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 publish-
ning transducer generates a new XML tree rather than performing a
query on an existing tree. We study a variety of publishing trans-
ducers 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., ex-
cluded 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 prob-
lems for various classes of transducers and existing publishing lan-
guages. We establish lower and upper bounds, all matching, rang-
ing from PTIME to undecidability. Finally, we investigate the expres-
sive 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 Man-
agement]: Languages — Query Languages; F.4.1 [Mathematical
Logic and Formal Languages]: Mathematical Logic — Compu-
tational 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, τ(I)
is an XML tree.

A number of languages have been developed for XML publish-
ing, 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 proto-
types XPERANTO [20], TreeQL [11][2] and ATG [5][9]. 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 it-
ervatively 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 natu-
rially ask which language should be used to define an XML view. Is
the view expressive in one language but not in another? How ex-
ensive 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 al-
ways produce the same output tree from the same relational source?

Example 1.1: Consider a registrar database I₀ of a rela-
tional schema I₀ 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 (c₁, c₂) indicates
that c₂ is a prerequisite of c₁. That is, relation prereq gives the
prerequisite hierarchy of the courses.

The registrar office wants to export two XML views:
XML view $\tau_1$ contains the list of all the CS courses extracted from the database $I_0$. Under each course are the cno (number) 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 $\tau_2$ is a tree of depth three. As depicted in Fig. 1(b), it consists of the list of all the CS courses. Below each course $c$, 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:

- $L$ (logic): the relational query language in which queries on relational data are expressed; we consider conjunctive queries with ‘=’ and ‘£’ (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 $PT(L, S, O)$ various classes of publishing transducers, where $L, S, O$ 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 $PT(L, S, O)$ 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 $PT(L, S, O)$ 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 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 $PT(L, S, O)$ and for the special cases that characterize existing publishing languages. All these upper and lower bounds match.

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 $PT(L, S, O)$ 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., [13]). 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 $PT(L, S, O)$ and logical interpretations of transductions [8]. For example, we show that $PT(CQ, tuple, virtual)$ contains the $L$-transducers for $L$ 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 [10] for a survey). However,
Section 7 summarizes the main results of the paper. Publishing languages in terms of these transducers. Section 5 studies data streams as input and produces one or more XML streams. As remarked earlier, tree recognizers and the automata studied. There has also been recent work on data exchange, the complexity and expressiveness of these languages have not been studied. Section 2 reviews Organization. Section 3 defines XML Trees with local Storage. Section 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 \( \Sigma \) of tags. A tree domain \( \text{dom} \) is a tree domain, and \( \text{dom} \) is a tree domain, and \( \text{dom} \) is a tree domain. An \( \Sigma \)-tree \( t \) is defined to be \( (\text{dom}(t), \text{lab}) \), where \( \text{dom}(t) \) is a tree domain, and \( \text{lab} \) is a function from \( \text{dom}(t) \) to \( \Sigma \).

Intuitively, \( \text{dom}(t) \) is the set of the nodes in \( t \), while the empty string \( \epsilon \) represents the root of \( t \), denoted by \( \text{root}(t) \). Each node \( v \in \text{dom}(t) \) is labeled by the function \( \text{lab} \) with a tag \( a \) of \( \Sigma \), called an \( a \)-element. Moreover, \( v \) has a possibly empty list of elements as its children, denoted by \( \text{children}(v) \). Here \( v \in \text{dom}(t) \) is the \( i \)-th child of \( v \), and \( v \) is called the parent of \( v \). 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 \( \Sigma \) contains a special root tag \( r \), such that \( \text{lab}(\epsilon) = r \) and moreover, for any \( v \in \text{dom}(t) \), \( \text{lab}(v) \neq r \) if \( v \neq \epsilon \). To simplify the discussion we also assume a special tag, text, in \( \Sigma \). Any node labeled text carries a string (PCDATA) and is referred to as a text node.

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

We consider two classes of trees: for all \( v \in \text{dom}(t) \), (a) either \( R(v) \) stores a finite relation over \( D \), (b) or \( R(v) \) is a single tuple over \( D \). These are referred to as \( \Sigma \)-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 \( R(v) \) is computed via a relational query on a database over \( D \), 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 \( L \) 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; \( \Sigma \) is a finite alphabet of tags; \( \Theta \) is a function from \( \Sigma \) to \( \mathbb{N} \) associating the arity of registers \( R_{\text{reg}} \) to each \( \Sigma \)-tag \( a \); \( q_0 \) is the start state; and \( \delta \) is a finite set of transition rules such that for each \((q, a) \in Q \times \Sigma \), if \( q \neq q_0 \) and \( a \) is not the root tag \( r \), there is a unique rule of the form:

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

Here \( k \geq 0 \), and for \( i \in [1, k] \), \((q_i, a_i) \in Q \times \Sigma \), and \( \phi_i \in L \) is a query from \( R \) and \( R_{\text{reg}} \) to \( R_{\text{reg}_i} \), where \( R_{\text{reg}} \) and \( R_{\text{reg}_i} \) are \( \Theta(a_i)- \) and \( \Theta(a_i)- \) ary relation, respectively. To simplify the discussion we assume that \( a_i \neq a \) 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 \( R_{\text{reg}_i} \) 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 \((q, \text{text}) \), 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 $Q_1 = \{q_0, q\}, \Sigma_1 = \{db, course, prereq, cno, title, text\}$, and the root tag is db; we associate four sets of registers $\text{Reg}_e, \text{Reg}_p, \text{Reg}_g$ and $\text{Reg}_c$ with course, prereq, cno and title nodes, to which the arity-function $\Theta_1$ assigns 2, 1, 1, 1, respectively; finally, $\delta_1$ is defined as follows:

$\delta_1(q_0, db) = (q, \text{course}, \phi_1(\text{cno}, \text{title}, \theta_0))$, where

$\phi_1(\text{cno}, \text{title}) = \exists \text{dept}(\text{course(cno, title, dept)} \land \text{dept} = \text{CS'})$

$\delta_1(q, \text{course}) = (q, \text{cno}, \phi_2(\text{cno}, \theta_0), (q, \text{title}, \phi_2(\text{title}, \theta_0),$ $\text{prereq}, \phi_2(\text{cno}, \theta_0),)$

$\phi_2(\text{cno}) = \exists \text{cno} \text{Reg}_c(\text{cno}, \text{title})$, and

$\phi_2(\text{title}) = \exists \text{cno} \text{Reg}_c(\text{cno}, \text{title})$.

$\delta_1(q, \text{prereq}) = (q, \text{course}, \phi_2(\text{cno}, \text{title}, \theta_0))$, where

$\phi_2(\text{cno}, \text{title}, \theta_0) = \exists d'd'(\text{Reg}_p(d') \land \text{prereq}(	ext{cno}, c) \land \text{course}(c, t, d))$

$\delta_1(q, \text{cno}) = (q, \text{cno}, \phi_3(\text{cno}, \theta_0))$, where $\phi_3(\text{cno}, \theta_0) = \exists \text{prereq}(\text{cno}, \text{course})$

$\phi_3(\text{cno}, \text{title}) = \exists q \text{Reg}_g(q)$

Note that in each query $\phi(\bar{x}, \bar{y})$ in the rules, $|\bar{y}| = 0$, i.e., $\bar{y} = \emptyset$. The semantics of $\tau_1$ will be given in Example 3.2.

A publishing transducer $\tau$ can be recursive. To illustrate this we define the dependency graph $G$ of $\tau$. For each $(q, a) \in Q \times \Sigma$ there is a unique node $v(q, a) \in G$, 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 $\tau$ is recursive iff there is a cycle in $G$.  

Transformations. In a nutshell, $\tau$ generates a tree from a database $I$ of schema $\mathcal{R}$ in a top-down fashion. Initially, $\tau$ constructs a tree $t$ consisting of a single node labeled $(q_0, r)$ with an empty storage. At each step, $\tau$ expands $t$ by simultaneously operating on the leaf nodes of $t$. At each leaf $u$ labeled $(q, a)$, $\tau$ generates new nodes by finding the rule for $(q, a)$ from $\delta$, issuing queries embedded in the rule to the relational database $I$ and the register $\text{Reg}_g(u)$ 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 $\Sigma$-tree, which is the output of $\tau$.

We now formally define the transformation induced by $\tau$ from a database $I$ to a schema $\mathcal{R}$. As in [2], we assume an implicit ordering $\preceq$ on $\text{D}$, 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 $\Sigma$-trees with local storage by allowing nodes to be labeled with symbols from $\Sigma \cup \{\text{cno}, \text{title}\}$. We use $\text{Tree}_{\Sigma \cup \{\text{cno}, \text{title}\}}$ to denote the set of all such extended $\Sigma$-trees. Then, every step in the transformation rewrites a tree in $\text{Tree}_{\Sigma \cup \{\text{cno}, \text{title}\}}$, starting with the single-node tree $(q_0, r)$.

