Expressiveness and Complexity of XML Publishing Transducers

WENFEI FAN
University of Edinburgh and Bell Laboratories
FLORIS GEERTS
University of Edinburgh
and
FRANK NEVEN
Hasselt University and Transnational University of Limburg

A number of languages have been developed for specifying XML publishing, that is, 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 article proposes a notion of *publishing transducers*, which generate XML trees from relational data. 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, that is, excluded from the output tree. We first show how existing XML publishing languages can be characterized by such transducers, and thus provide a synergy between theory and practice. We then study the membership, emptiness, and equivalence problems for various classes of transducers. We establish lower and upper bounds, all matching, 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, among other things.

Categories and Subject Descriptors: H.2.5 [**Database Management**]: Heterogeneous Databases—*Data translation*; H.1.m [**Models and Principles**]: Miscellaneous

General Terms: Design, Languages, Theory

W. Fan is supported in part by EPSRC GR/S63205/01, GR/T27433/01 and EP/E029213/1. F. Geerts is supported in part by EPSRC GR/S63205/01 and EP/E029213/1.

Authors' addresses: W. Fan and F. Geerts, School of Informatics, Laboratory for Foundations of Computer Science, Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, Scotland, UK; email: {wenfei,fgeerts}@inf.ed.ac.uk; F. Neven, Research Group on Theoretical Computer Science, Department WNI, Agoralaan, B-3590 Diepenbeek, Belgium; email: frank.neven@uhasselt.be.

Additional Key Words and Phrases: XML publishing, data exchange, transducer, complexity, expressiveness

ACM Reference Format:

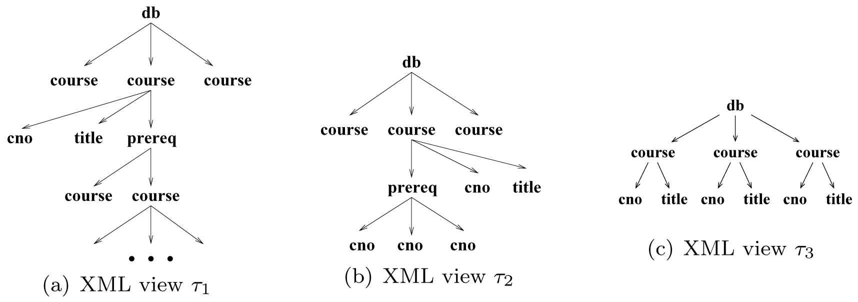
Fan, W., Geerts, F., and Neven, F. 2008. Expressiveness and complexity of XML publishing transducers. ACM Trans. Datab. Syst. 33, 4, Article 25 (November 2008), 49 pages. DOI = 10.1145/1412331.1412337 http://doi.acm.org/10.1145/1412331.1412337

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 [Alon et al. 2003; Benedikt et al. 2002; Fernandez et al. 2002; Krishnamurthy et al. 2003; Shanmugasundaram et al. 2001], 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.

A number of languages have been developed for XML publishing, including commercial products such as: annotated XSD, in Microsoft SQL Server 2005 [Microsoft 2005]; DAD, in IBM DB2 XML Extender [IBM], DBMS_XMLGEN of Oracle 10g XML DB [Oracle], and research prototypes XPERANTO [Shanmugasundaram et al. 2001], TreeQL [Fernandez et al. 2002; Alon et al. 2003] and ATG [Benedikt et al. 2002; Bohannon et al. 2004]. 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 [Alon et al. 2003; Benedikt et al. 2002]. It may also allow *virtual* nodes that are temporary, that is, they are eliminated from the final output tree. The usefulness of virtual nodes for XML publishing is illustrated in Alon et al. [2003] and Benedikt et al. [2002].

Just as with relational view definition languages, associated with XML publishing languages are a number of fundamental questions in connection with their complexity and expressiveness. These questions are not only of theoretical interest, but are also important in practice to both users and designers of XML publishing languages. 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 always yields an empty tree? Is this view equivalent to another view? To support recursively-defined XML views in a publishing language, database vendors may want to know whether or not certain high-end DBMS features are a must: is it necessary to upgrade the DBMS to support linear recursion of SQL'99 [Melton and Simon 1993]?



(a) XML view τ_1

(b) XML view τ_2

(c) XML view τ_3

Fig. 1. Example XML publishing.

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 a *course* relation and a relation *prereq*, in which a tuple (c_1, c_2) indicates that c_2 is an *immediate prerequisite* of c_1. That is, relation *prereq* gives the prerequisite hierarchy of the courses. The registrar office wants to export three 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* (number) and *title* of the course, as well as its prerequisite hierarchy. As shown in Figure 1(a), the depth of the *course* sub-tree is determined by its prerequisite hierarchy.

—View τ_2 is a tree of depth three, listing all the CS *course*s as depicted in Figure 1(b). Below each *course c* is a *prereq* child, followed by the *cno* and *title* of *c*; under *prereq* is the list of all the *cno*s that appear in the prerequisite *hierarchy* of *c*.

The user may ask the questions mentioned previously 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.

—XML view τ_3 is a tree of depth two, listing all the *course*s extracted from the database I_0 that do not have DB as its immediate prerequisite. Under each *course* element, its *cno* and *title* are listed.

We will see that most commercial languages can express this view.

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 to users, by providing 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 on 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 source, 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 automata for querying XML data [Neven 2002], it generates a new XML tree rather than evaluating a query on an existing tree. In order to encompass publishing languages used in practice, 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 (CQ), first-order queries (FO), and (inflationary) fixpoint queries (IFP), all with '$=$' and '$\neq$';

—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 previously. 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 embedded in some class of publishing transducers. For example, annotated XSD [Microsoft 2005] is a class of "nonrecursive" PT(CQ, tuple, normal), the FOR-XML constructs [Microsoft 2005] correspond to a class of nonrecursive PT(FO, tuple, normal), DBMS_XMLGEN [Oracle] can be expressed in PT(IFP, tuple, normal), and SQL/XML [IBM] is a class of nonrecursive PT(IFP, tuple, normal). Moreover, relation stores and virtual nodes are needed to characterize TreeQL [Fernandez et al. 2002; Alon et al. 2003] and ATG [Benedikt et al. 2002; Bohannon et al. 2004]. Conversely, for many classes PT($\mathcal{L}, S, O$) there are existing publishing languages corresponding to them. For a few, there are no corresponding commercial systems. For example, no commercial language corresponds to PT(IFP, relation, virtual). Our results, however, show that this class does not increase the expressive power over PT(FO, relation, virtual), and for the latter, a running prototype system [Benedikt et al. 2002] has already been in place.

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 can generate a non-empty XML tree (emptiness), whether an XML tree of particular interest can be generated by a publishing transducer (membership), and whether a more efficient publishing transducer can in fact generate the same set of XML trees as an expensive one (equivalence). We establish *matching* lower and upper bounds for all these problems, ranging from PTIME to undecidable, for all the classes PT($\mathcal{L}, S, O$) and for the special cases that contain publishing languages being used in practice. 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 leave the characterization open. For example, the largest class PT(IFP, relation, virtual) captures PSPACE, and the smallest PT(CQ, tuple, normal) captures linear datalog (see, e.g., [Grädel 1992]). Along the same lines, we characterize the existing publishing languages. For example, we show that SQL/XML [IBM] is in FO, and annotated XSD [Microsoft 2005] is in the language of unions of CQ queries.

For tree generation, we establish connections between certain fragments of PT($\mathcal{L}, S, O$) and logical interpretations [Flum and Ebbinghaus 1999] or logical transductions [Courcelle 1994]. For example, we show that PT($\mathcal{L}$, tuple, virtual) contains the $\mathcal{L}$-transducers for $\mathcal{L}$ ranging over CQ, FO and IFP, and that regular unranked tree languages are contained in PT(FO, tuple, normal) but not in PT(CQ, relation, virtual). Furthermore, we show the ability and inability of certain fragments of PT($\mathcal{L}, S, O$) for defining DTDs and specialized DTDs [Papakonstantinou and Vianu 2000] and, as a result, regular tree languages and MSO definable trees.

In both settings, we also provide separation and equivalence results for various classes of publishing transducers. For example, we show that PT(IFP, 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(IFP, relation, normal) but in contrast, PT(FO, relation, virtual) and PT(IFP, 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, indefinite order databases, and a wide range of simulations and reductions.

Related work. As remarked earlier, a number of XML publishing languages have been proposed (see [Krishnamurthy et al. 2003] 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. [Arenas and Libkin 2005; Fagin et al. 2005]. This article differs in that we focus on (a) transformations from relational data to XML, as defined in terms of transducers with embedded relational queries, rather than relation-to-relation [Fagin et al. 2005] or XML-to-XML [Arenas and Libkin 2005] 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 [Gécseg and Steinby 1996] for a survey), some particularly for XML (e.g., [Ludäscher et al. 2002; Milo et al. 2003; Neven 2002; Neven and Schwentick 2002]). As remarked earlier, tree recognizers [Gécseg and Steinby 1996] and the automata for querying XML [Neven 2002; Neven and Schwentick 2002] 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 Milo et al. [2003] 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 [Ludäscher et al. 2002] 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 (and therefore, relational view definition languages; see, e.g.,[Abiteboul et al. 1995; Dantsin et al. 2001] 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., [Courcelle 1994] for a survey of graph transductions). Recently logical tree-to-tree interpretations are used in Benedikt and Koch [2006] to characterize XQuery. We employ transductions to characterize the tree generating power of publishing transducers.

This article is an extension of earlier work [Fan et al. 2007] by including (a) proofs for all the theorems; some of the proofs are nontrivial and the techniques are interesting in their own right; (b) matching lower bounds for the equivalence problem for two nonrecursive classes of publishing transducers (Section 5); and (c) more detailed discussions of XML publishing languages being used in practice (Section 4).

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 our main results and outlines future research directions. Due to the space limitations, some of the proofs are moved to the electronic appendix.

2. XML TREES WITH LOCAL STORAGE

We first review XML trees and then introduce a notion of trees with registers. We also review relational query languages considered in this article.

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 ith child of v, and v is called the parent of $v.i$. Note that t is *unranked*, that is, 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*, denoted by r, unless specified otherwise, 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. Only leaf nodes can be labeled with *text*; they carry a string (PCDATA) and are referred to as a *text* nodes. A node can have both *text* and non-*text* nodes as children.

Trees with local storage. We study Σ-trees generated from relational data, in a context-dependent fashion. To do this, one needs to pass information top-down, and 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 registers 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 register* or the *register* of v, and use Tree_Σ 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* registers and *tuple* registers, respectively. Note that trees with tuple registers are a special case of trees with relation registers. 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.

Relational query languages. A relational schema R is a finite collection of relation names and associated arities. We consider conjunctive queries over R built up from atomic formulas—including relations in R, equality ($=$), and inequality ($\neq$)—by closing under conjunction $\wedge$ and existential quantification $\exists$. We refer to this class of queries as CQ. First-order queries (FO) are built from these atomic formulas using conjunction $\wedge$, disjunction $\vee$, negation $\neg$, and universal $\forall$ and existential $\exists$ quantifications. We also consider inflational fixpoint queries (IFP), an extension of FO, with the following formation rule: If $\varphi(S, \bar{x})$ is an IFP formula, where S is k-ary, $\bar{x}$ are free variables of φ and $|\bar{x}| = k$, and $\bar{t}$ is a tuple of terms, where $|\bar{t}| = k$, then $[\mu^{+}_{S,\bar{x}}(\varphi(S, \bar{x}))](\bar{t})$ is an IFP formula whose free variables are those in $\bar{t}$. Given an instance I of R,

$I \models [\mu^+_{S,\bar{x}}(\varphi(S,\bar{x}))](\bar{a})$ if and only if $\bar{a}$ is in the inflationary fixed point $\mu^+(F_\varphi)$ of the mapping $F_\varphi : \mathcal{P}(\mathbf{D}^k) \rightarrow \mathcal{P}(\mathbf{D}^k)$. Here, $\mathcal{P}(\mathbf{D}^k)$ denotes the powerset of $\mathbf{D}^k$ and $F_\varphi(X) = \{\bar{a} \mid I \models \varphi(X/R, \bar{a})\}$, where $\varphi(X/R, \bar{a})$ means that R is interpreted by $X \in \mathcal{P}(\mathbf{D}^k)$. That is, $\mu^+(F_\varphi)$ is the union of all sets J^i, where $J^0 = \emptyset$ and $J^{i+1} = J^i \cup F_\varphi(J^i)$ for $i > 0$ (see, e.g., [Abiteboul et al. 1995; Libkin 2004] for detailed discussions).

3. 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 and creates the root of the tree. It then treats the leaf nodes in the tree created so far as current nodes, and expands the tree by spawning the children of all the current nodes in parallel, following *deterministically* a transition based on the current state of the transducer, and the tag and register of each current node. The transition directs how the children of a node are generated, by providing the tags of the children, as well as relational queries that extract data from the source. The process proceeds until all current nodes satisfy certain stop conditions. We next formally define publishing transducers.

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 $(q, a) \in (Q \setminus \{q_0\}) \times (\Sigma \setminus \{r\}) \cup \{(q_0, r)\}$:

(i) if $a \notin \{text\}$, 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 $i \in [1, k]$, $(q_i, a_i) \in (Q \setminus \{q_0\}) \times (\Sigma \setminus \{r\})$, and $\phi_i(\bar{x}_i; \bar{y}_i) \in \mathcal{L}$ 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, and where $\bar{x}_i$ and $\bar{y}_i$ are disjoint sets of variables. The different roles of the sets of variables $\bar{x}_i$ and $\bar{y}_i$ will be explained shortly. The rule for (q_0, r) is referred to as the *start rule* of τ. We always assume $\Theta(r) = 0$. Moreover, to simplify the discussion, we assume that $a_i \neq a_j$ if $i \neq j$.

(ii) if $a = text$, then $(q, a) \rightarrow$. That is, its rule has an empty righthand side (RHS).

In a nutshell, τ is a *deterministic* transducer that 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. For each $i \in [1, k]$, the a_i children and their associated registers Reg_{a_i} are produced as follows. The query $\phi_i(\bar{x}_i; \bar{y}_i)$ extracts data from a database instance of R and from the parent register Reg_a. The result of the query is grouped by the distinct tuples corresponding to the variables in $\bar{x}_i$, yielding sets of tuples $S_1, \ldots, S_m$. For each set S_j, a distinct a_i child is created, carrying S_j as the content of its register Reg_{a_i}. These a_i children are ordered based on an implicit ordering on the domain of data. If $|\bar{x}_i| = 0$, then no grouping takes place and the query result is partitioned in one relation S_1 (i.e., $m = 1$). If $|\bar{y}_i| = 0$, then the result is grouped by the entire tuple, and each S_j consists of single tuple only (i.e.,$|S_j| = 1$, $1 \leq j \leq m$). In general, there might be several sets S_j that might contain more than one tuple. When the result is grouped by the entire tuple (i.e.,$|\bar{y}_i| = 0$), we refer to each register Reg_{a_i} as a *tuple register*. Otherwise, Reg_{a_i} is called a *relation register*. Hence tuple registers are a special case of relation registers. 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.

Transformations. We now formally define *the transformation induced by* τ from a database I. As in Alon et al. [2003], 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 u labeled with (q_0, r) and $Reg_r(u) = \emptyset$ (recall that $\Theta(r) = 0$). More specifically, this is determined by a step-relation.

For two trees $\xi, \xi' \in \text{Tree}_{Q\times\Sigma}$, we define the step-relation $\Rightarrow_{\tau,I}$ as follows: $\xi \Rightarrow_{\tau,I} \xi'$ if and only if 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 the label (q, a) of 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 lists of nodes $f_1 \cdots f_k$ 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 $\{\bar{d}_i\} \times \{\bar{e} \mid I \cup Reg_a(u) \models \phi_j(\bar{d}_i; \bar{e})\}$, where Reg_a and Reg_{a_j} 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 $k = 0$, that is, the RHS of the rule is empty, then ξ' is obtained from ξ by labeling u with a. Moreover, if 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 second condition (2) states how to generate the children of the leaf u via a transduction rule. As remarked earlier, 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 $Reg_{a_j}(w)$.

Stop condition. The first condition (1), 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. This stop condition is the same as the one used in ATGs [Bohannon et al. 2004].

The transformation stops at the leaf u, that is, 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 which case all the forests f_j are empty; or (c) the RHS of the rule for (q, a) is empty, that is, $k = 0$ in condition (2) above—this is particularly the case for $a = text$, since 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.

