ PT(FO, relation,O)
(5) $=$ PT(FP, relation, O),

(6) PT(CQ, tuple, O) $\subset$ PT(CQ, relation, O)
(7) $\subset$ PT(FO, relation, O),

(8) PT(CQ, relation, O) $\not\subseteq$ PT(FO, tuple, O),

where O is either normal or virtual. The containment in statement (3) is proper if NLOGSPACE $\neq$ PTIME. Moreover,

(a) PT(FO, relation, O) captures PSPACE.

(b) PT(FP, tuple, O) $=$ (inflationary) FP on ordered databases and thus captures PTIME.

(c) PT(FO, tuple, O) $=$ TC_0[FO] on ordered databases, and thus captures NLOGSPACE. On unordered databases, PT(FO, tuple, O) $\subset$ NLOGSPACE.

(d) PT(CQ, relation, O) $\supseteq$ DDATALOG.

(e) PT(CQ, tuple, O) $=$ LINDATALOG. □

Among other things, this tells us the following in the relational setting. Virtual nodes do not add expressive power (statement (1)) and thus we only need to consider PT($\mathcal{L}$, S, normal). In contrast, we have to treat publishing transducers with relation stores and those with tuple stores separately (4, 6, 8). While FP does not add expressive power over FO in PT($\mathcal{L}$, relation, O), it does in PT($\mathcal{L}$, tuple, O) (5, 3). Moreover, replacing CQ with FO in PT(CQ, S, O) leads to increase in expressiveness when S is either relation or tuple (2, 7). The rest of the results position the expressive power of these transducers *w.r.t.* complexity classes and datalog fragments.

PROOF. To show that PT(FO, relation, O) captures PSPACE, we first show that for each τ in PT(FO, relation, O), its recognition problem can be determined by using nondeterministic PSPACE Turing machine. Conversely, we simulate every partial fixpoint query (known to capture PSPACE on ordered instances) using a transducer and show that a total order is definable in this class. Similarly, we simulate each τ in PT(FP, tuple, O) (resp. PT(FO, tuple, O)) in FP (resp. in TC_0[FO]), and vice versa. Thus on ordered databases, PT(FP, tuple, O) and PT(FO, tuple, O) capture PTIME and NLOGSPACE, respectively. On unordered structures, since the parity query *even* is not expressible in FP, it is not definable in PT(FP, tuple, O). We simulate DDATALOG and LINDATALOG in PT(CQ, relation, O) and PT(CQ, tuple, O), respectively, and vice versa for LINDATALOG.

Statement (1) holds because for any tree ξ induced by a transducer in PT($\mathcal{L}, S$, virtual), and for any normal a-element v in ξ, removing virtual nodes from ξ does not change the content of the register $Reg_a(v)$. Statement (5) is verified by simulating FP queries in PT(FO, relation, normal). Statements (4, 6) follow from (5) and the fact that each transducer in PT($\mathcal{L}$, tuple, O) is a special case of PT($\mathcal{L}$, relation, O) in which for any query $\phi(\bar{x}, \bar{y})$, $\bar{y}$ is the empty list. The containment in (4) is proper since on unordered structures, *even* is expressible in PT(FO, relation, O) but not in FP. We show that the containment of (6) is proper by defining a transducer τ in PT(CQ, relation, O) that takes a relation encoding the edges of a rooted graph G as input, expands G into a tree, and adds a certain node to the tree iff the root of G has two particular descendants on different branches of G. One can verify that τ is not expressible even in PT(FO, tuple, O) (this requires an EF-game argument to show that the relational query defined by τ is not definable in FO). From this also follows (8). To prove that the containments in (2, 7) are proper, we give an FO query q, which is clearly definable in PT(FO, tuple, O), and show that q is not definable in PT(CQ, relation, O) due to the monotonicity of CQ queries. □

6.3 Tree Generating Power

For tree generation, we provide separation and equivalence results for various classes of publishing transducers, and establish their connection with logical transducers [8] and regular tree languages (specialized DTDs).

Equivalence and separation. As opposed to Theorem 6.1, Proposition 6.2 below shows that when it comes to tree generation, virtual nodes do add expressive power to publishing transducers. Moreover, if $\mathcal{L} \subset \mathcal{L}'$, then PT($\mathcal{L}'$, S, normal) properly contains PT($\mathcal{L}$, S, normal) whereas in the relational query setting, PT(FP, relation, normal) = PT(FO, relation, normal). The other results in Proposition 6.2 are comparable to their counterparts in Theorem 6.1.