More specifically, for two trees $\xi, \xi' \in \text{Tree}_{\Sigma \cup \{\text{cno}, \text{title}\}}$, we define the step-relation $\Rightarrow_{\tau, I}$ as follows: $\xi \Rightarrow_{\tau, I} \xi'$ iff there is a leaf $u$ of $\xi$ labeled $\text{cno}$ (and $\text{title}$) 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 $\text{Reg}_g(v) = \text{Reg}_g(u)$, then $\xi'$ is obtained from $\xi$ by changing $\text{lab}(u)$ to $a$. Otherwise,

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

$$(q, a) \rightarrow (q_1, a_1, \phi_1(\bar{x}_1; y_1), \ldots, q_k, a_k, \phi_k(\bar{x}_k; y_k)).$$

If $k > 0$, then $\xi'$ is obtained from $\xi$ by rooting the forest $f_1, \ldots, f_k$ under $u$. For each $j \in [1, k]$, $f_j$ is constructed as follows. Let $\{d_1, \ldots, d_m\} = \{d \mid I \cup \text{Reg}_g(u) = \exists \text{prereq}(\text{cno}, \text{course})\}$ and $d_1 \leq \cdots \leq d_m$ with $d_i$ extended to tuples in the canonical way. Then $f_j$ is a list of nodes $\{v_1, \ldots, v_n\}$, where $v_i$ is labeled with $(q_j, a_j)$ and its register $\text{Reg}_g(v_i)$ stores the relation $\{\bar{d}_i \times \bar{e} \mid I \cup \text{Reg}_g(u) = \phi_i(\bar{d}_i; e)\}$; here we use $\text{Reg}_g$ and $\text{Reg}_c$ to denote the registers associated with the $a$-node $u$ and the $a_j$-node $v_i$, respectively. If all $f_j$’s are empty, $\xi'$ is obtained from $\xi$ by labeling the label of $u$ to $a$. In particular, if the tag $a$ is text, then in $\xi'$, $u$ carries a string representation of $\text{Reg}_g(u)$ (assuming a function that maps relations over $\text{D}$ to strings, based on the order $\preceq$).

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 $\text{Reg}_g(v)$ of $v$. Since the subtree rooted at $u$ is uniquely determined by $q, a, \text{Reg}_g(v)$ 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 [2]. As will be seen in the next section, most commercial systems support only nonrecursive 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 transformation rule. Observe that the children spawned from $u$ can be characterized by a regular expression $a_1^* \cdots a_k^*$. For each $j \in [1, k]$, the $a_j$ children are grouped by the values of $d$ of the parameter $x$ in the query $\exists \text{prereq}(\text{cno}, \text{course})$. That is, for each distinct $d \in I \cup \text{Reg}_g(u)$, the $a_j$ children will be seen in the next section, most commercial systems support only nonrecursive publishing transducers and thus do not necessarily need a stop condition.

Example 3.2: Given an instance $I_0$ of the schema $R_0$ described in Example 3.1 the publishing transducer $\tau_1$ 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 $\phi_1(x)$ on $I_0$, and for each distinct tuple in the result, it spawns a course child $v$ carrying the tuple in its register $\text{Reg}_g(v)$. At node $v$ it issues queries $\phi_2$ on $\text{Reg}_g(x)$, 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 it issues query $\phi_3$ against both $I_0$ and $\text{Reg}_g(u)$; i.e., it extracts all (immediate) prerequisites of the course $v$, for which the cno is stored in $\text{Reg}_g(u)$. 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 $\phi_3$, it generates a course child of $u$. The transformation continues until either it reaches some course for which there is no prerequisite, i.e., $\phi_3$ 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 $\Sigma$-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 $\tau$-transformation on $I$ w.r.t. $\preceq$ is the tree $\xi$ such
that \((q_0, r) \Rightarrow^* \xi\) and all leaf nodes of \(\xi\) carry a label from \(\Sigma\).
This means that \(\xi\) is final and cannot be expanded anymore. We use \(\tau(I)\) to denote the \(\Sigma\)-tree obtained from \(\xi\) by striking out the local storage and states from \(\xi\). We denote by \(\tau(R)\) the set \(\{\tau(I) \mid I\) is an instance of \(R\}\); i.e., the set of trees induced by \(\tau\)-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 \(\tau = (Q, \Sigma, \Theta, q_0, \delta, \Sigma_\xi)\), where \(\Sigma_\xi\) is a designated subset of \(\Sigma\), referred to as the virtual tags of \(\tau\), and \(Q, \Sigma, \Theta, q_0, \delta\) are the same as described in Definition 5.1. We require that \(\Sigma_\xi\) does not contain the root tag. On a relational database \(I\) the transducer \(\tau\) behaves the same as a normal transducer, except that the \(\Sigma\)-tree \(\tau(I)\) is obtained from the result \(\xi\) of the \(\tau\)-transformation on \(I\) as follows. First, the local registers and states are removed from \(\xi\). Second, for each node \(v\) in \(dom(\xi)\), if \(v\) is labeled in a tag in \(\Sigma_\xi\), we shortcut \(v\) by replacing \(v\) with \(\text{children}(v)\), i.e., treating \(\text{children}(v)\) 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 \(\Sigma_\xi\).

**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 \(\tau_2\) with virtual nodes. Indeed, capitalizing on a virtual node \(c\), we can generalize the virtual node \(\alpha\) to a virtual node \(\alpha\) in \(\mathcal{L}\) with virtual nodes. The former extracts relational data 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 \(PT_{\text{w}}(\text{FO}, \text{tuple}, \text{normal})\).