Recursive vs. nonrecursive transducers. 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 $v(q', a')$ if and only if (q', a') is on the RHS of the rule for (q, a). We say that the transducer τ is *recursive* if and only if there is a cycle in G_τ. As will be seen in the next section, most commercial systems support only *nonrecursive* publishing transducers. Nonrecursive publishing transducers do not necessarily need a stop condition.

We next illustrate the syntax and semantics of publishing transducers. For ease of readability, we abuse notation and write $\emptyset$ rather than () for the empty sequence variables.

Example 3.2. The view shown in Figure 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 six sets of registers—$Reg_{db}, Reg_c, Reg_p, Reg_\#, Reg_t$, and Reg_{text}—with *db, course, prereq, cno, title*, and *text* nodes, to which the arity-function Θ_1 assigns 0, 2, 1, 1, 1, 1, respectively; finally, δ_1 is defined as follows:

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

$\delta_1(q, course) = (q, cno,\ \phi_2^1(cno; \emptyset)), (q, title,\ \phi_2^2(title; \emptyset)), (q, prereq,\ \phi_2^1(cno; \emptyset))$,
where $\phi_2^1(cno) = \exists\, title\ Reg_c(cno, title)$, and $\phi_2^2(title) = \exists\, cno\ Reg_c(cno, title)$

$\delta_1(q, prereq) = (q, course,\ \phi_3(cno, title; \emptyset))$, where
$\phi_3(c, t) = \exists\, c'\ d\ (Reg_p(c') \wedge prereq(c', c) \wedge course(c, t, d))$

$\delta_1(q, cno) = (q, text,\ \phi_4(cno; \emptyset))$, where $\phi_4(c) = Reg_\#(c)$ /*similarly for δ_1
$(q, title)$ */

$\delta_1(q, text) = .$ /* empty RHS */.

In each query $\phi(\bar{x}; \bar{y})$ in the rules, $|\bar{y}| = 0$, i.e., $\bar{y}$ is $\emptyset$. Thus Reg_c, Reg_p, $Reg_{\#}$, Reg_t, and Reg_{text} are tuple registers. The semantics of τ_1 is as follows. Given an instance I_0 of the schema R_0 described in Example 1.1, the publishing transducer τ_1 first generates the root of the tree t, labeled with (q_0, db). The register of the root node is empty by default. 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 $Reg_c(v)$, 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 $Reg_p(u)$; that is, it extracts all immediate prerequisites of the course of node 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, that is, ϕ_3 returns empty at its *prereq* child, or a course requires itself as a prerequisite, 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 Figure 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. Note also that τ_1 is recursive: G_{τ_1} contains the cycle $(v(q, course), v(q, prereq)), (v(q, prereq), v(q, course))$.

Output. We denote by $\Rightarrow^*_{\tau,I}$ the reflexive and transitive closure of $\Rightarrow_{\tau,I}$. The result of the τ*-transformation* on I with respect to $\leq$ 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 any more. 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 $\{\tau(I) \mid I \text{ is an instance of } R\}$, that is, 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 versus normal nodes. 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), that is, 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 Figure 1(b), and that the query language $\mathcal{L}$ is FO. One can show, via a simple argument using an Ehrenfeucht-Fraïssé (EF)-style game, that this is not expressible as a normal transducer of Definition 3.1 with FO (see, e.g., [Libkin 2004] 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 the same as $\delta_1(q_0,db)$ and $\delta_1(q,course)$ in Example 3.2

$\delta_2(q, prereq) = (q, l,\ \varphi_1(\emptyset; cno))$, where $\varphi_1(c) = \exists c'\ (Reg_p(c') \wedge prereq(c', c))$

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

In φ_1 and φ_1', $|\bar{x}| = 0$ and thus the result of φ_1 and 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 Figure 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 (IFP), 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. More specifically, 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 3.2. *Output O* is either normal or virtual, indicating whether a transducer allows virtual nodes or not. Thus PT(IFP, relation, virtual) is the largest class considered in this article, 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 3.2 an 3.3, are in PT(CQ, tuple, normal) and PT(FO, relation, virtual), respectively τ_2 is also definable in PT$_{nr}$(IFP, tuple, normal); we omit this definition due to the lack of space.

4. CHARACTERIZATION OF XML PUBLISHING LANGUAGES

We examine XML publishing languages that are either supported by commercial products or are representative research proposals (see [Krishnamurthy et al. 2003] for a survey). We classify these languages in terms of various classes of

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

Fig. 2. An XML view expressed with the FOR-XML construct of Microsoft SQL Server 2005.

publishing transducers with certain restrictions. We do not provide an exact correspondence between existing languages and classes of publishing transducers, but instead identify, for each language, the smallest class of publishing transducers that can express them. Furthermore, we make the implicit assumption that SQL and FO coincide (See [Libkin 2003] for a discussion concerning the differences between SQL and FO), and, similarly, that recursive SQL (when supported) can be embedded in IFP. All examples in this section refer to XML views of a registrar database I_0, as specified in Example 1.1.

Microsoft SQL Server 2005 [Microsoft 2005]. Two main XML publishing methods are supported by Microsoft: 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). For example, Figure 2 defines the XML view of Figure 1(c) using the FOR-XML construct. In a nutshell, the view is a tree of depth two, containing the list of all courses in I_0 that do not have DB as its immediate prerequisite, that is, for any such *course* c, (c, c') is not in *prereq* if the title of c' is DB.

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. Annotated XSD schema 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 [IBM]. IBM also supports two main methods, namely, SQL/XML and document access definition (DAD).

The first method extends SQL by incorporating XML constructs XMLELEMENT, XMLATTRIBUTE, XMLFOREST, XMLCONCAT, XMLAGG, and XMLGEN. It extracts relational data in parallel with XML-element creation. Nested queries are used to

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

Fig. 3. An XML view expressed in SQL/XML.

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

Fig. 4. An XML view expressed in terms of SQL_MAPPING of IBM DB2 XML Extender.

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. Although only nonrecursive XML trees can be generated, recursive SQL queries can be used to populate its elements. Indeed, IBM supports recursive SQL queries by means of Common Table Expressions. Hence, SQL/XML is essentially PT_{nr}(IFP, tuple, normal). For instance, Figure 3 shows the view of Figure 2 expressed in SQL/XML.

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 constructs, one for each tuple column, 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 view of Figure 2, for example, can be expressed in terms of SQL_MAPPING as shown in Figure 4.

The latter embeds nested RDB_NODE expressions in a DAD. The DAD is basically a tree template with a fixed depth, and the 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}(IFP, tuple, normal), and RDB_MAPPING in PT_{nr}(CQ, tuple, normal).

Oracle 10g XML DB [Oracle]. Oracle supports SQL/XML as described previously, 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 resulting from the joins, a child node is created whose children are in turn created in the next iteration of the fixpoint computation. For example, Figure 5 shows a recursive XML view containing the list of all courses;

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

Fig. 5. An XML view expressed in terms of DBMS_XMLGEN of Oracle 10*g* XML DB.

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

Fig. 6. An XML view expressed in ATG of PRATA.

under each course c are the cno and title of c followed by the hierarchy of the prerequisite courses of c.

DBMS_XMLGEN allows neither virtual nodes nor an explicit stop condition. If the stop condition given in Section 3 is imposed, XML views defined in DBMS_XMLGEN are expressible in PT(IFP, tuple, normal).

XPERANTO [Shanmugasundaram et al. 2001]. XPERANTO supports essentially the same XML views as SQL/XML, namely, those definable in PT_{nr}(FO, tuple, normal).

TreeQL [Fernandez et al. 2002; Alon et al. 2003]. TreeQL was proposed for the XML publishing middleware SilkRoute. Here, we consider its abstraction developed in Alon et al. [2003]. 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 [Benedikt et al. 2002; Bohannon et al. 2004]. Attribute transformation grammar (ATG) was proposed in Benedikt et al. [2002] and revised in Bohannon et al. [2004] for XML publishing middleware PRATA. An ATG defines an XML view based on a (normalized) 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 that access the underlying data source and the register associated with a. More specifically, for each sub-element type b in the regular expression α, it defines a query to populate the b sub-elements of an a element with the result of the query. It supports recursive DTDs, and thus recursive XML views, as well as virtual nodes to cope with XML entities. For example, Figure 6 shows an ATG that lists all courses in I_0 and is required to conform to a DTD d_0. It lists all productions of d_0 and, below each production, it specifies the queries for spawning sub-elements. While the early version of Benedikt et al. [2002] employs FO queries and tuple registers, the revised ATGs [Bohannon et al. 2004] adopt relation registers and the stop condition of Section 3. Thus ATGs are expressible in PT(FO, relation, virtual).

Table I. Characterization of Existing XML Publishing Languages

Language		Publishing transducers
Microsoft SQL Server 2005	FOR XML	PT$_{nr}$(FO, tuple, normal)
	annotated XSD	PT$_{nr}$(CQ, tuple, normal)
IBM DB2 XML Extender	SQL/XML	PT$_{nr}$(IFP, tuple, normal)
	DAD (SQL_MAPPING)	PT$_{nr}$(IFP, tuple, normal)
	DAD (RDB_MAPPING)	PT$_{nr}$(CQ, tuple, normal)
Oracle 10g XML DB	SQL/XML	PT$_{nr}$(FO, tuple, normal)
	DBMS_XMLGEN	PT(IFP, tuple, normal)
XPERANTO		PT$_{nr}$(FO, tuple, normal)
TreeQL		PT$_{nr}$(CQ, tuple, virtual)
ATG		PT(FO, relation, virtual)

The classification is summarized in Table I, which, for each publishing language mentioned previously, gives the smallest class of publishing transducers that can express all XML views definable in the language. Except for DBMS_XMLGEN and ATGs, these languages do not support recursive XML views of relational data. Indeed, one can verify, via a simple EF-game argument, that the XML views of Example 3.2 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. As remarked in Section 1, the static analyses of these problems are important in practice. Consider a class PT($\mathcal{L}$, S, O) of publishing transducers.

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 or not there exists an instance I such that $t = \tau(I)$, that is, τ evaluated on I returns the tree t.

The *emptiness problem* for PT($\mathcal{L}$, S, O) is to determine, given τ in this class, whether there exists an instance I with $\tau(I) \neq r$, that is, the tree with the root only. In other words, it is to decide whether τ can induce *nontrivial trees*.

The *equivalence problem* for PT($\mathcal{L}$, S, O) is to determine, given two transducers τ_1 and τ_2 in this class, both defined for relational databases of the same schema R, whether or not $\tau_1(I) = \tau_2(I)$ for all instances I of R, that is, the two transducers produce the same Σ-trees on all the instances of R.

We first establish matching upper and lower bounds for these problems, for all classes of transducers defined in Section 3. We then revisit the decision problems 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 Basic Complexity for Publishing Transducers

We first give some basic complexity bounds for computing the transformations defined by publishing transducers. As is accustomed, we define the size of a tree as its number of nodes.

Proposition 5.1. *Let τ be a publishing transducer in PT($\mathcal{L}$, S, O). Let I be an instance.*

(1) *The τ-transformation on I always terminates and returns a unique tree $\tau(I)$.*
(2) *Computing the output tree $\tau(I)$ can be done in time, exponential and doubly exponential in the size of I, for the cases where S is tuple and relation, respectively, and where $\mathcal{L}$ is CQ, FO, or IFP, and O is normal or virtual.*
(3) *There is a publishing transducer τ_1 in PT(CQ, tuple, normal) and a family of instances $(I_n)_{n\in\mathbb{N}}$, such that the size of each I_n is $\mathcal{O}(n)$ and the size of $\tau_1(I_n)$ is at least 2^n.*
(4) *There is a publishing transducer τ_2 in PT(CQ, relation, normal) and a family of instances $(J_n)_{n\in\mathbb{N}}$, such that the size of each J_n is $\mathcal{O}(n)$ and the size of $\tau_2(J_n)$ is at least 2^{2^n}.*

Proof. The proof is referred to the Appendix. □

5.2 Decision Problems for Publishing Transducers

We now turn to the classical decision problems associated with transducers. We show that when the relational query language $\mathcal{L}$ is FO or beyond, all these problems are undecidable, but some of the problems become decidable when $\mathcal{L}$ is CQ.

Proposition 5.2. *The membership, emptiness, and equivalence problems are undecidable for PT($\mathcal{L}$, S, O) when $\mathcal{L}$ is FO or IFP, no matter whether S is relation or tuple, and O is virtual or normal.*

Proof. We show that these problems are already undecidable for nonrecursive transducers in PT$_{nr}$(FO, tuple, normal). From this, the theorem immediately follows.

We show the undecidability by reduction from the *equivalence problem for relational FO queries*, which is to determine, given any FO queries Q_1, Q_2 on a relational schema R, whether or not for any instance I of R, $Q_1(I) = Q_2(I)$ (denoted by $Q_1 \equiv Q_2$). This problem is known to be undecidable [Abiteboul et al. 1995].

Given any FO queries Q_1, Q_2, we use ΔQ to denote their symmetric difference $(Q_1 \setminus Q_2) \cup (Q_2 \setminus Q_1)$. Obviously, $Q_1 \equiv Q_2$ if and only if $\Delta Q(I) = \emptyset$ for all instances I of R.

The membership problem. The reduction consists of a transducer τ_0 in PT$_{nr}$(FO, tuple, normal) and a tree t_0 such that $t_0 \in \tau_0(R)$ if and only if $Q_1 \not\equiv Q_2$. We define t_0 to be $r(a)$ (i.e., a root r with a single a-child), and $\tau_0 = (Q_0, \Sigma_0, \Theta_0, q_0, \delta_0)$, where $Q_0 = \{q_0, q\}$, $\Sigma_0 = \{r, a\}$, and δ_0 is given as follows, from which Θ_0 is clear:

$\delta_0(q_0, r) = (q, a, \phi(x;\emptyset))$, where $\phi(x;\emptyset) = \exists \bar{s} \Delta Q(\bar{s}) \wedge x =$ ‘c’ and $|\bar{s}|$ is the same as the arity of the result of Q_1 and Q_2;

$\delta_0(q, a) = (q, a, \phi_\emptyset(x;\emptyset))$, where $\phi_\emptyset(x;\emptyset) = (x =$ ‘c’$) \wedge \neg(x =$ ‘c’$)$, i.e., it is a query that returns the empty set on any database instance.

Then, obviously, if t_0 is in $\tau_0(R)$, that is there must exist an instance I of R such that $\phi(I)$ is nonempty. Hence $\Delta Q(I)$ is nonempty, that is, $Q_1 \not\equiv Q_2$. Conversely, if $Q_1 \not\equiv Q_2$, then there exists an instance I of R such that $\Delta Q(I)$ is nonempty. As a result, $\phi(I)$ yields a single tuple (c), and hence $t_0 \in \tau_0(R)$.

The emptiness problem. It suffices to define τ_1 in PT_{nr}(FO, tuple, normal) over R such that $\tau_1(R)$ consists of a single-node tree if and only if $Q_1 \equiv Q_2$. We define $\tau_1 = (Q_1, \Sigma_1, \Theta_1, q_0, \delta_1)$ to be the same as τ_0 except δ_1. Here, we define $\delta_1(q_0, r)$ = $(q, a, \phi(\bar{x}; \emptyset))$, where $\phi(\bar{x}; \emptyset) = \Delta Q(\bar{x})$, and $\delta_1(q, a)$ to be the same as $\delta_0(q, a)$. Then, obviously, $\tau_1(R) = \{r\}$ if and only if $\Delta Q(I) = \emptyset$ for all instances I of R, that is, $Q_1 \equiv Q_2$.

The equivalence problem. Given Q_1, Q_2, we construct τ_2^1, τ_2^2 in PT_{nr}(FO, tuple, normal) over R such that, for any instance I of R, $\tau_2^1(I) = \tau_2^2(I)$ if and only if $Q_1 \equiv Q_2$.

For $i \in [1, 2]$ we define τ_2^i to be the same as τ_0 except δ_2^i, given as follows:

$$\begin{array}{lll} \delta_2^i(q_0, r) & = (q, a, \phi(\bar{x}; \emptyset)), & \text{where } \phi(\bar{x}; \emptyset) = Q_i(\bar{x}); \\ \delta_2^i(q, a) & = (q, \mathit{text}, \phi_1(\bar{x}; \emptyset)), & \text{where } \phi_1(\bar{x}; \emptyset) = \mathit{Reg}_a(\bar{x}). \end{array}$$

Obviously, for each instance I of R, $\tau_2^1(I) = \tau_2^2(I)$ if and only if $Q_1(I) = Q_2(I)$. This is because, for each tuple in $Q_i(I)$, a distinct a child is created under the root r, which carries the tuple in its register, and the value of the tuple is given in the *text*-node child of the a-node. Thus $\tau_2^1 \equiv \tau_2^2$ if and only if $Q_1 \equiv Q_2$. □

The situation gets slightly better when considering conjunctive queries.