Proposition 6.2: For tree generation,

(1) PT($\mathcal{L}$, S, normal) $\subset$ PT($\mathcal{L}$, S, virtual),

(2) PT($\mathcal{L}$, S, normal) $\subset$ PT($\mathcal{L}'$, S, normal) if $\mathcal{L} \subset \mathcal{L}'$,

(3) PT(CQ, tuple, virtual) $\subset$ PT(FO, tuple, virtual)
(4) $\subseteq$ PT(FP, tuple, virtual)

(5) PT(CQ, relation, virtual) $\subset$ PT(FO, relation, virtual)
(6) $=$ PT(FP, relation, virtual),

(7) PT($\mathcal{L}$, tuple, O) $\subset$ PT($\mathcal{L}$, relation, O).

(8) PT(CQ, relation, normal) $\not\subseteq$ PT(FP, tuple, virtual),

where $\mathcal{L}, \mathcal{L}_1, \mathcal{L}_2$ are FP, FO or CQ, and O is normal or virtual. The containment in (4) is proper if PTIME $\neq$NLOGSPACE. □

PROOF. We prove that the containments in (1-5) are proper as follows. For (1), we define τ_1 in PT(CQ, tuple, virtual) that can generate a Σ-tree in which the root has exponentially many children, which is not doable even by transducers in PT(FP, relation, normal). For (2-5), observe that by Theorem 6.1, there exists a Boolean query q in $\mathcal{L}'$ not expressible in PT($\mathcal{L}, S, O$) if PT($\mathcal{L}, S, O$) is considered as a relational query language (for (4) if PTIME $\neq$NLOGSPACE). We define τ_2 in PT($\mathcal{L}', S$, normal) such that τ_2 generates a nontrivial tree iff q is satisfied. Statement (6) holds since each FP query can be simulated in PT(FO, relation, virtual) by using virtual nodes. The containment of (7) is proper since transducers in PT($\mathcal{L}$, relation, O) can induce trees of exponential depth, as opposed to trees of polynomial depth induced by those in PT($\mathcal{L}$, tuple, O); similarly for (8). □

Logical transducers. For a logic $\mathcal{L}$, an $\mathcal{L}$-tree-transduction defines a mapping from relations over a schema R to a tree with a sequence of $\mathcal{L}$-formulas ϕ_e, $\phi_<$ and $(\phi_a)_{a\in\Sigma}$ such that on every R-structure I, $\phi_e(I)$, $\phi_<(I)$ and $\phi_a(I)$ define the edge relation, the ordering on the siblings, and the a-labeled nodes of the tree, respectively. To express transformations of exponential size increase (like publishing transducers can), $\phi_e(I)$ defines a DAG, and we consider its unfolding as a tree when making a comparison with publishing transducers. First-order (resp. second-order) transductions are those where nodes of the output tree are k-ary tuples (resp. k-ary relations) over the input structure, for some fixed k. An $\mathcal{L}$-transduction T is *fixed-depth* when there is an ℓ such that for any input I, $T(I)$ is a tree of depth at most ℓ. In a similar way to logical transductions, we can also define $\mathcal{C}$-transductions (both first and second order) for a complexity class $\mathcal{C}$ where there are $\mathcal{C}$-Turing machines to decide the relations ϕ_e, $\phi_<$ and $(\phi_a)_{a\in\Sigma}$.

Theorem 6.3:

1. When $\mathcal{L}$ ranges over CQ, FO and FP, every $\mathcal{L}$-transduction is definable in PT($\mathcal{L}$, tuple, virtual).
2. When $\mathcal{L}$ ranges over FO and FP, every transducer in PT$_{nr}$($\mathcal{L}$, tuple, virtual) is definable as a fixed-depth $\mathcal{L}$-transduction.
3. There is a recursive transducer in PT(FO, tuple, O) that is not definable as an FO-transduction.
4. When $\mathcal{L}$ ranges over CQ, FO and FP, over unordered trees, fixed-depth $\mathcal{L}$-transductions are equivalent to PT$_{nr}$($\mathcal{L}$, tuple, O).
5. Over ordered input structures, PT(FO, relation, virtual) and PT(FP, tuple, virtual) contain the PSPACE second-order and PTIME first-order transductions. □

PROOF. (1) A direct simulation using virtual nodes to express arbitrary sequences of labels shows that PT($\mathcal{L}$, tuple, virtual) are at least as expressive as $\mathcal{L}$-transductions. (2) When transducers are nonrecursive, there is no stop condition, and PT($\mathcal{L}$, tuple, virtual) corresponds precisely to $\mathcal{L}$-transductions generating trees of a fixed depth. (3) This statement holds because recursive PT(FO, tuple, O) can express graph-reachability known not definable in FO. (4) When disregarding the order of siblings, virtual nodes are no longer needed and fixed depth FO-transductions become equivalent to PT$_{nr}$(FO, tuple, O). (5) PT(FO, relation, virtual) and PT(FP, tuple, virtual) are quite expressive as they contain all transformations in PSPACE and PTIME, respectively, over ordered structures. Here the correspondence between FP and PTIME, and between partial fixpoint and PSPACE on ordered structures, is exploited. □

Regular tree languages. It is known [22] 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]. Recall that 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. A Σ-tree t conforms to d iff for each a-element v in t, the list of labels of *children*(v) is a string in α.