**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 \(\not\in\) (CQ), first-order logic (FO) and (inflationary) fixpoint logic (FP), all with equality \(\in\). **Store** \(S\) is either relation or tuple, indicating that the \(\Sigma\)-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 \(\tau\) with tuple stores, \([\bar{g}]_1 = 0\) in each query \(\phi_0(s_1; \bar{g})\) in \(\tau\), as illustrated in Example 3.4.

**Output O** is either normal or virtual, indicating whether a transducers allow virtual nodes or not. Thus \(PT(\text{FP}, \text{relation}, \text{virtual})\) 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, \(PT(\text{CQ}, \text{tuple}, \text{normal})\) is the smallest.

For each class \(PT(\mathcal{L}, S, O)\), we denote by \(PT_{\text{w}}(\mathcal{L}, S, O)\) its subclass consisting of all nonrecursive transducers in it.

For instance, the transducers \(\tau_1\) and \(\tau_2\) given in Examples 3.1 and 3.2 are in \(PT_{\text{w}}(\text{FO}, \text{relation}, \text{virtual})\), respectively (\(\tau_2\) is also definable in \(PT_{\text{w}}(\text{FP}, \text{tuple}, \text{normal})\); 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 19 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 \(PT_{\text{w}}(\text{FO}, \text{tuple}, \text{normal})\).

The second method specifies an XML view by annotating a (non-recursive) 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 \(PT_{\text{w}}(\text{FO}, \text{tuple}, \text{normal})\).

**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 \(PT_{\text{w}}(\text{FO}, \text{tuple}, \text{normal})\).

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 SQL queries for populating elements and attributes specified in the DAD. Neither of these two
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 is imposed in Section 3, is imposed, XML views defined in DBMS_XMLGEN are expressible in PT(FP, tuple, normal).

XPERANTO \[25\]. It supports essentially the same XML views as SQL/XML, and thus in PT_{ad}(FO, tuple, normal).

TreeQL \[11\]. 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 XML queries. It supports virtual 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_{ad}(FP, 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 \(e \rightarrow e\) in the DTD with a set of relational queries, one for each sub-element type \(b\) in the regular expression \(\alpha\), 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 ATG \[6\] adopt CQ queries, relation registers and the stop condition of Section 3. ATGs of \[5, 6\] are basically PT_{ad}(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 5.1 and 5.3 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(L, S, O) of publishing transducers. (i) The membership problem for PT(L, S, O) is to determine, given a \(\Sigma\)-tree \(t\) and a transducer \(\tau\) in this class, whether there is an instance \(I\) with \(t = \tau(I)\), i.e., \(\tau\) on \(I\) computes the tree \(t\). (ii) The emptiness problem for PT(L, S, O) is to determine, given \(\tau\) in this class, whether there is an instance \(I\) with \(\tau(I) \neq \emptyset\), i.e., the tree with the root only. So, it is to decide whether \(\tau\) can produce nontrivial trees. (iii) The equivalence problem for PT(L, S, O) is to determine, given two transducers \(\tau_1\) and \(\tau_2\) 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 \(\Sigma\)-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 3. 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 \(\tau\) in PT(L, S, O), where \(L\) 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 \(L\) is CQ. Worst-case data-complexity is \(\text{EXPTIME}\) and \(2\text{EXPTIME}\) 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 \(\tau_1\) 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 \(\tau_2\) 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(L, S, O) when \(L\) 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 \(L\) equal to CQ, the situation gets slightly better.

**Theorem 5.3:** For PT(CQ, S, O),
- the emptiness problem is decidable in \(\text{PTIME}\) for PT(CQ, S, normal), but becomes \(\text{NP}\)-complete for PT(CQ, S, virtual);
• the equivalence problem is undecidable;
• the membership problem is $\Sigma_2^p$-complete for PT(CQ, tuple, normal), but becomes undecidable when either $S$ is relation or $O$ is virtual.

Proof. For the emptiness problem for $\tau$ in PT(CQ, S, normal), it is sufficient to test emptiness of the CQ queries in the start rule of $\tau$. 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 $\tau$ in PT(CQ, relation, virtual) is proved by providing an NP algorithm that (1) guesses a path from the root of the dependency graph $G_\tau$ of $\tau$ 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., \([5]\)) 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 $\Sigma_2^p$ lower bound for the membership problem for PT(CQ, tuple, normal) in the absence of `$\neq$' is by a reduction from $\exists^+\forall^*\Sigma^p$-3SAT \([24]\). The upper bound is proved by (1) establishing a small model property: for any $\Sigma$-tree $I$ and $\tau$ in the class, if $t \in \tau(R)$, then there exists an $I$ such that $\tau(I) = t$ and $\|I\| \in \{|I|; (2)$ providing an algorithm for checking the existence of $I$ by using a nondeterministic PTIME Turing machine with an 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., \([24]\)). 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 $\tau$ 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$_{w_0}$(FO, tuple, normal) (FOR-XML, SQL_mapping, SQL/XML), PT$_{w_0}$(CQ, tuple, normal) (annotated XSD, RDB_mapping), and PT$_{w_0}$(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 $\tau$ in one of these nonrecursive classes, the $\Sigma$-tree induced by $\tau$ on any database is bounded by $\tau$. From this it follows:

Corollary 5.4: For publishing transducers $\tau$ in PT$_{w_0}$(FO, tuple, normal) (or PT$_{w_0}$(CQ, tuple,O)), the worst-case data complexity for $\tau$-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$_{w_0}$(FO, tuple, normal). The emptiness problem for PT$_{w_0}$(CQ, tuple, normal) is in PTIME; it is NP-complete for PT$_{w_0}$(CQ, tuple,O). The membership and equivalence problems for PT$_{w_0}$(CQ, tuple, O) are $\Sigma_2^p$-complete, and in $\Pi_2^p$-complete, respectively.

Proof. The proof of Proposition 5.2 remains intact for PT$_{w_0}$(FO, tuple, normal). Similarly, the PTIME upper bound for emptiness of Theorem 5.3 trivially holds for PT$_{w_0}$(CQ, tuple, virtual). Since the NP lower bound proof of Theorem 5.3 for emptiness uses a non-recursive transducer, the lower bound extends to PT$_{w_0}$(CQ, tuple, virtual). The NP upper bound of Theorem 5.3 trivially holds for PT$_{w_0}$(CQ, tuple, virtual). Similarly, for PT$_{w_0}$(CQ, tuple, O), the $\Sigma_2^p$ upper-bound proof of Theorem 5.3 for membership extends to PT$_{w_0}$(CQ, tuple, virtual). For the equivalence problem for PT$_{w_0}$(CQ, tuple, normal), we prove the $\Pi_2^p$-upper bound lower bound by reduction from the $\forall^*\exists^+\forall^*\Sigma^p$-3SAT problem. We give a $\Pi_2^p$-time checking algorithm for PT$_{w_0}$(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 Relation 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 $\tau$-transformation on $I$ yields a final tree $Q$ with local storage in Tree$_{\mathbf{Q}_R \times I}$, from which the output $\Sigma$-tree $\tau(I)$ is obtained by removing local stores and transducer states (recall from Section 5).

The relation induced by $\tau$ 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 $Q$. Therefore, we refer to $\tau$ as a relational query when $\tau$ is viewed as a mapping from instances $I$ to the relation induced by $\tau$ on $I$. When $\tau$ is viewed as a relation-to-tree mapping, we refer to $\tau$ as a tree generating mapping.

We want to compare the expressive power of one class PT($L_1, S_1, O_1$) with that of another class PT($L_2, S_2, O_2$) both as a tree generation and a relational query language. We say that PT($L_1, S_1, O_1$) is contained in PT($L_2, S_2, O_2$) as a tree/relation query language, denoted by PT($L_1, S_1, O_1$) $\subseteq$ PT($L_2, S_2, O_2$), if for any $\tau_1$ in PT($L_1, S_1, O_1$) defined for a relational schema $R$, there exists $\tau_2$ in PT($L_2, S_2, O_2$) for the same $R$ such that they define the same tree/relation query.

The two classes are said to be equivalent in expressive power, denoted by PT($L_1, S_1, O_1$) $\equiv$ PT($L_2, S_2, O_2$), if PT($L_1, S_1, O_1$) $\subseteq$ PT($L_2, S_2, O_2$) and PT($L_2, S_2, O_2$) $\subseteq$ PT($L_1, S_1, O_1$). We say that PT($L_1, S_1, O_1$) is properly contained in PT($L_2, S_2, O_2$), denoted by PT($L_1, S_1, O_1$) $\subset$ PT($L_2, S_2, O_2$), if PT($L_1, S_1, O_1$) $\subseteq$ PT($L_2, S_2, O_2$) but PT($L_2, S_2, O_2$) $\not\subseteq$ PT($L_1, S_1, O_1$). These notions extend to comparing PT($L, S, O$) vs. other tree generating formalisms, and to comparing PT($L, S, O$) vs. relational query languages.

We also characterize PT($L, S, O$) with respect to complexity classes. Treating PT($L, S, O$) as a relational query language, for example, we consider the recognition problem for its transducers $\tau$: given a tuple $t$ and an instance $I$ of the schema for which $\tau$ is defined, it is to determine whether $t$ is in the relation $R_{a_o}(\tau)$ induced by $\tau$ on $I$. We say that PT($L, S, O$) captures a complexity class $\mathcal{C}$ if the recognition problem for all transducers in PT($L, S, O$) is in $\mathcal{C}$ and moreover, for any query $q$ whose recognition problem is in $\mathcal{C}$, there exists a $\tau$ in PT($L, S, O$) defined on the same schema $R$ as $q$, such that $q$ and $\tau$ return the same output relation on all instances.
Outline. We study the expressive power of all the classes $PT(L, S, O)$ 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 $L$ have explicit access to the order $\leq$ on the domain $D$, unless explicitly stated otherwise.