THEOREM 5.3. *For PT(CQ, S, O),*

(1) *the emptiness problem is decidable in* PTIME *for PT(CQ, S, normal), but it becomes* NP-*complete for PT(CQ, S, virtual);*
(2) *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;*
(3) *the equivalence problem is undecidable.*

PROOF. The proof is rather lenghty. In particular, for the membership problem, we provide three proofs: one for the Σ_2^p-completeness of PT(CQ, tuple, normal), and two separate proofs for the undecidability of PT(CQ, tuple, virtual) and PT(CQ, relation, normal); as will be shown by Proposition 6.3, the latter two classes are incomparable and thus require different treatments. We assume the presence of two distinct constants 0 and 1 in the domain $\mathbf{D}$ of data values. Most proofs remain intact in the absence of $\neq$ in CQ and, therefore, so do their corresponding results.

(1) *The emptiness problem.*

PT(CQ, S, normal). We first provide a *quadratic time* algorithm for testing emptiness for PT(CQ, S, normal), regardless of whether S is relation or tuple. For each τ in PT(CQ, S, normal) defined on a relational schema R, consider the start rule $(q_0, r) \rightarrow (q_1, a_1, \phi_1(\bar{x}_1; \bar{y}_1)), \ldots, (q_k, a_k, \phi_k(\bar{x}_k; \bar{y}_k))$. It is obvious that $\tau(R)$ contains a nontrivial tree if and only if one of the ϕ_is is satisfiable, since we only have normal nodes. The latter can be determined syntactically by first finding the equivalence class of each variable and constant involved in each ϕ_i, based on the equalities in ϕ_i, and then checking within each equivalence class to determine whether it contains (i) two distinct constants, (ii) a constant c and variable x for which $x \neq c$ is in θ, or (iii) two variables x and y for which $x \neq y$ is in θ. One can show that none of these cases occurs if and only if θ, and therefore, ϕ_i, are satisfiable. This can be done in $O(|\phi_1|^2 + \cdots + |\phi_k|^2)$ time.

PT(CQ, S, virtual). It suffices to show that the emptiness problem for PT(CQ, tuple, virtual) is NP-hard and that it is in NP for PT(CQ, relation, virtual).

Lower bound. We show the NP lower bound by reduction from the 3SAT-problem, an NP-complete problem [Papadimitriou 1994]. An instance of 3SAT is a well-formed Boolean formula $\varphi = C_1 \wedge \cdots \wedge C_n$, in which the variables are $X = \{x_1, \ldots, x_m\}$, and each clause C_i, for $i \in [1, n]$, is of the form $\ell_1^i \vee \ell_2^i \vee \ell_3^i$, where ℓ_j^i is either $x_s \in X$ or $\overline{x}_s$. Given such a φ, 3SAT is to determine the satisfiability of φ.

Given φ, we define a relational schema R and a transducer τ_φ in PT(CQ, tuple, virtual) over R such that φ is satisfiable if and only if there exists an instance I of R such that $\tau_\varphi(I)$ is nonempty. More specifically, R consists of an m-ary relation $R_X(A_1, \ldots, A_m)$. An instance I_X of R_X is to encode truth assignments of the variables in φ. The transducer $\tau_\varphi = (Q_0, \Sigma_0, \Theta_0, q_0, \delta_0, \Sigma_e)$, where $Q_0 = \{q_0, q_1, \ldots, q_n, q_t\}$, $\Sigma_0 = \{r, a, v\}$, and *virtual tag* $\Sigma_e = \{v\}$. The rules in δ_0 are as follows.

$$(q_0, r) \rightarrow (q_1, v, \psi_0(\bar{x}; \emptyset) = R(\bar{x})).$$
$$(q_{i-1}, v) \rightarrow (q_i, v, \psi_i^1(\bar{x}; \emptyset) = Reg_v(\bar{x}) \wedge (x_j = t_1^i[1] \wedge x_k = t_1^i[2] \wedge x_\ell = t_1^i[3])),$$
$$\ldots, (q_i, v, \psi_i^s(\bar{x}; \emptyset) = Reg_v(\bar{x}) \wedge (x_j = t_s^i[1] \wedge x_k = t_s^i[2] \wedge x_\ell = t_s^i[3])).$$
$$(q_n, v) \rightarrow (q_t, a, \psi_t = Reg_v(\bar{x})).$$

Here, $\bar{x}$ denotes $(x_1, \ldots, x_m)$. The rule for (q_0, r) copies I_X to the register of a v-child of the root r. For $i \in [1, n]$, the rule for (q_{i-1}, v) generates a virtual node if and only if the register of the current node u is a truth assignment that makes C_i true. For instance, suppose that $C_i = x_j \vee \bar{x}_k \vee x_\ell$, and denote by $T(C_i)$ the set of truth assignments of x_j, x_k, and x_ℓ that make C_i true, that is, $T(C_i) = \{(1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1), (0, 0, 1), (0, 1, 1), (0, 0, 0)\}$. Note that there are at most 8 tuples $T(C_i)$, denoted by $t_1^i, \ldots, t_s^i$. Then, ψ_j^i checks whether t_j^i is in I_X. If $Reg(u)$ is such a truth assignment, then the rule spawns a v-child of u and copies the content of $Reg(v)$ to $Reg(u)$. Hence (q_n, v) is reached if and only if I_X is a truth assignment that satisfies all n clauses of φ. The rule for (q_n, v) is defined with a normal tag a. It is easy to see that there exists an instance I_X of R_X such that $\tau_\varphi(I_X)$ is a nontrivial tree if and only if I_X contains a truth assignment satisfying φ.

Upper bound. We provide a NP algorithm for deciding the emptiness of transducers τ in PT(CQ, relation, virtual). To do this, we make use of the dependency graph G_τ of τ (recall from Section 3). Let ρ be a simple path in G_τ and $n = |\rho|$. Consider $Q^n = Q_n \circ \cdots \circ Q_1$, the CQ query obtained by composing the CQ queries encountered along the path ρ. Then, τ can produce a nontrivial tree if and only if Q^n is satisfiable for one of such paths ρ. Although $|Q^n|$ can be of exponential size, we show subsequently that the satisfiability of Q^n can be decided in PTIME by using n new CQ queries $\bar{Q}_i$, each of them of size polynomial in $|\tau|$, such that Q^n is satisfiable if and only if $\bar{Q}_i$ and Q_i are satisfiable for all $i \in [1, n]$. Based on this, given τ, the decision algorithm (i) guesses a simple path ρ in G_τ that leads to a nonvirtual node; (ii) constructs n queries $\bar{Q}_i$; and (iii) checks whether all $\bar{Q}_i$ and Q_i are satisfiable and, if so, it concludes that τ can produce a nontrivial tree. This clearly results in an NP-algorithm, provided that the last two steps can be done in PTIME, which we show next.

Let $\mathcal{S}_i$ be the relation schema of Q_i. We assume without loss of generality, that $Q_n(\bar{x}) = \exists \bar{x}_1 \cdots \bar{x}_k \bigwedge_{i=1}^{k_1} R_1(\bar{x}_i) \wedge \bigwedge_{j=k_1+1}^{k} \alpha_j(\bar{x}_i) \wedge H_n(\bar{x}) \wedge C_n(\bar{x}, \bar{x}_1, \ldots, \bar{x}_k)$, where $\bar{x}, \bar{x}_i$, for $i \in [1, k]$, consist of disjoint variables, $\alpha_j(\bar{x}_j) = R_k(\bar{x}_j)$ for some R_k $(k > 1)$ in $\mathcal{S}_n$, H_n denotes the conjunction of all (in-)equality constraints on variables in $\bar{x}$, and C_n denotes the conjunction of the remaining constraints. Moreover, assume that Q^n is obtained by substituting $Q^{n-1}(\bar{x}_i)$ for $R_1(\bar{x}_i)$ in Q_n, for $i \in [1, k_1]$. We construct the CQ query $\bar{Q}_n(\bar{x})$ and the conjunction of constraints $\bar{H}_n(\bar{x})$ (whose purpose is explained below) inductively. For $n = 1$, we let $\bar{Q}_1(\bar{x}) = Q_1(\bar{x})$ and let $\bar{H}_1(\bar{x})$ be the *completion* of $H_1(\bar{x})$ with respect to C_1. That is, we complete $H_1(\bar{x})$ with all (in-)equalities on $\bar{x}$ *inferred* from H_1 and C_1. For $n > 1$, we define $\bar{Q}^n(\bar{x}) = \exists \bar{x}_1 \cdots \bar{x}_k \bigwedge_{i=1}^{k_1} (R_1(\bar{x}_i) \wedge \bar{H}_{n-1}(\bar{x}_i)) \wedge \bigwedge_{j=k_1+1}^{k} \alpha_j(\bar{x}_i) \wedge H_n(\bar{x}) \wedge C_n(\bar{x}, \bar{x}_1, \ldots, \bar{x}_k)$ and let $\bar{H}_n(\bar{x})$ be the completion of $H_n(\bar{x}_i)$ with respect to C_n and $\bar{H}_{n-1}(\bar{x}_i)$, for $i \in [1, k_1]$. Then we have the following.

CLAIM 1. *Q^n is satisfiable if and only if $\bar{Q}^n$ and Q^{n-1} are satisfiable.*

PROOF. We show the claim by induction on n. Suppose that Q^n is satisfiable. We verify (in the induction) the following additional property, denoted by (a_n): for any relation I and tuple $\bar{t}$, if $\bar{t} \in Q^n(I)$, then $\bar{H}_n(\bar{t})$ holds. The case $n = 1$ is trivial. Let $n > 1$. Note that Q^n is obtained from Q_n by replacing $R_1(\bar{x}_i)$ by $Q^{n-1}(\bar{x}_i)$. By the induction hypothesis (a_{n-1}), we can obtain an equivalent query by replacing $R_1(\bar{x}_i)$ by $Q^{n-1}(\bar{x}_i) \wedge \bar{H}_{n-1}(\bar{x}_i)$. Hence any $\bar{t} \in Q^n(I)$ satisfies all constraints inferred by $H_n(\bar{x})$, $\bar{H}_{n-1}(\bar{x}_i)$ and C_n, and therefore $\bar{H}_n(\bar{t})$ holds; thus (a_n) holds. Let I be an instance such that $Q^n(I) \neq \emptyset$. Then clearly Q^i is satisfiable for all $i \in [1, n-1]$. Let $J_1 = Q^{n-1}(I)$ be the instance of R_1 in Q_n, and $J_i = I_i$ be the instances of R_i $(i > 1)$ in $\mathcal{S}_n$ (as given by I). By (a_{n-1}) we have that $\bar{Q}^n(J) \neq \emptyset$, as desired.

Conversely, we show another property, denoted by (b_n): if Q_n and Q^{n-1} are satisfiable and $\bar{H}_n(\bar{t}_i)$ holds for $i \in [1, \ell]$, then there exists an instance I such that $\bar{t}_i \in Q^n(I)$ for all $i \in [1, \ell]$. The case $n = 1$ is trivial. Let $n > 1$. Since Q_n is satisfiable, for each $\bar{t}_i$ that satisfies $\bar{H}_n(\bar{t}_i)$, we can find k source tuples $\bar{s}_1^i, \ldots, \bar{s}_k^i$ such that $C_n(\bar{t}_i, \bar{s}_1^i, \ldots, \bar{s}_k^i)$ holds, and moreover, $\bar{H}_{n-1}(\bar{s}_1^i)$ holds for $i \in [1, k_1]$ by the definition of $\bar{H}_n$. Since Q^{n-1} is satisfiable, so are Q_{n-1} and Q^{n-2}. Hence, by (b_{n-1}), we obtain an instance J_1 such that $\{s_j^i \mid j \in [1, k_1], i \in [1, \ell]\} \subseteq Q^{n-1}(J_1)$. Let $J_i = \{s_i^j \mid j \in [1, \ell]\}$ and $J = (J_1, \ldots, J_k)$. Then, by the monotonicity of CQ queries, $\bar{t}_i \in Q^n(J)$ for $i \in [1, \ell]$; thus (b_n) holds. Note that if $\bar{Q}_n$ is satisfiable, then so is Q_n, and there exist $\bar{t}_i$s satisfying $\bar{H}_n$; thus, by (b_n), Q^n is satisfiable. □

Note that $\bar{H}_i$ is at most of quadratic size in the number of variables in the head of Q_i, and thus $|\bar{Q}_i|$ is bounded by $O(|Q_i|^2)$. Then from the proof of Theorem 5.3 (1), it follows that the satisfiability of $\bar{Q}_i$ can be decided in PTIME.

(2) *The membership problem.*

PT(CQ, tuple, normal). We first show that the problem is Σ_2^p-hard, and then give a Σ_2^p-algorithm for deciding the membership of PT(CQ, tuple, normal).
Lower bound. We show the Σ_2^p lower bound by reduction from the $\exists^*\forall^*$-3SAT problem, which is known to be Σ_2^p-complete [Papadimitriou 1994]. The latter problem is to determine, given $\varphi = \exists Y \forall Z\ C_1 \wedge \cdots \wedge C_r$, whether or not φ evaluates to true. Here, $Y = \{y_1, \ldots, y_n\}$ and $Z = \{z_1, \ldots, z_k\}$, and $\exists Y$ is a shorthand for $\exists y_1 \ldots \exists y_n$; similarly for $\forall Z$. The clauses $C_1 \wedge \cdots \wedge C_r$ are an instance of 3SAT, given previously, in which each literal is either a variable in $Y \cup Z$ or a negation thereof.

Given φ, we define a relational schema R, a transducer τ_φ in PT(CQ, tuple, normal), and a tree t_φ such that $t_\varphi \in \tau_\varphi(R)$ if and only if φ is true.

(a) The relational schema R consists of a unary relation $R_C(B)$ and a ternary relations R_{OR}. We shall use the instance $I_C = \{0, 1\}$ of R_C to construct the Cartesian product $I_Y = \times_{i \in [1,n]} I_C$ to encode the existential quantification: there exists a tuple $t_Y \in I_Y$, that is, a truth assignment for Y, such that t_Y satisfy $\psi = \forall Z\ C_1 \wedge \cdots \wedge C_r$. An instance of I_{OR} of R_{OR} consists of $\{(0,0,0), (1,0,1), (0,1,1), (1,1,1)\}$ and encodes disjunction. This is needed, since CQ does not allow disjunction.

(b) To define τ_φ, observe that $\varphi \equiv \exists Y\, \varphi_1 \wedge \cdots \wedge \varphi_r$, where $\varphi_j = \forall z_1 \cdots \forall z_k\ C_j$, for $j \in [1, y]$. Hence, given a truth assignment for the variables in Y, it suffices to test whether all φ_js are true. We express $\varphi_1 \wedge \cdots \wedge \varphi_r$ as a CQ query $\psi(Y)$, as follows. For each $j \in [1, r]$, denote by l the number of universally quantified variables in C_j. For each binary vector $\bar{b}$ of length l, let $\psi_j^{\bar{b}}(Y) = \exists x_1, x_2, x_3, s\ R_{\mathsf{OR}}(x_1, x_2, s) \wedge R_{\mathsf{OR}}(s, x_3, 1) \wedge \bigwedge_{i=1}^{3} \theta_i^j(x_i)$, where $\theta_i^j(x_i) = y_p$ (resp. $\theta_i^j(x_i) \neq y_p$) if the ith literal in C_j is $y_p \in Y$ (resp. $\bar{y}_p$), and $\theta_i^j(x_i) = b[i]$ otherwise. Let $\psi_j = \bigwedge_{\bar{b}} Q_j^{\bar{b}}$, where $\bar{b}$ ranges over all possible truth assignments of the universally quantified variables in C_j. Since $l \leq 3$, there are at most eight such assignments. Now we define $\psi(Y) = \bigwedge_{j=1}^{r} Q_j$. It is easily verified that $\psi(Y)$ is satisfiable if and only if φ is true.