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 Σ-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 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 $L(d) = \tau(R)$.

The next result 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 contrast, PT(CQ, S, O) does not have sufficient expressive power to define even DTDs. We defer a full treatment of the connection between publishing transducers (*e.g.*, PT(FP, S, normal) and PT(FO, S, normal)) and regular tree languages to the full version of the paper due to the space constraint.

Theorem 6.4: When $\mathcal{L}$ is FO or FP, every specialized DTD over Σ is definable in PT($\mathcal{L}$, tuple, virtual). There exist DTDs that are not definable in PT(CQ, relation, virtual). □

PROOF. For each specialized DTD d, we define τ in PT(FO, tuple, virtual) for a schema R encoding a graph such that all trees in $\tau(R)$ conform to d, and for any $t \in L(d)$, there is an instance I of R such that $t = \tau(I)$. We show that DTDs with disjunctive rules are not definable in PT(CQ, relation, virtual) due to the monotonicity of CQ queries. □

6.4 Expressiveness of Existing Languages

We next study the expressiveness of existing publishing languages in the relational-query and tree generation settings.

Relational Query Languages. It can be verified that the results of Theorem 6.1 for PT(FP, tuple, normal), PT(FO, tuple, virtual) and PT(CQ, relation, virtual) also hold for DBMS_XMLGEN and ATGs, respectively. The theorem below settles the issue for PT$_{nr}$(FO, tuple, normal) (FOR-XML, SQL_mapping, XPERANTO, SQL/XML), PT$_{nr}$(CQ, tuple, O) (annotated XSD, RDB_mapping, TreeQL).

Theorem 6.5: When treated as relational query languages, PT$_{nr}$(FO, tuple, O) $=$ FO, and PT$_{nr}$(CQ, tuple, O) $=$ UCQ (UCQ denotes union of conjunctive queries with '$=, \neq$'). □

PROOF. Every UCQ query can be simulated in PT$_{nr}$(CQ, tuple, O). Conversely, for each transducer τ in PT$_{nr}$(CQ, tuple, O) and a designated output tag a_o, the output relation $R_o(a_o)$ of a τ-transformation is computed by the union of all *path queries*, where each path query is the compositions of the CQ queries on a path in the dependency graph of τ from the root to a leaf node labeled a_o. Similarly, PT$_{nr}$(FO, tuple, O) $=$ FO can be verified. □

Tree generation. The proof for (1, 2) of Proposition 6.2 remains intact for nonrecursive transducers. As an immediate corollary, PT$_{nr}$(CQ, tuple, normal) $\subset$ PT$_{nr}$(FO, tuple, normal) and PT$_{nr}$(CQ, tuple, normal) $\subset$ PT$_{nr}$(CQ, tuple, virtual). Theorem 6.3 has shown that for unordered trees fixed-depth FO-transductions are equivalent to PT$_{nr}$(FO, tuple, O).

Publishing languages characterized by nonrecursive publishing transducers do not have sufficient expressive power to define DTDs, due to the bound on the depth of the trees induced. From Theorem 6.4 it follows that specialized DTDs are definable in ATGs of [5].

7. Conclusion

We have proposed the notion of publishing transducers and characterized several existing XML publishing languages in terms of these transducers. For a variety of classes of publishing transducers, including both generic PT($\mathcal{L}, S, O$) and nonrecursive PT$_{nr}$($\mathcal{L}, S, O$) characterizing existing publishing languages, we

Fragments	Equivalence	Emptiness	Membership
PT(FP, S, O) (Th. 5.2)	undecidable	undecidable	undecidable
PT(FO, S, O) (Th. 5.2)	undecidable	undecidable	undecidable
PT(CQ, tp, nm) (Th. 5.3)	undecidable	PTIME	Σ_2^p-complete
PT(CQ, rl, nm) (Th. 5.3)	undecidable	PTIME	undecidable
PT(CQ, S, vr) (Th. 5.3)	undecidable	NP-complete	undecidable
PT$_{nr}$(FO, tp, nm) (Th. 5.5)	undecidable	undecidable	undecidable
PT$_{nr}$(CQ, tp, nm) (Th. 5.5)	Π_3^p/Π_2^p	PTIME	Σ_2^p-complete
PT$_{nr}$(CQ, tp, vr) (Th. 5.5)	Π_3^p/Π_2^p	NP-complete	Σ_2^p-complete