6.2 Expressiveness in Terms of Relational Queries

We start by treating $PT(L, S, O)$ as a relational query language. We first review two fragments of datalog. One fragment is \textit{linear datalog} (see e.g., [4]), denoted by \textsc{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 $p_i$, to be $\varnothing$. The other, referred to as \textit{deterministic datalog} and denoted by \textsc{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 $TC_\varnothing[L]$ to denote a fragment of transitive closure logic: the set of all formulas $[TC_\varnothing \phi][\bar{a}, \bar{b}]$, where $\varphi \in L$. Following [14] one can verify that $TC_\varnothing[L] = \textsc{Lindatalog}$.

The main result of Section 6.2 is given as follows.

Theorem 6.1: When treated as relational query languages, (1) $PT(L, S, \text{virtual}) = PT(L, S, \text{normal})$, (2) $PT(CQ, \text{tuple, } O) \subseteq PT(FO, \text{tuple, } O)$, (3) $PT(FP, \text{tuple, } O) \subseteq PT(FO, \text{tuple, } O)$, (4) $PT(FP, \text{relation, } O) = PT(FO, \text{relation, } O)$, (5) $PT(CQ, \text{tuple, } O) \subseteq PT(CQ, \text{relation, } O)$, (6) $PT(\text{relation, } O) \subseteq \textsc{Ddatalog}$, where $O$ is either normal or virtual. The containment in statement (3) is proper if \textsc{Nlogspace} $\neq \textsc{Ptime}$. Moreover, (a) $PT(FO, \text{relation, } O)$ captures \textsc{Pspace}, (b) $PT(FO, \text{tuple, } O) = \text{(inflationary) } \textsc{Pspace}$ on ordered databases and thus captures \textsc{Ptime}, (c) $PT(FO, \text{tuple, } O) \subseteq \text{TC}_\varnothing[FO]$ on ordered databases, and thus captures \textsc{Nlogspace}. On unordered databases, $PT(FO, \text{tuple, } O) \subset \textsc{Nlogspace}$. (d) $PT(CQ, \text{relation, } O) \supseteq \textsc{Ddatalog}$, (e) $PT(CQ, \text{tuple, } O) = \textsc{Lindatalog}$.

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 $PT(L, S, \text{normal})$. In contrast, we have to treat publishing transducers with relation stores and those with tuple stores separately (4, 6, 8). While \textsc{FP} does not add expressive power over \textsc{FO} in $PT(L, \text{relation, } O)$, it does in $PT(L, \text{tuple, } O)$ (5, 3). Moreover, replacing \textsc{CQ} with \textsc{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 $PT(FO, \text{relation, } O)$ captures \textsc{Pspace}, we first show that for each $\tau$ in $PT(FO, \text{relation, } O)$, its recognition problem can be determined by using nondeterministic \textsc{Pspace} Turing machine. Conversely, we simulate every partial fixpoint query (known to capture \textsc{Pspace} on ordered instances) using a transducer and show that a total order is definable in this class. Similarly, we simulate each $\tau$ in $PT(FP, \text{tuple, } O)$ (resp. $PT(FO, \text{tuple, } O)$) in \textsc{FP} (resp. in $TC_\varnothing[FO]$), and vice versa. Thus on ordered databases, $PT(\text{FP, tuple, } O)$ and $PT(FO, \text{tuple, } O)$ capture \textsc{Ptime} and \textsc{Nlogspace}, respectively. On unordered structures, since the parity query even is not expressible in \textsc{FP}, it is not definable in $PT(\text{FP, tuple, } O)$. We simulate \textsc{Ddatalog} and \textsc{Lindatalog} in $PT(CQ, \text{relation, } O)$ and $PT(CQ, \text{tuple, } O)$, respectively, and vice versa for \textsc{Lindatalog}.