We now define the tree t_φ to be $r(b, d)$ (a root node with a single b and a single d child), and define the transducer τ_φ, for which the start rule is:

$(q_0, r) \rightarrow (q_1, b, \phi_1(x; \emptyset)), (q_1, c, \phi_2(x; \emptyset)), (q_1, d, \phi_3(x; \emptyset))$, where

$$\phi_1(x; \emptyset) \equiv R_C(0) \wedge R_C(1) \wedge R_{\mathsf{OR}}(0,0,0) \wedge R_{\mathsf{OR}}(1,0,1) \wedge R_{\mathsf{OR}}(0,1,1) \wedge R_{\mathsf{OR}}(1,1,1) \wedge x = 1$$
$$\phi_2(x; \emptyset) \equiv R_C(x) \wedge x \neq 0 \wedge x \neq 1, \qquad \phi_3(x; \emptyset) \equiv \exists Y \left(\bigwedge_{j=1}^{n} R_C(y_j) \wedge \psi(Y)\right) \wedge x = 1.$$

In the rules for (q_1, b), (q_1, c), and (q_1, d), the RHS is empty. Intuitively, $\phi_1(x; \emptyset)$ assures that 0 and 1 are in the instance of R_C, and that I_{OR} is contained in the instance of R_{OR} (which is not necessarily I_{OR}). The formula $\phi_2(x; \emptyset)$ checks whether instances of R_C have Boolean values only. By not including a c node in t_φ, ϕ_2 and ϕ_1 assure that any instance of R_C is precisely $\{0, 1\}$. The formula $\phi_3(x; \emptyset)$ computes all truth assignments of Y, and checks whether any of these satisfies $\psi(Y)$. It is easy to verify that φ is true if and only if there exists an instance I of R such that $\tau_\varphi(I) = t_\varphi$.

Upper bound. We next provide a Σ_2^p algorithm that, given a transducer τ in PT(CQ, tuple, normal) and a tree t, guesses an instance I and then verifies, using an NP-oracle, whether $\tau(I) = t$. A crucial observation is that it suffices to guess an instance I of polynomial size, by the following small model property:

CLAIM 2. *Let τ be a transducer in PT(CQ, tuple, normal) over R, and let I be an instance of R such that $t = \tau(I)$. Then, there exists an instance $I' \subseteq I$ such that (i) $t = \tau(I')$; and (ii) $|I'|$ is of size at most $K|t|$, where K is the maximal number of Cartesian products in any of the CQ queries in τ.*

PROOF. Assume that there exists an instance I such that $t = \tau(I)$. To simplify the discussion, assume furthermore that relational schema consists of a single relation schema R. In case the schema consists of more than one relation, one can simulate these with a single relation, and change the CQ queries in τ accordingly, and obtain in this way an equivalent transducer, albeit on a different schema.

Let v be an arbitrary node in t, and $(q, a) \rightarrow (q_1, a_1, \phi_1(\bar{x}_1; \emptyset)), \ldots, (q_k, a_k, \phi_k(\bar{x}_k; \emptyset))$ be the rule in τ that is used to generate v. More specifically, assume that v is generated by ϕ_i and hence has label a_i. Since τ has a tuple store, each v is associated with a distinct tuple (stored in $Reg_{a_i}(v)$) from the result of the CQ query ϕ_i on I and $Reg_a(u)$, where u is the parent of v and $Reg_a(u)$ is the tuple register of u.

If we express ϕ_i as an SPC (selection, projection and Cartesian product) query in the normal form, it is clear that $Reg_{a_i}(v)$ comes from the Cartesian product of at most k tuples in I, where k is the number of R's (perhaps renamed) involved in ϕ_i, and thus is determined by the size of ϕ_i. Let us refer to these k tuples as the *source tuples* for v. Putting together all the source tuples for all the nodes in t, we get another instance I' of R. One can verify that $t = \tau(I')$, since all the queries in τ are CQ queries with '$\neq$' and are thus monotonic. Clearly, $|I'| \leq K|t|$, where K is the maximal number of Cartesian products in any of the CQ queries in τ. □

Given a transducer τ in PT(CQ, tuple, normal) and a tree t, the following algorithm checks whether there exists an instance I such that $t = \tau(I)$:

(1) Guess an instance I consisting of at most $K|t|$ tuples (here K is as in the statement of Claim 2). The active domain $\mathbf{U}$ of I consists of the constants appearing in any CQ query embedded in the rules of τ plus a set of $K|t|$ other arbitrary constants. It is easily verified that for any other such domain $\mathbf{U}'$ and bijective mapping $f : \mathbf{U} \rightarrow \mathbf{U}'$ such that $f(a) = a$ for any constant a appearing in queries in τ, $\tau(I) = \tau(f(I))$. Hence we can choose $\mathbf{U}$ arbitrarily.

(2) We then guess $|t|$ tuples, one for the register of each distinct node in t.

(3) Given I and the tree t annotated with the registers, we then use the following NP-oracle for testing whether $\tau(I) = t$ as follows. We traverse the tree t top-down. For each node v encountered, let the rule generating v

be $(q, a) \rightarrow (q_1, a_1, \phi_1(\bar{x}; \emptyset)), \ldots, (q_k, a_k, \phi_k(\bar{x}; \emptyset))$, which is unique as τ is deterministic.

(a) Let $[v_1, \ldots, v_\ell]$ be the list of *children*(v). Starting from $i = 1$ and $j = 1$, we check whether $lab(v_i) = a_j$, and whether $Reg_{lab(v_i)}(v_i)$ is in the query result $\phi_j(I)$. This can be done in NP. If either fails, we put $j = j + 1$ and continue. Otherwise, we move on to the next child, that is, we put $i = i + 1$ and continue. If we reach $i = \ell + 1$ and $j = k + 1$, then all children of v can be generated by the τ on I, and move on to step (b). Otherwise, if for some v_i with $i \leq \ell$ we reach $j = k + 1$, then we reject the guess.

(b) For $i \in [1, k]$, we check whether there are more tuples in $\phi_i(I)$ than those already identified in the previous step. This can be verified in NP as well. If the answer is negative, we accept I, otherwise we reject the input I.

This algorithm can be simulated using a nondeterministic polynomial time Turing machine with a NP-oracle. Hence this algorithm is in $\text{NP}^{\text{NP}} = \Sigma_2^p$.

PT(*CQ, tuple, virtual*). We show that the membership problem becomes undecidable in the presence of virtual nodes, by reduction from the emptiness problem of deterministic finite two-head automata, which is undecidable [Spielmann 2000]. Our reduction follows closely the reduction presented in [Spielmann 2000, Theorem 3.3.1], which shows that the satisfiability of the existential fragment of transitive-closure logic, E+TC, is undecidable over a schema having at least two non-nullary relation schemas, one of them being a function symbol. Although E+TC allows the negation of atomic expression in contrast to CQ, the undecidability proof only uses a very restricted form of negation, which we can simulate in PT(CQ, tuple, virtual).

For the reader's convenience we include the necessary definitions taken from Spielmann [2000]. A *deterministic finite two-head automaton* (or two-head DFA for short) is a quintuple $\mathcal{A} = (Q, \Sigma, \Delta, q_0, q_{acc})$ consisting of a finite set of states Q, an input alphabet $\Sigma = \{0, 1\}$, an initial state q_0, an accepting state q_{acc}, and a transition function $\Delta : Q \times \Sigma_\varepsilon \times \Sigma_\varepsilon \rightarrow Q \times \{0, +1\} \times \{0, +1\}$, where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

A *configuration of* $\mathcal{A}$ is a triple $(q, w_1, w_2) \in Q \times \Sigma^* \times \Sigma^*$, representing that $\mathcal{A}$ is in state q and the first and second head of $\mathcal{A}$ are positioned on the first symbol of w_1 and w_2, respectively. On an input string $w \in \Sigma^*$, $\mathcal{A}$ starts from the initial configuration (q_0, w, w); the successor configuration is defined as usual. The two-head DFA $\mathcal{A}$ *accepts* w if it can reach a configuration (q_{acc}, w_1, w_2) from the initial configuration for w; otherwise, $\mathcal{A}$ rejects w. The *language accepted by* $\mathcal{A}$ is denoted by $L(\mathcal{A})$. The emptiness problem for two-head DFA's is to determine, given a two-head DFA $\mathcal{A}$, whether $L(\mathcal{A})$ is empty or not.

Given a two-head DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, q_{acc})$, we define a schema R, a transducer $\tau_{\mathcal{A}}$ in PT(CQ, tuple, virtual) and a tree $t_{\mathcal{A}}$ such that $t_{\mathcal{A}} \in \tau_{\mathcal{A}}(R)$ if and only if $L(\mathcal{A})$ is nonempty.

(a) The relational schema R consists of three relations: (i) two unary relations $P(A)$ and $\bar{P}(A)$ and (ii) a binary relation $F(A_1, A_2)$. Intuitively, an instance

$I = (I_P, I_{\bar{P}}, I_F)$ of R is to represent a string w such that elements in P represent the positions in w, where an 1 occurs; similarly, $\bar{P}$ holds those positions where w equals 0. The relation F encodes a successor relation over these positions.

As before, we shall use transduction rules in $\tau_{\mathcal{A}}$ and the tree $t_{\mathcal{A}}$ to assure that we only consider well-formed instances of P, $\bar{P}$ and F. That is, (i) instances I_P and $I_{\bar{P}}$ of P and $\bar{P}$ are disjoint; and any instance I_F of F must (ii) be a function, (iii) contain a tuple of the form $(0, i)$, where 0 represents the initial position and i is some constant, and (iv) contain a unique tuple of the form (k, k) for some constant k indicating the final position.

(b) We define the transducer $\tau_{\mathcal{A}}$ and tree $t_{\mathcal{A}}$ as follows. First, to assure that instances of P and $\bar{P}$ are disjoint, we add the rule $(q_0, r) \to (q, a_1, \phi_1 = \exists x P(x) \wedge \overline{P}(x))$ and by not including an a-child of the root in $t_{\mathcal{A}}$. Second, we assure the properties (ii)–(iv) on instances I_F of F. This is achieved by adding $(q, a_2, \phi_2 = \exists y F(0, y))$, $(q, a_3, \phi_3(x, y; \emptyset) = F(x, y) \wedge x = y)$ and $(q, a_4, \phi_4 = \exists x, y, z F(x, y) \wedge F(x, z) \wedge y \neq z)$ to the RHS of the start rule given above, and by adding a single a_2 and a_3-child to the root of t_1. We do not include an a_4-child of the root to $t_{\mathcal{A}}$. Then, for $\tau_{\mathcal{A}}$ and $t_{\mathcal{A}}$ defined so far, $\tau_{\mathcal{A}}(I) = t_{\mathcal{A}}$ if and only if I is a well-formed instance of R.

Before we continue with the definition of $\tau_{\mathcal{A}}$ and $t_{\mathcal{A}}$ we show, following Spielmann [2000], how nonemptiness of $L(\mathcal{A})$ can be expressed in terms of an E+TC-formula over R. Consider a transition $\delta \in \Delta$ of the form $\delta = (q, \mathsf{in}_1, \mathsf{in}_2) \to (q', \mathsf{move}_1, \mathsf{move}_2)$. This can be encoded by means of the conjunctive query

$$\varphi_\delta(x, y, z, x', y', z') = (x = q \wedge x' = q' \wedge \alpha_1(y) \wedge \alpha_2(z) \wedge \beta_1(y, y') \wedge \beta_2(z, z')),$$

where $\alpha_i(x) = \exists y F(x, y) \wedge x \neq y \wedge P(x)$ if $\mathsf{in_i} = 1$; $\alpha_i(x) = \exists y F(x, y) \wedge x \neq y \wedge \overline{P}(x)$ if $\mathsf{in_i} = 0$; and $\alpha_i(x) = F(x, x)$ if $\mathsf{in_i} = \varepsilon$. Moreover, $\beta_i(x, y) = F(x, y)$ if $\mathsf{move_i} = +1$ and $\beta_i(x, y) = x = y$ if $\mathsf{move_i} = 0$. Intuitively, $\alpha_i(x)$ enforces x to be a position in the string coded by P and $\bar{P}$ that has a successor, unless x is the final position where $\alpha_i(x)$ demands $F(x, x)$. Moreover, $\beta_i(x, y)$ ensures that x and y are consecutive positions when $\mathcal{A}$ makes a move (with head i), and $x = y$ otherwise. Then, $\Phi = \exists y_1 \exists y_2 [\mathsf{TC}_{x,y,z;x',y',z'} \bigvee_{\delta \in \Delta} \varphi_\delta](q_0, 0, 0, q_{acc}, y_1, y_2)$ is satisfiable if and only if $L(\mathcal{A}) \neq \emptyset$.

The transducer $\tau_{\mathcal{A}}$ simulates the transitive closure (TC) in Φ by means of virtual nodes, and an s-node in output if $(q_0, 0, 0, q_{acc}, y_1, y_2)$ is encountered during its evaluation on an instance I of R. By including an s-node as a child of the root of $t_{\mathcal{A}}$, we can then test whether Φ is satisfiable, or equivalently, whether $L(\mathcal{A}) \neq \emptyset$.

To do so, we first initialize the TC-computation by extending the RHS of the start rule of $\tau_{\mathcal{A}}$ with $(q, v, \kappa_0(q, x, y; \emptyset) = (q = q_0 \wedge x = 0 \wedge y = 0))$, where v is a virtual tag. Suppose that $\Delta = \{\delta_1, \ldots, \delta_m\}$. For each $\delta_i \in \Delta$, we introduce a new state q_i in $\tau_{\mathcal{A}}$ and define $\kappa_i(q, x, y; \emptyset) = \exists q', x', y' Reg_v(q', x', y') \wedge \varphi_{\delta_i}(q', x', y', q, x, y)$. That is, κ_i encodes a valid transition (given by δ_i) in the TC-computation starting from the configuration stored in the current register. We simulate Φ by simultaneously executing all valid transitions. For each $i \in [1, m]$, we define the rule:

$$(q_i, v) \to (q_1, v, \kappa_1(q, x, y; \emptyset)), \ldots, (q_m, v, \kappa_m(q, x, y; \emptyset)), (q, s, \phi_f = \exists x, y Reg_v(q_{acc}, x, y)).$$

As mentioned, these rules create a (nonvirtual) node s if and only if the final configuration is encountered. We remark that, since $\mathcal{A}$ is deterministic, if the final configuration is encountered (and hence Φ is satisfiable), then this only happens once. Hence $\tau_{\mathcal{A}}$ creates a single s-node as a child of the root in its output tree. Thus the inclusion of a single s-child of the root in $t_{\mathcal{A}}$ indicates whether Φ is satisfiable.

Taken together, it is easily verified that $t_{\mathcal{A}} \in \tau_{\mathcal{A}}(R)$ if and only if $L(\mathcal{A}) \neq \emptyset$. As a result, the membership problem for PT(CQ, tuple, virtual) is undecidable.

PT(CQ, relation, normal). We show that the problem also becomes undecidable with relation registers, but without virtual nodes, by reduction from the satisfiability problem for relational algebra, which is undecidable [Abiteboul et al. 1995].

Given a relational algebra query Q over a schema S, we define a relational schema R, a transducer τ_Q over R in PT(CQ, relation, normal), and a tree t_Q such that $t_Q \in \tau_Q(R)$ if and only if Q is satisfiable. We define a nonrecursive τ_Q: it generates trees of depth bounded by the size of Q. Observe that, although PT(CQ, relation, O) is incapable of expressing FO queries, as will be seen in Section 6, its membership analysis can validate FO-query evaluation, as shown by the subsequent proof.

The construction of τ_Q and t_Q is based on a parse-tree $\mathsf{parse}(Q)$ of Q, which is a node-labeled tree in which each interior node is labeled with a relational operator, that is, projection (π_X), selection ($\sigma_{A=B}$), renaming ($\rho_{A/B}$), Cartesian product ($\times$), union ($\cup$), or difference ($\setminus$). The leaves of $\mathsf{parse}(Q)$ are the base relations in S. Intuitively, each node in $\mathsf{parse}(Q)$ indicates a subquery Q' of Q.

(a) The schema R consist of all base relations of S and, in addition, for each subquery Q' of Q, as given by $\mathsf{parse}(Q)$, a relation schema $R_{Q'}$. The intuition is that in any instance I of R, the instance I_S of S stores the base relations, and the instance $I_{Q'}$ of $R_{Q'}$ encodes the query result $Q'(I_S)$. To simplify the handling of subqueries that return an empty set, we add a special attribute A to each relation of R. As before, we say $I_{Q'}$ is well-formed if (i) it contains the tuple $t_0 = (0, 0, \ldots, 0)$, that is, it is nonempty; (ii) all other tuples have their A-attribute set to 1; and (iii) the set of tuples with their A-attribute set to 1 is precisely $Q'(I_S)$ (after their A-attributes are stripped off). That is, a well-formed $I_{Q'}$ is equal to $(\{1\} \times Q'(I_S)) \cup \{(0, 0, \ldots, 0)\}$. We use $\mathsf{att}(R_1)$ to denote the set of attributes in a relation schema R_1 of R. The schema R also contains auxiliary relations for coding union and set difference, which will be introduced as they come along.