Table 2: Complexity of decision problems (S: relation or tuple; O: normal or virtual; tp: tuple; rl: relation; nm: normal; vr: virtual)

Fragments	Complexity/Language
PT(FP, rl, O) (Th. 6.1)	PSPACE
PT(FO, rl, O) (Th. 6.1)	PSPACE
PT(FP, tp, O) (Th. 6.1)	FP, PTIME (ordered database)
PT(FO, tp, O) (Th. 6.1)	TC_0[FO], NLOGSPACE (ordered)
PT(CQ, rl, O) (Th. 6.1)	$\supseteq$ DDATALOG
PT(CQ, tp, O) (Th. 6.1)	TC_0[CQ], LINDATALOG
PT$_{nr}$(FO, tp, O) (Th. 6.5)	FO
PT$_{nr}$(CQ, tp, O) (Th. 6.5)	UCQ

Table 3: Expressive power characterized in terms of relational query languages

have provided (a) a complete picture of the membership, equivalence and emptiness problems, (b) a comprehensive expressiveness analysis in terms of both querying and tree generating power, as well as a number of separation and equivalence results. We expect these results will help the users decide what publishing languages to use, and database vendors develop or improve commercial XML publishing languages.

The main results for the static analyses and querying power (for relational queries only due to lack of space) are summarized in Tables 2 and 3, respectively, annotated with their corresponding theorems and conditions (*e.g.,* ordered). These tables show that different combinations of logic $\mathcal{L}$, store S and output O, as well as the presence of recursion, lead to a spectrum of publishing transducers with quite different complexity and expressive power.

The study of publishing transducers is still preliminary. An open issue is the exact complexity for the equivalence problem for PT$_{nr}$(CQ, tuple, O), the only problem for which we do not have matching upper and lower bounds. Another open question concerns, when treated as a relational query language, whether or not PT(CQ, relation, O) captures DDATALOG? We only know that DDATALOG is contained PT(CQ, relation, O). A third interesting topic is the typechecking problem for publishing transducers. Our preliminary results (not included due to lack of space) show that while this is undecidable in general, there are interesting decidable cases. This issue deserves a full treatment of its own. Finally, we plan to investigate two-way and nondeterministic publishing transducers.

Acknowledgments. We thank Michael Benedikt, Christoph Koch and Leonid Libkin for helpful discussions. Wenfei Fan is supported in part by EPSRC GR/S63205/01, GR/T27433/01 and BBSRC BB/D006473/1. Floris Geerts is a postdoctoral researcher of the FWO Vlaanderen and is supported in part by EPSRC GR/S63205/01.

8. 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. Arenas and L. Libkin. XML data exchange: consistency and query answering. In *PODS*, 2005.

[4] M. Benedikt and C. Koch. Interpreting tree-to-tree queries. In *ICALP*, pages 552–564, 2006.

[5] M. Benedikt, C. Chan, W. Fan, R. Rastogi, S. Zheng and A. Zhou. DTD-directed publishing with attribute translation grammars. In *VLDB*, 2002.

[6] P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, 2004.

[7] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1997.

[8] B. Courcelle. Monadic second-order definable graph transductions: A survey. *TCS*, 126(1):53–75, 1994.

[9] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv*, 33(3):374–425, 2001.

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

[11] M. F. 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.

[12] J. Flum and H. Ebbinghaus. *Finite Model Theory*. Springer, 2nd edition, 1999.

[13] F. Gécseg and M. Steinby. Tree languages. In *Handbook of Formal Languages*, volume 3. Springer, 1996.

[14] E. Grädel. On Transitive Closure Logic. In *CSL*, 1992.

[15] IBM. DB2 XML Extender. *http://www-3.ibm.com/software/data/db2/extended/xmlext/.*

[16] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Xsym*, 2003.

[17] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[18] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In *VLDB*, 2002.

[19] Microsoft. XML support in microsoft SQL server 2005, 2005. *msdn.microsoft.com/library/en - us/dnsql90/html/sql2k5xml.asp/.*

[20] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *JCSS*, 66(1):66–97, 2003.

[21] F. Neven. On the power of walking for querying tree-structured data. In *PODS*, 2002.

[22] F. Neven and T. Schwentick. Query automata over finite trees. *TCS*, 275(1-2):633–674, 2002.

[23] Oracle. Oracle Database 10g Release 2 XML DB Whitepaper. *http://www.oracle.com/technology/tech/xml/ xmldb/index.html.*

[24] C. H. Papadimitriou. *Computational Complexity*. AW, 1994.

[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. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB J.*, 10(2-3):133–154, 2001.

[27] M. Spielmann. *Abstract State Machines: Verification Problems and Complexity*. PhD thesis, RWTH Aachen, 2000.

[28] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. *JCSS*, 54(1), 1997.

[29] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.