Statement (1) holds because for any tree $\xi$ induced by a transducer in $PT(L, S, \text{virtual})$, and for any normal $\alpha$-element $v$ in $\xi$, removing virtual nodes from $\xi$ does not change the content of the register $Reg_v(v)$. Statement (5) is verified by simulating FP queries in $PT(FO, \text{relation, normal})$. Statements (4, 6) follow from (5) and the fact that each transducer in $PT(L, \text{tuple, } O)$ is a special case of $PT(L, \text{relation, } O)$ in which for any query $\phi(x, \bar{y})$, $\bar{y}$ is the empty list. The containment in (4) is proper since on unordered structures, even is expressible in $PT(FO, \text{relation, } O)$ but not in \textsc{FP}. We show that the containment of (6) is proper by defining a transducer $\tau$ in $PT(CQ, \text{relation, } O)$ 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 $\tau$ is not expressible even in $PT(FO, \text{tuple, } O)$ (this requires an EF-game argument to show that the relational query defined by $\tau$ is not definable in \textsc{FO}). From this also follows (8). To prove that the containments in (2, 7) are proper, we give an \textsc{FO} query $q$, which is clearly definable in $PT(FO, \text{tuple, } O)$, and show that $q$ is not definable in $PT(CQ, \text{relation, } O)$ due to the monotonicity of \textsc{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 to \textsc{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 $L \subset L'$, then $PT(L', S, \text{normal})$ properly contains $PT(L, S, \text{normal})$ whereas in the relational query setting, $PT(\text{FP, relation, normal}) = PT(FO, \text{relation, normal})$. The other results in Proposition 6.2 are comparable to their counterparts in Theorem 6.1.

Proposition 6.2: For tree generation, (1) $PT(L, S, \text{normal}) \subset PT(L, S, \text{virtual})$, (2) $PT(L, S, \text{normal}) \subset PT(L', S, \text{normal})$ if $L \subset L'$, (3) $PT(CQ, \text{tuple, virtual}) \subset PT(FO, \text{tuple, virtual})$ (4) $PT(FP, \text{tuple, virtual}) \subseteq PT(FO, \text{tuple, virtual})$, (5) $PT(CQ, \text{relation, virtual}) \subset PT(FO, \text{relation, virtual})$, (6) $PT(CQ, \text{tuple, } O) \subseteq PT(FO, \text{relation, } O)$, (7) $PT(CQ, \text{tuple, } O) \subseteq PT(L, \text{relation, } O)$, (8) $PT(CQ, \text{relation, normal}) \not\subseteq PT(FO, \text{tuple, virtual})$, where $L_1, L_2$ are \textsc{FP}, \textsc{FO} or \textsc{CQ}, and $O$ is normal or virtual. The containment in (4) is proper if \textsc{Ptime} $\neq \textsc{Nlogspace}$.

Proof. We prove that the containments in (1-5) are proper as follows. For (1), we define $\tau_1$ in $PT(CQ, \text{tuple, virtual})$ that can generate a $\Sigma_2$-tree in which the root has exponentially many children, which is not doable even by transducers in $PT(\text{FP, relation, normal})$. For (2-5), observe that by Theorem 6.1 there exists a Boolean query $q$ in $L'$ not expressible in $PT(L', S, O)$ if $PT(CQ, S, O)$ is considered as a relational query language (for (4) if \textsc{Ptime} $\neq \textsc{Nlogspace}$). We define $\tau_2$ in $PT(L', S, \text{normal})$ such...
that $\tau_2$ 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($L$, 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 $\phi_\times, \phi_<$ and $(\phi_n)_{n \in \Sigma}$ such that on every $R$-structure $I$, $\phi_\times(I), \phi_<(I)$ and $\phi_n(I)$ 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), $\phi_n(I)$ 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 $\ell$ such that for any input $I$, $T(I)$ is a tree of depth at most $\ell$. In a similar way to logical transductions, we can also define $C$-transductions (both first and second order) for a complexity class $C$ where there are $C$-Turing machines to decide the relations $\phi_\times, \phi_<$ and $(\phi_n)_{n \in \Sigma}$.

**Theorem 6.3:**
1. When $\mathcal{L}$ ranges over CQ, FO and FP, every $\mathcal{L}$-transduction is definable in PT($L$, tuple, virtual).
2. When $\mathcal{L}$ ranges over FO and FP, every transducer in PT$_{un}(\mathcal{L}$, 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$_{un}(\mathcal{L}$, 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($L$, 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$_{un}$(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 $\Sigma$ is defined by a set of rules of the form $a \rightarrow \alpha$, where $a$ is a tag in $\Sigma$ and $\alpha$ is a regular expression over $\Sigma$. A $\Sigma$-tree $t$ conforms to $d$ if for each $a$-element $v$ in $t$, the list of labels of children($v$) is a string in $\alpha$.

A specialized DTD $d$ over $\Sigma$ is a triple $(\Sigma', d', g)$, where $\Sigma \subseteq \Sigma'$, $g$ is a mapping $\Sigma' \rightarrow \Sigma$, and $d'$ is a DTD over $\Sigma'$. A $\Sigma$-tree $t$ conforms to $d$ if there exists a $\Sigma'$-tree $t'$ that satisfies $d'$ and moreover, $t = g(t')$. We denote by $L(d)$ the set of all $\Sigma$-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 $\tau$ 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 discussion 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 $\Sigma$ 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 $\tau$ 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$_{XML}$GEN and ATGs, respectively. The theorem below settles the issue for PT$_{un}$($\mathcal{L}$, tuple, normal) (FOR-XML, SQL$_{mapping}$, XPERANTO, SQ$\_XML$), PT$_{un}$($\mathcal{L}$, tuple, $O$) (annotated XSD, RDB$_{mapping}$, TreeQL).