(b) The transducer τ_Q and tree t_Q are defined such that $\tau_Q(I) = t_Q$ if and only if I is well-formed and Q is satisfiable. The tree t_Q is used for two purposes: by including certain nodes, it requires some of the queries in the rules of τ_Q to return a non-empty answer; in contrast, excluding certain nodes in t either enforces the corresponding queries to return the empty set, or enforces a stop condition to hold.

We define τ_Q and t_Q simultaneously. For each rule in τ_Q presented in the following, nodes produced by the rule will be added to t_Q if they have a bold label; otherwise, the nodes will not appear in t_Q. We employ different labels in the rules such that each label a is mapped to a distinct node (i.e., a subquery) in $\mathsf{parse}(Q)$, denoted by $\mathsf{p}(a)$. The rule for nodes labeled a is determined by the structure of the subquery represented by the subtree rooted at $\mathsf{p}(a)$. We denote by y_A a variable coding attribute A, and by $\bar{x}$ a set of variables where the size of $\bar{x}$ will be clear from the context. To simplify the discussion, the queries in τ_Q are given in relational algebra, but can be easily written in CQ. We define τ_Q and t_Q top-down starting from the root r, inductively following a top-down traversal of $\mathsf{parse}(Q)$, as follows.

(*Case 0*): The start rule of τ_Q is $(q_0, r) \rightarrow (q, \mathbf{a_1}, \phi_1^0(\emptyset; y_A, \bar{x}) = R_Q(y_A, \bar{x})), (q, \mathbf{b_1}, \phi_2^0 = \pi_{\mathsf{att}(R_Q)} R_Q(1, \bar{x}))$. On instances I of R, this rule creates an a_1-node w_1 such that its relation register $Reg_{a_1}(w_1)$ stores $I_Q(I_s)$, that is, the final query result. Moreover, a b_1-node w_2 is created, provided that $I_Q(I_S)$ contains tuples with their A-attribute set to 1. For any well-formed instances, this implies that $Q(I_S) \neq \emptyset$. Note that both a_1 and b_1 are shown in bold, and hence t_Q is expanded by making w_1 and w_2 children of r. Clearly, the presence of w_2 in t_Q indicates nonemptiness of $Q(I_S)$.

We next give the rule for (q, a_i), where a_i corresponds to $\mathsf{p}(a_i)$ in $\mathsf{parse}(Q)$, which indicates a subquery Q' of Q. From the inductive construction, we have that, for any a_i-node v generated and included in t_Q, $Reg_{a_i}(v)$ stores the relation $R_{Q'}$. We define the rule for (q, a_i) and expand t_Q, based on the structure of Q'.

(1) $Q' = Q_1 \times Q_2$. We can express $R_{Q'}$ in terms of R_{Q_1} and R_{Q_2} by the query $Q_{prod} = \pi_{\mathsf{att}(Q_1 \times Q_2) - A'} \sigma_{A=A'}(R_{Q_1} \times \rho_{A/A'}(R_{Q_2}))$, where A is the special attribute mentioned previously. We define the rule for (q, a_i) to be $(q, a_i) \rightarrow (q, \mathbf{a_{i+1}}, \phi_1^i(\emptyset; y_A, \bar{x}) = R_{Q_1}(y_A, \bar{x})), (q, \mathbf{a_{i+2}}, \phi_2^i(\emptyset; y_A, \bar{x}) = R_{Q_2}(y_A, \bar{x})), (q, a_i, \phi_3^i(\emptyset; y_A, \bar{x}) = Q_{prod}(y_A, \bar{x}))$. The purpose of this rule is two-fold. First, given an instance I of R, the rule generates an a_{i+1}-child and an a_{i+2}-child of v, containing I_{Q_1} and I_{Q_2}, respectively, in their registers. These nodes are included in t_Q as indicated by their labels (bold). Second, it assures that if I_{Q_1} and I_{Q_2} are well-formed, then so is $I_{Q'}$. Indeed, it generates an a_i-node if and only if $I_{Q'} \neq Q_{prod}(I_{Q_1}, I_{Q_2})$, due to the stop condition. Further, v does not have an a_i-child in t_Q (note that a_i is not in bold). These ensure that $I_{Q'}$ is well-formed as long as I_{Q_1} and I_{Q_2} are. We omit the construction for the simple cases (2, 3, 4) where Q' is a selection, projection, or renaming subquery, due to space constraints.

(5) $Q' = Q_1 \cup Q_2$. This case is a bit tricky, since CQ does not allow disjunction. To cope with this, we employ an additional relation $R_{Q_1+Q_2}$ in R such that $I_{Q_1+Q_2} = \{0\} \times I_{Q_1} \cup \{1\} \times I_{Q_2}$. In terms of $R_{Q_1+Q_2}$, we can keep track of tuples in I_{Q_1}, I_{Q_2} and inspect their union. More specifically, we express $R_{Q'}$, R_{Q_1} and R_{Q_2} as $Q_+ = \pi_{\mathsf{att}(R_{Q_1})} R_{Q_1+Q_2}$, $Q_+^1 = \pi_{\mathsf{att}(R_{Q_1})} \sigma_{A'=0}(R_{Q_1+Q2})$, and $Q_+^2 = \pi_{\mathsf{att}(R_{Q_1})} \sigma_{A'=1}(R_{Q_1+Q_2})$, respectively, where A' is the first attribute in $R_{Q_1+Q_2}$ that holds tags 1 or 0. Then, as in case (1), we can assure the following in the rule for (q, a_i). First, $I_{Q'} = Q_+(I_{Q_1+Q_2})$, and $Q_+^1(I_{Q_1+Q_2}) = I_{Q_1}$ (resp. $Q_+^2(I_{Q_1+Q_2}) = I_{Q_2}$), by not including certain nodes in t_Q and leveraging the stop condition. Second, $I_{Q_1+Q_2}$ is well-formed as long as I_{Q_1} and I_{Q_2} are. We omit the details for the lack of space.

(6) $Q' = Q_1 \setminus Q_2$. As in case (5), since CQ does not allow negation, we use two auxiliary relations $R_{Q_1 \cap Q_2}$ and $R_{Q_1 | Q_2}$ in R. For an instance I of R, $I_{Q_1 \cap Q_2}$ is to store $I_{Q_1} \cap I_{Q_2}$, and $I_{Q_1|Q_2}$ is to store $\{0\} \times I_{Q_1 \cap Q_2} \cup \{1\} \times I_{Q'}$. Intuitively, we inspect set difference by checking whether $I_{Q'} \cap I_{Q_1 \cap Q_2} = \emptyset$ and $I_{Q'} \cup I_{Q_1 \cap Q_2} = I_{Q_1}$. To do this, in terms of $R_{Q_1|Q_2}$, we express $R_{Q'}$ and R_{Q_1} as $Q_{\mathsf{diff}} = \pi_{\mathsf{att}(R_{Q_1})} \sigma_{A'=1}(R_{Q_1|Q_2})$ and $Q^1 = \pi_{\mathsf{att}(R_{Q_1})} R_{Q_1|Q_2}$, respectively. Furthermore, we express $R_{Q_1 \cap Q_2}$ as both $Q^1_\cap = R_{Q_1} \cap R_{Q_2}$ and $Q^2_\cap = \pi_{\mathsf{att}(R_{Q_1})} \sigma_{A'=0}(R_{Q_1|Q_2})$. We also define $Q_\emptyset = \sigma_{A'=1}(R_{Q'} \cap R_{Q_1 \cap Q_2})$. Then, as in case (1), we can assure the following in the rule for (q, a_i). First, $I_{Q'} = Q_{\mathsf{diff}}(I_{Q_1|Q_2})$ and $I_{Q_1|Q_2}$ is precisely $\{0\} \times I_{Q_1 \cap Q_2} \cup \{1\} \times I_{Q'}$, by not including certain nodes in t_Q. Second, if I_{Q_1} and I_{Q_2} are well-formed, then so are $I_{Q_1 \cap Q_2}$ and $I_{Q_1|Q_2}$. We omit the details for lack of space.

(7) Q' is a base relation S_i in S. We need to verify that I_{S_i} contains $(0, 0, \ldots, 0)$ and all other tuples in I_{S_i} have their A-attribute set to 1. To do this, it suffices to use the rule $(q, a_i) \rightarrow (q, \mathbf{a_{i+1}}, \phi^i_1 = R_{S_i}(0, 0, \ldots, 0))$, $(q, \mathbf{a_{i+2}}, \phi^i_2(\emptyset; \bar{x}) = R_{S_i}(1, \bar{x}))$, $(q, \mathbf{d_i}, \phi^i_3(y_A; \emptyset) = \exists \bar{x}\ \pi_A R_{S_i}(y_A, \bar{x}))$. We include one a_{i+1}-node, one a_{i+2}-node, and two d_i-nodes to v in t_Q. This assures that I_{S_i} is well-formed.

We now show that $\tau_Q(I) = t_Q$ if and only if Q is satisfiable. First, suppose that Q is satisfiable, and let I_S be an instance of S such that $Q(I_S)$ is nonempty. Then, we obtain an instance I of R by letting $R_{Q'} = (\{1\} \times Q'(I_S)) \cup \{(0, 0, \ldots, 0)\}$ for any subquery Q' of Q given by $\mathsf{parse}(Q)$. Clearly, $\tau_Q(I) = t_Q$. Conversely, by the previous construction, we know that all instances I of R, such that $\tau_Q(I) = t_Q$, are necessarily well-formed. In particular, $\sigma_{A=1} R_Q$ holds the query result $Q(I_S)$, where I_S is the instance of base-relations S. Then, if $\tau_Q(I) = t_Q$, by the definition of the start rule of τ_Q, together with the presence of a b_1-child of the root in t_Q, Q is satisfiable.

(3) *The equivalence problem.* It suffices to show that the equivalence problem for PT(CQ, tuple, normal) is undecidable, by reduction from the halting problem for two-register machines (2RM) on the empty input string [Börger et al. 1997].

A two-register machine M has two registers, $\mathsf{register}_1, \mathsf{register}_2$, and is programmed by a numbered sequence, $I_0, I_1, \ldots, I_\ell$, of instructions. Each register contains a natural number. An *instantaneous description* (ID) of M is (i, m, n), where $i \in [0, \ell]$, m and n are natural numbers. It indicates that M is to execute instruction I_i (or is at "state i") with $\mathsf{register}_1$ and $\mathsf{register}_2$ containing m and n, respectively.

An instruction I_i of M is as follows, which defines a relation $\rightarrow_M$ between IDs.

(a) *addition* (i, rg, j): at state i, M adds 1 to the content of rg, and then goes to state j; for example, when $\mathsf{rg} = \mathsf{register}_1$ then $(i, m, n) \rightarrow_M (j, m+1, n)$.

(b) *subtraction* (i, rg, j, k): at state i, M tests whether rg is 0; if so, it goes to state j, otherwise subtracts 1 from rg and goes to the state k. When $\mathsf{rg} = \mathsf{register}_1$, $(i, m, n) \rightarrow_M (j, 0, n)$ if $m = 0$, and $(i, m, n) \rightarrow_M (k, m-1, n)$ otherwise.

Here, rg is either $\mathsf{register}_1$ or $\mathsf{register}_2$, and $0 \leq i, j, k \leq \ell$. Similarly, addition and subtraction are defined when $\mathsf{rg} = \mathsf{register}_2$.

Assume, without loss of generality, that the initial ID is $id_0 = (0, 0, 0)$, and that the final ID is $id_f = (f, 0, 0)$, that is, a halting state $f \in [0, \ell]$ with 0 in both registers. The *halting problem for 2RM* is to determine, given a 2RM M, whether or not $id_0 \Rightarrow_M id_f$, where $\Rightarrow_M$ is the reflexive and transitive closure of $\rightarrow_M$. A *valid run* of M is a sequence of IDs $id_0, id_1, \ldots$ such that, for each $i = 0, 1, \ldots$, we have that $id_i \rightarrow_M id_{i+1}$.

We give a reduction from the halting problem for 2RMs to the complement of the equivalence problem. Given a 2RM M, we construct a relational schema R and two transducers τ_1 and τ_2 over R in PT(CQ, normal, tuple) such that there exists an instance I of R such that $\tau_1(I) \neq \tau_2(I)$ if and only if M is halting.

(a) The schema R is a 6-ary relation with attributes prev (for previous), next (for next), cs (for current state), reg1 (for register 1), reg2 (for register 2), and ns (for next state). Intuitively, an instance I of R consists of tuples t, where t[cs, reg1, reg2] encodes an ID of M, t[ns] encodes the next state of M; and t[prev] and t[next] provide an ordering on the tuples in t. An instance I of R is said to be well-formed if prev determines next and vice versa, that is, for any $t_1, t_2 \in I$, if t_1[prev] $=$ t_2[prev] then t_1[next] $=$ t_2[next], and vice versa; if this holds, we say that prev is *a key* for next; similarly, next is a key for prev. Any well-formed instance I of R necessarily contains a unique sequence σ_I of tuples $t_0 = (0, a_1, \bar{c}_0)$, $t_1 = (a_1, a_2, \bar{c}_1)$, $\ldots$, $t_n(a_{n-1}, a_n, \bar{c}_n)$, $\ldots$, coding the numbers (the contents of the registers).

(b) We construct τ_1 and τ_2 over R such that, when applied to a well-formed instance I of R, they behave almost the same and both verify whether σ_I forms a valid run of M. At each step, both transducers spawn an a-node if the transition between the two consecutive tuples in σ_I is valid. If either σ_I does not form a valid run or σ_I forms a valid run but the halting state is not reached, both τ_1 and τ_2 simply stop. If σ_I is a valid run leading to a halting state, τ_1 and τ_2 exhibit a different behavior. More specifically, while τ_1 creates an extra a-node, τ_2 simply stops. Therefore, for any well-formed instance I of R, $\tau_1(I)$ and $\tau_2(I)$ will be the same tree, except for when M halts. Indeed, in the latter case $\tau_1(I)$ has one a-node more than $\tau_2(I)$.

To accommodate instances of R that are not well-formed, we modify τ_1 and τ_2. When a halting state is reached by τ_1 and τ_2, τ_1 generates an extra a-node (apart from the one it already created) if and only if neither prev is a key for next, nor is next a key for prev. In contrast, τ_2 will generate an a-node if prev is not a key for next, and another a-node if next is not a key for prev. This suffices. Indeed, consider the following three scenarios when a halting state is encountered: (i) prev is a key for next and vice versa; in this case, τ_1 generates a single a-node (because the halting state is encountered), while τ_2 does not generate anything (since both are keys); (ii) prev is a key for next, but not conversely (the symmetric case is analogous); then τ_1 will generate a single a-node, and so does τ_2; and finally (iii) prev is not a key for next and next is not a key for prev; in this case, both τ_1 and τ_2 generate two a-nodes. These are expressible as rules (with $\neq$ in particular) in PT(CQ, tuple, normal). Hence τ_1 and τ_2 only generate different trees on instances that are well-formed and that hold a halting sequence of moves of M, as desired.

We now define τ_1 and τ_2. The transducer τ_1 consists of the following rules:

$$\begin{aligned}(q,r) \;\rightarrow\; & (q_1, a, \phi_0(a_1, a_2, i, m, n, j; \emptyset) = R(a_1, a_2, i, m, n, j) \wedge a_1 = 0 \wedge i = 0 \wedge m = 0 \\ & \qquad \wedge n = 0 \wedge \exists z_1 z_2 z_3\ R(0, 0, j, z_1, z_2, z_3))\end{aligned}$$

$$\begin{aligned}(q_1, a) \rightarrow\ & \text{add to register}_1\text{, add to register}_2\text{, subtract from register}_1\text{, subtract from register}_2, \\ & (q_3, a, \phi_{halt} = \exists a_1, a_2, i, m, n, j\ Reg(a_1, a_2, i, m, n, j) \wedge i = f \wedge m = 0 \wedge n = 0 \\ & \qquad \wedge j = f), (q_4, a, \phi_{halt+nokeys} = \phi_{halt} \wedge \phi_{Pnokey} \wedge \phi_{Nnokey}),\end{aligned}$$

where $\phi_{Pnokey} = \exists a_1, a_2, b_1, b_2, \bar{x}, \bar{x}' R(a_1, a_2, \bar{x}) \wedge R(b_1, b_2, \bar{x}') \wedge a_1 = b_1 \wedge a_2 \neq b_2$, and similarly, $\phi_{Nnokey} = \exists a_1, a_2, b_1, b_2, \bar{x}, \bar{x}' R(a_1, a_2, \bar{x}) \wedge R(b_1, b_2, \bar{x}') \wedge a_2 = b_2 \wedge a_1 \neq b_1$.

The transducer τ_2 consists of the same set of rules, except that $(q_4, a, \phi_{halt+nokeys})$ is replaced by $(q_4, a, \phi_{Pnokey}), (q_4, a, \phi_{Nnokey})$.

We now explain how to simulate the addition and subtraction. The register contents of M are stored in the reg1 and reg2 attributes of R. The order induced by the key constraints on the prev and next-attributes of well-formed instances of R is used to increment and decrement the register contents. That is, assume that (a_1, a_2, i, m, n, j) is a tuple in an instance I of R. Suppose that register_1 needs to be incremented. Then, the next tuple should be of the form (a_2, a_3, j, m', n, k), where the new content m' of rg1 is defined such that there exists a tuple in I of the form $(m, m', \bar{x})$. Similarly, when register_1 needs to be decremented, m' is defined such that there is a tuple in I of the form $(m', m, \bar{x})$ (provided that $m \neq 0$).

Let A_1 (resp. D_1) be the set of states that correspond to additions (resp. subtractions) of register_1. We explain "add to register_1" in the rule for (q_1, a) in τ_1 and τ_2. For each $i \in A_1$, the instruction I_i is fixed (s_1, rg_1, s_2); we add (q_1, a, α_i), where

$$\begin{aligned}\alpha_i(a_1, a_2, s_1, m, n, s_2; \emptyset) \;=\; & \exists b_1, b_2, s_1', m', n', s_2'\ Reg(b_1, b_2, s_1', m', n', s_2') \wedge s_1' = i \\ & \wedge R(a_1, a_2, s_1, m, n, s_2) \wedge a_1 = b_2 \wedge s_1 = s_2' \wedge n = n' \wedge \\ & (\exists c_1, c_2, s_1'', m'', n'', s_2''\ R(c_1, c_2, s_1'', m'', n'', s_2'') \wedge m' = c_1 \wedge m = c_2)).\end{aligned}$$

Intuitively, for an instance I of R, it verifies whether for all states in A_1 (resp. A_2) there exists a tuple t in I that is the next to the tuple stored in the current register and constitutes a valid transition. Additions to register_2 are encoded similarly.

The "subtract from register_1" part in the rule for (q_1, a) in τ_1 (and τ_2) is encoded in a similar way, and is omitted due to the lack of space. The difference here is that, for each instruction $I_i = (s_1, \text{rg}_1, j, s_2) \in D_1$, we need to add rules for $(q_1, a, \sigma_i^{=0})$ and $(q_1, a, \sigma_i^{\neq 0})$ to separate the case when $\text{rg}_1 \neq 0$ from that when $\text{rg}_1 = 0$.

One can verify that $\tau_1 \equiv \tau_2$ if and only if, M does not halt. Thus the equivalence problem for PT(CQ, S, O) is undecidable.

5.3 Complexity of Existing Publishing Languages

The results of the previous section carry over immediately to the existing publishing languages that support recursion, which are PT(IFP, tuple, normal) (DBMS_XMLGEN) and PT(FO, relation, virtual) (ATG). Table I shows that, however, most of these languages are nonrecursive: PT_{nr}(IFP,tuple,normal) (SQL_mapping, SQL/XML), PT_{nr}(FO, tuple, normal) (FOR-XML), PT_{nr}(CQ, tuple, normal) (annotated XSD, RDB_mapping), and PT_{nr}(CQ, tuple, virtual) (TreeQL). Each of these nonrecursive classes is treated here.

We show that the absence of recursion in these publishing languages simplifies the analyses. Indeed, the evaluation cost of transformations is much lower:

PROPOSITION 5.4. *For publishing transducers τ in PT_{nr}(IFP, tuple, O), the worst-case data complexity for τ-transformations is in* PTIME *(for O normal or virtual).*

PROOF. For any transducer τ in PT_{nr}(IFP, tuple, O) over a relational schema R and for any instance I of R, the depth of the Σ-tree $\tau(I)$ induced by τ-transformation on I is bounded by a fixed k, which is determined by $|\tau|$. From the proof of Proposition 5.1, it follows that the size of $\tau(I)$ is bounded by $O(p(|I|)^k)$, where p is a polynomial. Since each query evaluated during the transformation takes at most PTIME in $|I|$, it takes at most PTIME in $|I|$ in total to generate $\tau(I)$. □

The decision problems also become simpler, to an extent.

THEOREM 5.5. *(1) The emptiness, membership and equivalence problems are undecidable for $PT_{nr}(\mathcal{L}$, tuple, normal) for $\mathcal{L} = \{FO, IFP\}$. (2) The emptiness problem is in* PTIME *for PT_{nr}(CQ, tuple, normal), and is* NP*-complete for PT_{nr}(CQ, tuple, virtual). (3) The membership problem for PT_{nr}(CQ, tuple, O) is Σ_2^p-complete. (4) The equivalence problem for PT_{nr}(CQ, tuple, O) is Π_3^p-complete.*

PROOF. PT_{nr}(FO, tuple, normal) and PT_{nr}(IFP, tuple, normal). It suffices to prove the undecidability for PT_{nr}(FO, tuple, normal). The proof of Proposition 5.2 remains intact for PT_{nr}(FO, tuple, normal). Indeed, all transducers constructed in that proof are nonrecursive (in fact, they produce trees of depth of at most 3).

PT_{nr}(CQ, tuple, O). The upper bounds for the emptiness problem for PT(CQ, tuple, O) (Theorem 5.3(1)), namely, PTIME and NP when O is normal or virtual, respectively, trivially hold for PT_{nr}(CQ, tuple, O). Moreover, the NP hardness proof of the emptiness problem for PT(CQ, tuple, virtual) (Theorem 5.3(1)) uses a nonrecursive transducer and thus extends to PT_{nr}(CQ, tuple, virtual). From these follow the complexity bounds for the emptiness problem for PT_{nr}(CQ, tuple, O).

We next focus on the membership and equivalence problems for this class.

The membership problem. The Σ_2^p-lower bound follows from the proof for PT(CQ, tuple, normal), which uses a nonrecursive transducer (Theorem 5.3(2)).

We now extend the Σ_2^p-algorithm for PT(CQ, tuple, normal) to accommodate virtual nodes. We first establish a small model property. Given any τ in PT_{nr}(CQ, tuple, virtual) and a tree t, if there exists an instance I such that $\tau(I) = t$, then there exists an instance I' of size at most $K \times D \times |t|$, where K is bounded by the size of τ (see Claim 2), and D is the depth of τ, that is, the length of the longest path in its dependency graph G_τ, which is a DAG, since τ is nonrecursive. Indeed, suppose that u and v are nodes in t, u being the parent of v. Then there can be at most D virtual nodes between u and v. As shown by Claim 2, for each of these normal and virtual nodes, at most K

source tuples are needed to generate necessary tuples in the registers. Thus at most $K \times D \times |t|$ source tuples are needed to generate t.

Based on the small model property, a Σ_2^p-algorithm is given as follows. (1) Guess an instance I of at most $K \times D \times |t|$ many tuples. (2) Guess a tree t' as follows: (a) start with t, (b) between any parent-child pair in t, guess and add at most D many virtual nodes; (c) for each virtual node introduced, add a chain of depth at most D consisting of virtual nodes leading to a normal node, Thus t' consists of at most $(D \times |t|)^2$ many nodes. (3) Guess $(D \times |t|)^2$ many tuples, one for the register of each node in t', for all nodes in t'. (4) Extend step 3 of the Σ_2^p algorithm given for Theorem 5.3(2) to check whether t' is a subtree of a tree induced by the transducer on the instance, using a NP oracle. From this the Σ_2^p-upper bound follows.

The equivalence problem. It suffices to show that the problem is Π_3^p-hard for PT_{nr}(CQ, tuple, normal), and then give Π_3^p algorithms for checking the equivalence of transducers in PT_{nr}(CQ, tuple, O). In contrast, the problem was shown to be undecidable for the recursive counterpart of this class (Theorem 5.3(3)).

Lower bound. The proof is by reduction from the $\forall^*\exists^*\forall^*$-3SAT-problem, which is Π_3^p-complete [Papadimitriou 1994]. The latter problem is to determine, given $\varphi = \forall X \exists Y \forall Z \; C_1 \wedge \cdots \wedge C_r$, whether or not φ evaluates to true. Here $\exists Y \forall Z \; C_1 \wedge \cdots \wedge C_r$ is an instance of $\exists^*\forall^*$-3SAT problem, described in the proof of Theorem 5.3(2), in which each literal is either a variable in $X \cup Y \cup Z$, or a negation thereof. We assume that $X = \{x_1, \ldots, x_m\}$, $Y = \{y_1, \ldots, y_n\}$, $Z = \{z_1, \ldots, z_k\}$, and $\forall X$ is a shorthand for $\forall x_1 \cdots \forall x_m$; similarly for $\exists Y$ and $\forall Z$.

Given φ, we define a schema R and transducers τ_1 and τ_2 over R in PT_{nr}(CQ, tuple, normal) such that, for all instances I of R, $\tau_1(I) = \tau_2(I)$ if and only if φ is true.

(a) The relational schema R consists of $R_X(A_1, \ldots, A_m)$ for universal quantification, as well as $R_C(B)$ and R_{OR} as given in the proof of Theorem 5.3(2), for coding existential quantification and disjunction, respectively. An instance I_X of R_X indicates truth assignments for X. When I_X ranges over all truth assignments, we inspect the equivalence between τ_1 and τ_2 for each I_X. To ensure that the coding makes sense, we shall use CQ queries to assure that we only consider well-formed instances of R, that is, (i) instances of R_X contain tuples that are truth assignments for X; (ii) instances of R_C contain $\{0, 1\}$, and (iii) instances of R_{OR} contain I_{OR}.

(b) We next define τ_1 in the following steps (τ_2 is defined using τ_1 later on).

First, we assure that I_X is well-formed. This can be enforced as follows:

$$(q_0, r) \rightarrow (q_1, a, \phi_0(\bar{x}; \emptyset) \equiv R_X(\bar{x})); \quad (q_i, a) \rightarrow (q_{i+1}, a, \phi_i^0(\bar{x}; \emptyset)), (q_{i+1}, a, \phi_i^1(\bar{x}; \emptyset)),$$
$$(q_m, a) \rightarrow (q_{m+1}, b, \phi_m^0(\bar{x}; \emptyset)), (q_{m+1}, b, \phi_m^1(\bar{x}; \emptyset)),$$

where, for $i \in [1, m-1]$ and $j \in \{0, 1\}$, $\phi_i^j(X; \emptyset) = \mathsf{Reg}(X) \wedge (x_i = j)$, and Reg denotes the register. These enforce that in an instance I_X of R_X, only tuples that do encode a truth-assignment for the variables in X generate an a-chain of length m. Thus for each well-formed tuple in I_X, the transducer reaches the state q_{m+1}.

We then add $(q_{m+1}, b) \rightarrow (q_{m+2}, c, \phi_{m+1}(\bar{x}; \emptyset))$ to τ_1, where $\phi_{m+1}(X; \emptyset) \equiv \mathsf{Reg}(X) \wedge \phi(X; \emptyset) \wedge \phi_1(x; \emptyset)$. Here $\phi(X; \emptyset) \equiv \exists Y (\bigwedge_{j=1}^{n} R_C(y_j) \wedge \bigwedge_{i=1}^{r} \psi_i(X, Y))$, which is ϕ_3 given in the Σ_2^p-hardness proof of Theorem 5.3(2), coding $\exists Y \forall Z\ C_1 \wedge \cdots \wedge C_r$; and ϕ_1 is also given in that proof, assuring that instances of R_X and R_C are well-formed.

(c) We define τ_2 to be the same as τ_1, except that $(q_{m+1}, b) \rightarrow (q_{m+2}, c, \phi'_{m+1}(\bar{x}; \emptyset))$, where $\phi'_{m+1}(X; \emptyset) = \mathsf{Reg}(X)$. That is, the b node, if it exists, always has a c child.

To see that the coding is indeed a reduction, observe the following. (i) For the instance I_C of R_C and the instance I_{OR} of R_{OR}, φ evaluates to true if and only if $\tau_1(I_X, I_C, I_{\mathsf{OR}}) = \tau_2(I_X, I_C, I_{\mathsf{OR}})$, when I_X ranges over all truth assignments of X. (ii) For any instance I'_C of R_C, and any instance I'_{OR} of R_{OR}, $I_C \subseteq I'_C$ and $I_{\mathsf{OR}} \subseteq I'_{\mathsf{OR}}$. Then, by the definition of τ_1 and τ_2, if $\tau_1(I_X, I_C, I_{\mathsf{OR}}) = \tau_2(I_X, I_C, I_{\mathsf{OR}})$, then $\tau_1(I_X, I'_C, I'_{\mathsf{OR}}) = \tau_2(I_X, I'_C, I'_{\mathsf{OR}})$, for any instance I_X of R_X, since CQ queries are monotonic. Thus if $\tau_1(I_X, I_C, I_{\mathsf{OR}}) = \tau_2(I_X, I_C, I_{\mathsf{OR}})$ when I_X ranges over all instances of R_X, then $\tau_1 \equiv \tau_2$. Conversely, if $\tau_1 \equiv \tau_2$, then $\tau_1(I_X, I_C, I_{\mathsf{OR}}) = \tau_2(I_X, I_C, I_{\mathsf{OR}})$ when I_X ranges over all truth assignments of X. Putting (i) and (ii) together, we have that φ evaluates to true if and only if $\tau_1 \equiv \tau_2$. This completes the proof of the Π_3^p-lower bound.

Upper bound. We first provide a Π_3^p-algorithm for checking the equivalence of transducers in PT_{nr}(CQ, tuple, normal), and then extend it to PT_{nr}(CQ, tuple, virtual).

PT_{nr}(CQ, tuple, normal). We first present a characterization of the equivalence relation and then show that the characterization can be tested by a Π_3^p-algorithm.

We start with a characterization of when two CQ queries, Q_1 and Q_2, over a relational schema R, satisfy that $|Q_1(I)| = |Q_2(I)|$ for any instance I of R. We assume, without loss of generality, that R consists of a single k-ary relation and that R occurs the same number of times, say n, in both Q_1 and Q_2. We represent a CQ query Q by (i) the set of distinguished variables $X_Q = \{x_1, \ldots, x_p\}$ (the sequence $s_Q = (x_1, \ldots, x_p)$ is called the summary of Q), (ii) the set of nondistinguished variables (existentially quantified) $Y_Q = \{y_1, \ldots, y_q\}$, (iii) the set R_Q of atomic formulas of the form $R(z_1, \ldots, z_k)$, where $z_i \in X_Q \cup Y_Q$, (iv) the set K_Q of constants and (v) the set L_Q of (in)equality constraints $z_i\, \theta\, z_j$, with $z_i, z_j \in X_Q \cup Y_Q \cup K_Q$ and $\theta \in \{=, \neq\}$ (we assume that z_i and z_j are not both constants and that L_Q is consistent).

We denote by $\mathcal{E}_Q$ the equivalence relation $\sim$ on $X_Q \cup Y_Q$ induced by the equality constraints in L_Q, and by $[z]$ the equivalence class of variable z in $\mathcal{E}_Q$. If $z' \in [z]$ and $z' = c \in L_Q$, then we call c the value of $[z]$. The inequality constraints induce a binary relation F_Q on $\mathcal{E}_Q$ as follows: $([z_1], [z_2]) \in F_Q$ if and only if there exists a $z \in [z_1]$ and $z' \in [z_2]$ such that $z \neq z' \in L_Q$. If $z' \in [z]$ and $z' \neq c \in L_Q$, then we call c a nonvalue of $[z]$. Let $x \in X_Q$. We say that $[x]$ is a *constant* if (i) $[x]$ has a value c or (ii) none of the variables in $[x]$ appear in the relations in R_Q. We denote by X_Q^c the set of variables x in X_Q for which $[x]$ is constant; X_Q^{nc} denotes $X_Q \setminus X_Q^c$. We say that X_Q is *reduced* if (i) for all $x \in X_Q$, $[x]$ is not constant and (ii) no two variables in X_Q belong to the same equivalence class. Given X_Q, we construct a reduced version, denoted by X^r_Q of X_Q in PTIME. We define the reduced version of Q, denoted by Q^r, as the query that consists of X^r_Q, $Y^r_Q = Y_Q$, K^r_Q, that is, the subset of K_Q that consists of (non)values of $[y]$ with $y \in X^r_Q \cup Y^r_Q$, R^r_Q, in which all variables in $x \in X_Q \setminus X^r_Q$ are replaced by the variable $x' \in X^r_Q$ such that $x \in [x']$ or is the value of $[x]$, and L^r_Q is modified similarly. We say that Q_1 and Q_2 are *c-equivalent*, denoted by $Q_1 \equiv_c Q_2$, if $Q^r_1 \equiv Q^r_2$. The $\equiv_c$ relation is needed for characterizing transducer equivalence as indicated in the following, and can be tested in terms of $\equiv$.