**Theorem 6.5:** When treated as relational query languages, PT$_{un}$($\mathcal{L}$, tuple, $O$) = FO, and PT$_{un}$($\mathcal{L}$, tuple, $O$) = UCQ (UCQ denotes union of conjunctive queries with ‘$\lor$’, ‘$\neg$’).

**Proof.** Every UCQ query can be simulated in PT$_{un}$($\mathcal{L}$, tuple, $O$). Conversely, for each transducer $\tau$ in PT$_{un}$($\mathcal{L}$, tuple, $O$) and a designated output tag $a_o$, the output relation $R_{\mathcal{L}}(a_o)$ of a $\tau$-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 $\tau$ from the root to a leaf node labeled $a_o$. Similarly, PT$_{un}$($\mathcal{L}$, 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$_{un}$($\mathcal{L}$, tuple, normal) $\subseteq$ PT$_{un}$($\mathcal{L}$, tuple, normal) $\subseteq$ PT$_{un}$($\mathcal{L}$, tuple, virtual). Theorem 6.3 has shown that for unordered trees fixed-depth FO-transductions are equivalent to PT$_{un}$($\mathcal{L}$, 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 $\mathcal{L}$.

**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 \( PT_w(\mathcal{L}, S, O) \) characterizing existing publishing languages, we 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 open question concerns, when treated as a relational query (or tuple), whether or not \( PT(CQ, relation, O) \) captures \( \text{DATALOG} \)? We only know that \( \text{DATALOG} \) is contained in \( PT(CQ, relation, O) \) and nonrecursive \( PT_w(CQ, relation, O) \). Another 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 EPSRC GR/S63205/01, GR/D006473/1.

8. References

7. E. Börger, E. Grädel, and Y. Gurevich. The Classical Decision

\[\begin{array}{|c|c|c|c|}
\hline
\text{Fragments} & \text{Equivalence} & \text{Emptiness} & \text{Membership} \\
\hline
\text{PT}(\mathcal{F}, \mathcal{P}, \mathcal{O}) \quad (\text{Th. 5.2}) & \text{undecidable} & \text{undecidable} & \text{undecidable} \\
\text{PT}(\mathcal{F}, \mathcal{O}) \quad (\text{Th. 5.2}) & \text{undecidable} & \text{undecidable} & \text{undecidable} \\
\text{PT}(\mathcal{O}, \mathcal{P}) \quad (\text{Th. 5.3}) & \text{2\text{-}complete} & \text{undecidable} & \text{undecidable} \\
\text{PT}(\mathcal{O}, \mathcal{O}) \quad (\text{Th. 5.3}) & \text{2\text{-}complete} & \text{undecidable} & \text{undecidable} \\
\text{PT}(\mathcal{O}, \mathcal{P}) \quad (\text{Th. 5.5}) & \text{undecidable} & \text{NP-complete} & \text{undecidable} \\
\text{PT}(\mathcal{O}, \mathcal{O}) \quad (\text{Th. 5.5}) & \text{undecidable} & \text{NP-complete} & \text{undecidable} \\
\end{array}\]

\[\begin{array}{|c|c|}
\hline
\text{Fragments} & \text{Complexity/Language} \\
\hline
\text{PT}(\mathcal{F}, \mathcal{O}) \quad (\text{Th. 6.1}) & \text{PSPACE} \\
\text{PT}(\mathcal{F}, \mathcal{O}) \quad (\text{Th. 6.1}) & \text{PSPACE} \\
\text{PT}(\mathcal{F}, \mathcal{O}) \quad (\text{Th. 6.1}) & \text{NP-complete} \\
\text{PT}(\mathcal{F}, \mathcal{O}) \quad (\text{Th. 6.1}) & \text{NP-complete} \\
\text{PT}(\mathcal{O}, \mathcal{O}) \quad (\text{Th. 6.1}) & \text{PSPACE (ordered database)} \\
\text{PT}(\mathcal{O}, \mathcal{O}) \quad (\text{Th. 6.1}) & \text{PSPACE (ordered database)} \\
\text{PT}(\mathcal{O}, \mathcal{O}) \quad (\text{Th. 6.1}) & \text{PSPACE (ordered database)} \\
\text{PT}(\mathcal{O}, \mathcal{O}) \quad (\text{Th. 6.1}) & \text{PSPACE (ordered database)} \\
\end{array}\]

Table 2: Complexity of decision problems (\( \mathcal{S} \): relation or tuple; \( \mathcal{O} \): normal or virtual; tp: tuple; rt: relation; nm: normal; vr: virtual)

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

\[\text{Problem}, \text{Springer, 1997.}\]

8. References

7. E. Börger, E. Grädel, and Y. Gurevich. The Classical Decision