CLAIM 3. *$Q_1 \equiv_c Q_2$ if and only if for all instances I, $|Q_1(I)| = |Q_2(I)|$.*

PROOF. Assume that $Q^r_1 \equiv Q^r_2$ and let I be an instance of R. Then, for any $t \in Q^r_1(I)$, there exists a unique completion $\hat{t}$ such that $\hat{t} \in Q_1(I)$. Indeed, the remaining attributes are either a constant or are equal to attributes already appearing in t. Hence $|Q_1(I)| = |Q^r_1(I)| = |Q^r_2(I)| = |Q_2(I)|$, from which the result follows.

Conversely, assume that for all I, $|Q_1(I)| = |Q_2(I)|$ (or equivalently, $|Q^r_1(I)| = |Q^r_2(I)|$). We then show that $Q^r_1 \equiv Q^r_2$. We first show that $|X_{Q^r_1}| = |X_{Q^r_2}|$. Assume that $|X_{Q^r_1}| < |X_{Q^r_2}|$ (similarly for $>$). Let ρ_1 be a valuation of $X_{Q^r_1} \cup Y_{Q^r_1}$ that is *order-preserving* with respect to Q^r_1, that is, for all $z_1 \theta z_2 \in L_{Q^r_1}$, $\rho(z_1) \theta \rho(z_2)$ holds. Let ρ_2 be any order-preserving valuation that agrees with ρ_1 on $X_{Q^r_1}$. Denote by I_{ρ_1} the instance $\{(\rho(z_1), \dots, \rho(z_k)) \mid R(z_1, \dots, z_k) \in R_{Q^r_1}\}$; similarly for I_{ρ_2}. Since $X_{Q^r_1}$ is reduced, $Q^r_1(I_{\rho_1}) = Q^r_1(I_{\rho_2})$. However, since $X_{Q^r_2}$ is reduced too, there must exist a variable in $X_{Q^r_2}$ that is not constant and not equal to any other variable in $X_{Q^r_2}$, and that corresponds to a variable in $Y_{Q^r_1}$. Hence $Q^r_2(I_{\rho_1}) \neq Q^r_2(I_{\rho_2})$. Let $I = I_{\rho_1} \cup I_{\rho_2}$, we then get (by monotonicity) that $Q^r_1(I) = Q^r_1(I_{\rho_1})$, while $Q^r_2(I)$ strictly contains $Q^r_2(I_{\rho_1})$. Thus $|Q^r_1(I)| < |Q^r_2(I)|$; contradiction. Hence $|X_{Q^r_1}| = |X_{Q^r_2}|$.

Next, we use the characterization of the equivalence of CQ-queries (with $\neq$) given in Klug [1988]. That is, $Q^r_1 \subseteq Q^r_2$, if for each valuation ρ of $X_{Q^r_1} \cup Y_{Q^r_1}$ that is order-preserving with respect to Q^r_1, it is the case that $\rho(s_{Q^r_1}) \in Q^r_2(I_\rho)$. We show that $Q^r_1 \subseteq Q^r_2$ (similarly for $Q^r_2 \subseteq Q^r_1$). Denote the tuples in I_ρ by $t_i = (\rho(z^i_1), \dots, \rho(z^i_k))$, for $i \in [1, n]$. Here, the z^i_js denote the variables in the ith occurrence of R in Q^r_1. Now, for a given ρ, we have that $\rho(s_{Q^r_1}) \in Q^r_1(I_\rho)$ and therefore $Q^r_2(I_\rho)$ is nonempty (because $|Q^r_1(I_\rho)| = |Q^r_2(I_\rho)|$). Let $s \in Q^r_2(I_\rho)$ and assume that s is obtained by the combination of n tuples $s_1, \dots, s_n$ such that $s_i = t_{\pi(i)}$ for some mapping $\pi : [1, n] \to [1, n]$. Let h be the mapping from $X_{Q^r_2} \cup Y_{Q^r_2}$ to $X_{Q^r_1} \cup Y_{Q^r_1}$ defined by $h((z')^i_j) = z^{\pi(i)}_j$. Here, the primed variables correspond to variables in Q^r_2. Clearly, h maps an occurrence of R in Q^r_2 to an occurrence of R in Q^r_1. We now argue that h must also map the summary of Q^r_2 to that of Q^r_1. From this, it follows that the valuation $\rho \circ h$ of $X_{Q^r_2} \cup Y_{Q^r_2}$ gets $\rho(s_{Q^r_1})$ in $Q_2(I_\rho)$. Since this argument holds for any ρ, we may then conclude that $Q^r_1 \subseteq Q^r_2$, as desired.

It remains to show that $h(s_{Q^r_2}) = s_{Q^r_1}$. Suppose, otherwise, that either there exists a variable $x' \in s_{Q^r_2}$ such that $[h(x')]$ does not contain any variable from $s_{Q^r_1}$, or $[h(x')]$ has more than one variable in common with $s_{Q^r_1}$. Observe that, since Q^r_2 is reduced, $[h(x')]$ cannot have a value. Therefore, valuations can assign an arbitrary value to the variables in $[h(x')]$. Hence a similar argument as above shows the existence of an instance I such that $|Q^r_1(I)| < |Q^r_2(I)|$. This contradicts our assumption and therefore $h(s_{Q^r_2}) = s_{Q^r_1}$. □

We now continue toward the characterization of equivalence. We introduce some notations first. Let (q, a) be a state-label pair in τ and let $(q, a) \rightarrow (q_1, a_1, \psi_1), \ldots, (q_k, a_k, \psi_k)$ be the corresponding rule in τ. We partition indices $[1, k]$ and associate $S_\tau(q, a) = \{S_1, S_2, \ldots, S_\ell\}$ of $[1, k]$ with (q, a) such that (i) each S_i consists of consecutive indices in $[1, k]$; (ii) if $s, t \in S_i$ then $a_s = a_t$; and (iii) no two S_i and S_j's can be merged and still satisfy (i) and (ii). We denote by $lab(S_i)$ the (unique) label a_s of the indices in S_i. Let G_τ be the dependency graph of τ. We define the *type* of a node $v(q, a)$ in G_τ as the list $[lab(S_1), \ldots, lab(S_\ell)]$. Let ρ be a path in G_τ starting from the root node of G_τ. Denote by Q_ρ the composition of all queries along ρ; assume that ρ ends in $v(q, a)$. We denote by Q^i_ρ the CQ query obtained by composing ψ_i, that is, the ith query in the RHS of the rule for (q, a) in τ, with Q_ρ. We assume that G_τ does not contain nodes that are not reachable from the root r, and moreover, that for each path ρ in G_τ, Q_ρ is satisfiable (note that the latter can be tested in a similar way as in the proof of Theorem 5.3 (1)). Two graphs, G_{τ_1} and G_{τ_2}, are called equivalent, denoted by $G_{\tau_1} \cong G_{\tau_2}$, if there exists a homeomorphism $h : G_{\tau_1} \rightarrow G_{\tau_2}$ such that its inverse h^{-1} exists and is also a homeomorphism and, moreover, both h and h^{-1} preserve the labels and types of the nodes. It is natural to extend the notion of $\equiv_c$ and Claim 3 to union of CQ queries. We then have the following characterization of transducer equivalence:

CLAIM 4. *For any two publishing transducers τ_1 and τ_2 in PT$_{nr}$(CQ, tuple, normal), we have that $\tau_1 \equiv \tau_2$ if and only if (i) $G_{\tau_1} \cong G_{\tau_2}$, and (ii) for all paths ρ in G_{τ_1}, each $S_i \in S_{\tau_1}(q, a)$, $J_i \in S_{\tau_2}(h(q, a))$, $\bigcup_{j \in S_i} Q^j_\rho \equiv_c \bigcup_{j \in J_i} Q^j_{h(\rho)}$ in case ρ ends in (q, a) with a not equal to* text*; and $\bigcup_{j \in S_i} Q^j_\rho \equiv \bigcup_{j \in J_i} Q^j_{h(\rho)}$ otherwise. (Here, h is a homomorphism between G_{τ_1} and G_{τ_2}.)*

PROOF. Suppose that there exists an instance I of R such that $\tau_1(I) \neq \tau_2(I)$; let v be the first node encountered in the depth-first traversal of $\tau_1(I)$ such that *children*(v) is different from *children*(w), where w is the node in $\tau_2(I)$ corresponding to v. Assume that τ_1 (resp. τ_2) is in state (q, a) (resp. (q', a)) when reaching v (resp. w). Such an instance I exists if either $v(q, a)$ and $v(q', a)$ have different types, or they have the same type but some labels appear with a different multiplicity in *children*(v) and *children*(w). This clearly implies that either condition (i) or (ii) does not hold, by Claim 3.

Conversely, using the monotonicity of CQ queries and the assumption that all queries Q_ρ along paths ρ in G_{τ_1} and G_{τ_2} are satisfiable, it is easily verified that the failure of one of the conditions (i) or (ii) implies the existence of an instance I on which τ_1 and τ_2 disagree, again by Claim 3. □

As will be shown in Theorem 6.1(1), composed CQ queries can be rewritten as a program in nonrecursive LINDATALOG, in PTIME. A LINDATALOG program consists of rules of the form: $p(\bar{x}) \leftarrow p_1(\bar{x}_1), \ldots, p_n(\bar{x}_n)$, such that at most one p_i is an IDB predicate (i.e., relation name), and we allow some p_j to be $\neq$ [Abiteboul et al. 1995]. Let $Q^1_\rho \cup \cdots \cup Q^p_\rho$ be a union of conjunctive queries as in Claim 4. It is possible to encode each composed query $Q^i_\rho = Q^i_{n_i} \circ \ldots \circ Q^i_1$ in LinDatalog because in each Q^i_j, although there may be multiple occurrences of *Reg*, all of these indicate the same single tuple, since the transducer is in $\mathrm{PT}_{nr}(\mathrm{CQ}, \mathrm{tuple}, O)$; as a result, for each Q^i_j, we can define an IDB predicate p^i_j such that p^i_j has a unique rule and, in the RHS of the rule, there is a unique IDB predicate p^i_{j-1} that encodes Q^i_{j-1}. As a consequence, we obtain a LinDatalog program Π^i for each Q^i_ρ; the combination of these in a single nonrecursive program Π encodes the union of the Q^i_ρ's.

We call a LinDatalog program *deterministic* if each IDB predicate $p(\bar{x})$ has only one rule (including the initialization rule). For a LinDatalog program Π consisting of IDB predicates $p_1, \ldots, p_n$, we define a deterministic sub-query Π' of Π to be a deterministic nonrecursive LinDatalog program such that Π' consists of $p_1, \ldots, p_n$ and, moreover, for each $i \in [1, n]$, Π' contains only a single rule for p_i from Π.

Furthermore, a straightforward induction on n suffices to show the following claim. We remark that if Π is not linear or not deterministic, the claim does not hold, since some Q_j may appear in Q_n exponentially many times.

Claim 5. *For each nonrecursive deterministic* LinDatalog *program Π, a CQ query Q can be computed in $O(|\Pi|)$ time such that Q and Π are equivalent.*

It remains to show that for given transducers τ_1 and τ_2, we can check the conditions in Claim 4 in Π^p_3. We give an algorithm for testing that τ_1 and τ_2 are not equivalent, as follows. (a) Guess a mapping h from G_{τ_1} to G_{τ_2}. (b) Check whether h and h^{-1} make $G_{\tau_1} \cong G_{\tau_2}$. If not, reject. (c) If $G_{\tau_1} \cong G_{\tau_2}$, guess a path ρ from the root of G_{τ_1}, and compute nonrecursive LinDatalog programs Π^i_1 encoding the union of Q^i_ρ's, and Π^i_2 for $h(\rho)$ in G_{τ_2}, as described in Claim 4. Note that Π^i_1's and Π^i_2's can be computed in PTIME. (d) Check whether all Π^i_1 and Π^i_2 are (c-)equivalent as described in Claim 4. One can easily verify that $\equiv_c$ and Claim 3 can be extended to nonrecursive LinDatalog. By the definition of $\equiv_c$, (c-)equivalence can be checked in terms of equivalence $\equiv$, and thus we shall focus on $\equiv$ only. Provided that Π^i_1 and Π^i_2 are not equivalent for some i, then we can conclude that τ_1 and τ_2 are not equivalent. If step (d) is in Σ^p_2, we can decide whether τ_1 and τ_2 are not equivalent in $\Sigma^p_3 = \mathrm{NP}^{\Sigma^p_2}$, and thus its complement, testing the equivalence of two transducers in $\mathrm{PT}_{nr}(\mathrm{CQ}, \mathrm{tuple}, \mathrm{normal})$, is in Π^p_3.

For step (d), it is easy to verify that Π^i_1 and Π^i_2 are not equivalent if and only if either there exists a deterministic subquery $(\Pi^i_1)'$ of Π^i_1 that is not contained in Π^i_2, or vice versa. Leveraging this, in step (d) we guess a deterministic subquery $(\Pi^i_1)'$ of Π^i_1 and check whether $(\Pi^i_1)'$ is not contained in Π^i_2, and similarly with the roles of Π^i_1 and Π^i_2 reversed. By Claim 5, $(\Pi^i_1)'$ is PTIME definable as a CQ query Q. The next claim shows that deciding whether Q is not contained in Π^i_2 is in Σ^p_2, as desired.

Claim 6. *It is in Π^p_2 time to determine, given a CQ query Q and a nonrecursive* LinDatalog *program Π, whether or not Q is contained in Π.*

Proof. Proposition 2.10 of van der Meyden [1997] shows that the combined complexity of model checking for indefinite order databases and CQ queries, that is, checking whether an indefinite order database is a model of a CQ query, is PTIME equivalent to containment of CQ queries with $\neq$. We first generalize this and show that the containment of CQ queries with $\neq$ in nonrecursive LinDatalog programs, is PTIME reducible to model checking for indefinite order databases and nonrecursive LinDatalog. We then show that the combined complexity of the latter is in Π_2^p.

Given a CQ query $Q = \exists \bar{y}\, \psi(\bar{x}, \bar{y})$ (with $\neq$) and a nonrecursive LinDatalog program $\Pi(\bar{x})$, we define an indefinite order database D and a nonrecursive LinDatalog program Ψ. As in van der Meyden [1997], let D consist of atoms in the conjunction $\psi(\bar{a}, \bar{b})$, where $\bar{a}$ and $\bar{b}$ are fresh constant of appropriate sorts (with orderings to code $\neq$), and define Ψ to be $\Pi(\bar{a})$. Obviously, if Q is contained in Π, then every model (database) of D satisfies $\Pi(\bar{a})$. Conversely, suppose that D is a model of $\Pi(\bar{a})$. Then, every model of D satisfies $\Pi(\bar{a})$. Since D is the "canonical database" and model checking ranges over all models of D, Q is contained in Π. Hence the containment problem is PTIME reducible to model checking for indefinite order databases and nonrecursive LinDatalog programs.

We next give a Π_2^p-time model-checking algorithm. Given any indefinite order database D and nonrecursive LinDatalog program Π, the algorithm guesses a minimal model M of D, and checks whether M does not satisfy Π (see [van der Meyden 1997] for discussions of minimal models). One can verify that checking whether or not M satisfies Π can be done in NP. Thus the complement of the model checking problem is in Σ_2^p. Hence the combined complexity for model checking of indefinite order databases and nonrecursive LinDatalog programs is in Π_2^p. □

PT$_{nr}$(CQ, tuple, virtual). Given two transducers τ_1, τ_2 in PT$_{nr}$(CQ, tuple, virtual), we construct in PTIME equivalent τ_1', τ_2', without virtual nodes, such that the Π_3^p-algorithm for the equivalence problem of PT$_{nr}$(CQ, tuple, normal) can be used.

Given a transducer τ in PT$_{nr}$(CQ, tuple, virtual), we define an equivalent τ' such that it contains normal output nodes only, but some of its queries may be in nonrecursive LinDatalog rather than CQ. As previously, let G_τ be the dependency graph of τ extended with an order on the vertices. We refer to nodes labeled with a virtual tag as virtual nodes, and as normal nodes otherwise. We define a dependency graph G_τ' by removing virtual nodes from G_τ as follows. For any two normal nodes $n_1 = v(q, a), n_2 = v(q', a')$ in G_τ, let $G(n_1, n_2)$ be the largest connected subgraph of G_τ such that, except n_1, n_2, it consists of only virtual nodes, and for any virtual node in the graph, it is on a path from n_1 to n_2. Since τ is nonrecursive, $G(n_1, n_2)$ is a rooted DAG in which n_1 is the root and n_2 is the sink. We say that $G(n_1, n_2)$ is nonempty if it has at least one virtual node. The graph G_τ' is obtained from G_τ by substituting a new edge (n_1, n_2) for $G(n_1, n_2)$ in G_τ, as long as $G(n_1, n_2)$ is nonempty, when (n_1, n_2) range over all pairs of normal nodes in G_τ. The query associated with (n_1, n_2) is equivalent to the union of the composition of CQ queries along each path from n_1 to n_2. By treating $G(n_1, n_2)$ as a dependency graph, n_1 as the root, and n_2 as the output node, Theorem 6.1(1) gives a method to define the query in nonrecursive LinDatalog. Then, from G_τ', we can derive transducer τ' as follows: for each $v(q, a)$, define the rule $(q, a) \rightarrow (q_1, a_1, \psi_1) \ldots (q_k, a_k, \psi_n)$, where $(v(q_1, a_1), \ldots, v(q_k, a_k))$ is the list of *children*$(v(q, a))$, as ordered in G_τ, and ψ is the query associated with the edge from $v(q, a)$ to $v(q_i, a_i)$. It is straightforward to show that τ and τ' are equivalent, and that τ' can be constructed in PTIME.

We now obtain an algorithm for testing the equivalence of τ_1 and τ_2, as follows. First, construct equivalence τ_1' and τ_2' without virtual nodes, as described previously, in PTIME. Then check the equivalence of τ_1' and τ_2' by using the Π_3^p algorithm for PT_{nr}(CQ, tuple, normal), as stated earlier. It suffices to observe that the algorithm for PT_{nr}(CQ, tuple, normal) trivially extends to transducers in which some queries are in nonrecursive LINDATALOG, rather than CQ.

6. EXPRESSIVENESS OF PUBLISHING TRANSDUCERS

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

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 relations to relations. To this end, consider a publishing transducer τ. For the rest of this section, we fix a designated output label a_o, which is not a virtual tag. For any instance I of R, the τ-transformation on I yields a final tree ξ with local storage in $\mathrm{Tree}_{Q\times\Sigma}$, from which the output Σ-tree $\tau(I)$ is obtained by removing local stores and states (recall from Section 3). The *output relation* induced by τ on I, denoted by $R_\tau(I)$, is then defined to be the union of the registers $Reg_{a_o}(v)$ 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 $R_\tau(I)$. When τ is viewed as a relation-to-tree mapping, we refer to τ as a *tree generating mapping*.

We want to compare the expressiveness of one class $\mathcal{A} = \mathrm{PT}(\mathcal{L}_1, S_1, O_1)$ with that of another class $\mathcal{B} = \mathrm{PT}(\mathcal{L}_2, S_2, O_2)$, both as a tree generation and a relational query language. We say that $\mathcal{A}$ is *contained in* $\mathcal{B}$ as a tree/relational query language, denoted by $\mathcal{A} \subseteq \mathcal{B}$, if for any τ_1 in $\mathcal{A}$ defined for a relational schema R, there exists τ_2 in $\mathcal{B}$ 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 $\mathcal{A} = \mathcal{B}$, if $\mathcal{A} \subseteq \mathcal{B}$ and $\mathcal{B} \subseteq \mathcal{A}$. We say that $\mathcal{A}$ is *properly contained in* $\mathcal{B}$, denoted by $\mathcal{A} \subset \mathcal{B}$, if $\mathcal{A} \subseteq \mathcal{B}$ but not $\mathcal{A} = \mathcal{B}$. These notions extend to comparing $\mathrm{PT}(\mathcal{L}, S, O)$ versus other tree generating formalisms, and versus relational query languages.

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

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 publishing transducers as a relational query language. We characterize some of the fragments in terms of known query languages and complexity classes. From these, we can then compare the expressive power of the different publishing transducer fragments.

We review three fragments of datalog. One fragment is *linear datalog* (see e.g., [Abiteboul et al. 1995]), denoted by LinDatalog, which we have seen in the proof of Theorem 5.5. It consists of datalog programs in which each rule is of the 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 (i.e., relation name). We allow some p_j to be $\neq$. We assume an output relation ans containing the result of the query expressed by the program. Another fragment is LinDatalog(FO) (see e.g., Grädel [1992]) which extends LinDatalog by allowing p_i to be an arbitrary FO-formula over the EDB predicates. In Grädel [1992] it is shown that LinDatalog(FO) captures NLOGSPACE over ordered databases.

Recall from Section 3, that we do not assume an ordering to be available to the query language at hand.

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) =* LinDatalog.
(3) *PT(FO, tuple, O) =* LinDatalog(FO) $\subset$ NLOGSPACE.
(4) *PT(FO, relation, O) and PT(IFP, relation, O) capture* PSPACE.
(5) *PT(IFP, tuple, O) = IFP.*

Here, the output O can be either virtual or normal.

Proof. The proof is referred to the Appendix. □

Proposition 6.2 relates the different fragments of transducers. While Theorem 6.1(1) tells us that virtual nodes do not add expressive power, and thus we only need to consider PT($\mathcal{L}$, S, normal), Proposition 6.2 shows that we need to treat publishing transducers with relation stores and those with tuple stores separately (3, 5, 7). Moreover, while IFP does not add expressive power over FO in PT($\mathcal{L}$, relation, O), it does in PT($\mathcal{L}$, tuple, O) (4, 2). Moreover, replacing CQ with FO in PT(CQ, S, O) leads to increase in expressiveness when S is either relation or tuple (1, 6).

PROPOSITION 6.2. *When treated as relational query languages:*

(1) $PT(CQ, tuple, O) \subset PT(FO, tuple, O),$
(2) $\subseteq PT(IFP, tuple, O),$
(3) $\subset PT(FO, relation, O),$
(4) $= PT(IFP, relation, O),$
(5) $PT(CQ, tuple, O) \subset PT(CQ, relation, O),$
(6) $\subset PT(FO, relation, O),$
(7) $PT(CQ, relation, O) \not\subseteq PT(FO, tuple, O).$

The containment in statement (2) is proper if NLOGSPACE $\neq$ PTIME.

PROOF. The proof is referred to the Appendix. □

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 [Courcelle 1994] as well as with regular tree languages (specialized DTDs).

Equivalence and separation. As opposed to Proposition 6.2, Proposition 6.3 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(IFP, relation, normal) = PT(FO, relation, normal). The other results in Proposition 6.3 are comparable to their counterparts in Theorem 6.2. In particular, it shows that PT(FO, relation, virtual) = PT(IFP, relation, virtual).

PROPOSITION 6.3. *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) $\subseteq PT(IFP, tuple, virtual);$
(5) $PT(CQ, relation, virtual) \subset PT(FO, relation, virtual);$
(6) $= PT(IFP, relation, virtual);$
(7) $PT(\mathcal{L}, tuple, O) \subset PT(\mathcal{L}, relation, O);$
(8) $PT(\mathcal{L}, relation, normal) \not\subseteq PT(\mathcal{L}', tuple, virtual);$
(9) $PT(\mathcal{L}, tuple, virtual) \not\subseteq PT(\mathcal{L}', relation, normal)$ *with* $\mathcal{L}' \subset \mathcal{L};$
(10) $PT(CQ, tuple, virtual) \not\subseteq PT(CQ, relation, normal);$
(11) $PT(FO, tuple, virtual) \not\subseteq PT(FO, relation, normal)$

where $\mathcal{L}$ *and* $\mathcal{L}'$ *are in* {*IFP,FO,CQ*}, S *is tuple or relation, and* O *is normal or virtual. The containment in (5) is proper if* PTIME $\neq$NLOGSPACE.

PROOF. (1) The inclusion is immediate. To show proper containment it suffices to show that there is a transducer τ in PT(CQ, tuple, virtual) not expressible in PT(IFP, relation, normal). Consider the transducer τ_1 in the proof of Proposition 5.1 that unfolds a graph to a tree (the stop condition ensures that the process stops when cycles are involved). Modify this transducer in such a way that it, using virtual nodes, outputs a tree of depth one, gathering all expanded nodes in in order below the root. As shown in Proposition 5.1(3), there is an input graph I_n such that $\tau_1(I_n)$ contains 2^n nodes (all directly below the root). In contrast, any transducer in PT(IFP, relation, normal) can only output polynomially many nodes (in the size of the input graph) below the root. Therefore, it follows that, in fact, PT(CQ, tuple, virtual) $\not\subseteq$ PT(IFP, relation, normal).

(2) Let q be a Boolean query in $\mathcal{L}' \setminus \mathcal{L}$. We then simply define a transducer τ_q, which produces a tree $r(a)$, that is, a tree consisting of a root node (labeled with r) with a single child node (labeled with a), if q evaluates to true on the input database, and it produces a single-node tree r otherwise. Clearly, τ_q is not expressible in PT($\mathcal{L}$, S, normal), since q is not definable in $\mathcal{L}'$.

(3) Let q be a Boolean query definable in FO but not in LinDatalog. Such a query can always be found, since LinDatalog only defines monotone queries. Define a transducer τ_q, which produces a tree $r(a)$ if q evaluates to true on the input database, and it produces a single-node tree r otherwise. Assume there is a transducer τ in PT(CQ, tuple, virtual) equivalent to τ_q. Then modify τ into a transducer τ' such that it outputs a node a_o with the empty tuple in its register whenever it outputs an a. In addition, make all virtual nodes nonvirtual. Then, τ' defines q, which means, by Theorem 6.1(1), that q is definable in LinDatalog. Contradiction.

(4) We show that PT(FO, tuple, virtual) $\subset$ PT(IFP, tuple, virtual) if PTIME $\neq$ NLOGSPACE. Let q be a Boolean IFP query over ordered databases (a PTIME query) not in NLOGSPACE. Consider again the transducer τ_q, which produces a tree $r(a)$ if q evaluates to true on the input database, and produces a trivial tree r otherwise. Assume there is an equivalent transducer in PT(FO, tuple, virtual), it could easily be extended to compute q as a relation query. Indeed, simply output a node labeled a_o with the empty tuple when a is output. This contradicts Theorem 6.1(3).

(5) Let τ be a transducer in PT(FO, relation, virtual) defining a Boolean relational query q, which is not in PT(CQ, relation, virtual). By Theorem 6.2(1) and Proposition 6.2(6), such a transducer exists. Modify τ into τ' as follows. Change the arity of the register of the output label a_o from nullary to unary such that $R_{\tau'}(I)$ is nonempty if and only if $q(I)$ is true. Modify righthand sides of the rules such that, whenever an a_o is output with nonempty register, a b-labeled first-child is output as well. The latter is clearly definable in FO. Here, b is a new label. Denote by q' the relation-to-tree query expressed by τ'. We now argue that q' cannot be defined in PT(CQ, relation, virtual). Assume there is such a transducer τ''. Then, τ'' and τ' define the same relation-to-tree mappings. But, since the nonemptiness of the register of an a_o-labeled nodes is encoded in the tree, both τ'' and τ' also define the same relational query q, which leads to contradiction.

(6) To simulate a transducer in PT(IFP, relation, virtual), by one in PT(FO, relation, virtual), it suffices to remark that the evaluation of an IFP-formula can be simulated by a (possible unbounded) number of iterations of FO-formulas.

The simulation is conducted by constructing a linear tree such that each node in the tree performs one iteration. Furthermore, each node in the linear tree is virtual.

(7) The containment is proper because transducers in PT($\mathcal{L}$, relation, O) are capable of generating trees of exponential depth even when $\mathcal{L}$ =CQ (Proposition 5.1(4)), whereas transducers in PT($\mathcal{L}$, tuple, O) can only induce trees of polynomial depth (Proposition 5.1(2)).

(8) Proof is similar to (7).

(9) Take a sentence $\psi \in \mathcal{L}$ not definable in $\mathcal{L}'$. Then, the publishing transducer defined by the rule $\delta(q_0, r) \to (q, a, \psi)$ and $\delta(q, a) \to \varepsilon$. Clearly, the latter is not definable in PT($\mathcal{L}'$,relation,normal) without ψ being definable in $\mathcal{L}'$.

(10–11) Take R as a binary relation and let s and t be two constants. Consider the publishing transducer τ in PT(CQ,tuple,virtual) defined by the rules: $\delta(q_0, r) \to (q, v, R(s, x))$ and $\delta(q, v) \to (q, v, \exists y \, \mathrm{Reg}_v(y) \wedge R(y, x))$ $(q, a, \exists y \, \mathrm{Reg}_v(y) \wedge y = t)$. Here, v is a virtual label, while a is not. Then, τ outputs the tree $r(\underbrace{a \cdots a}_{k})$, where k is the number of simple paths from s to t. Since it cannot be checked in CQ or FO whether there is a (simple) path from s to t, τ is not definable in PT(CQ,relation,normal) and PT(FO,relation,normal). □

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 (as 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. In a way similar 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 $(\phi_a)_{a\in\Sigma}$. In the sequel, we characterize the expressive power of publishing transducers in terms of logical first- and second-order transductions and PTIME and PSPACE transductions, which we introduce next.

An immediate mismatch between transducers and transductions arises: logical first-order transductions of a fixed arity can only increase the size of the output structure by a polynomial, whereas publishing transducers (even with tuple registers) can generate output trees of exponential size (see Proposition 5.1).

Logical first-order transductions. Let R be a relational schema and $\mathcal{L}$ a logic over R. For any instance I of R, we denote by $adom(I)$ the set of constants appearing in I. For a formula $\phi(x_1, \ldots, x_k)$ in $\mathcal{L}$, we denote by $\phi(I)$ the relation $\{\bar{d} \mid I \models \phi(\bar{d})\}$.

For any natural number k, we define an *$\mathcal{L}$-transduction of width k* as a tuple

$$T = \left(\phi_{\mathrm{dom}}(\bar{x}), \phi_{\mathrm{root}}(\bar{x}), \phi_e(\bar{x}; \bar{y}), \phi_<(\bar{x}; \bar{y}; \bar{z}), \phi_{fc}(\bar{x}; \bar{y}), \phi_{\mathrm{ns}}(\bar{x}, \bar{y}), (\phi_a(\bar{x}))_{a\in\Sigma}\right),$$

consisting of formulas in $\mathcal{L}$, and $\bar{x}$, $\bar{y}, \bar{z}$ are k-ary variables. Moreover, these formulas satisfy the following constraints. For any instance I of R, we have that:

(a) $\phi_e(I)$ defines a directed, singly rooted, acyclic graph over k-tuples, i.e., a DAG;

(b) $\phi_{\text{root}}(I)$ contains one element and this element is the root of the DAG;

(c) for all $\bar{d}, \bar{d}', \bar{d}'' \in \text{adom}(I)^k$, if $I \models \phi_<(\bar{d}, \bar{d}', \bar{d}'')$, then $(\bar{d}, \bar{d}') \in \phi_e(I)$, $(\bar{d}, \bar{d}'') \in \phi_e(I)$, and $\phi_<(\bar{d}, \bar{y}, \bar{z})$ is a total order, that is, $\phi_<$ defines an ordering on the children of each $\bar{x}$;

(d) $\phi_{\text{fc}}(I)$ and $\phi_{\text{ns}}(I)$ are the first-child and next-sibling relations induced by $\phi_e, \phi_<$;

(e) for every a and a' in Σ such that $a \neq a'$, $\phi_a(I) \cap \phi_{a'}(I) = \emptyset$. That is, every k-tuple has at most one label; and finally,

(f) $\phi_{\text{dom}}(I)$ defines the domain of the tree; that is, the projection of any of the above relations on any column is always a subset of $\phi_{\text{dom}}(I)$. Moreover, $\phi_{\text{dom}}(I) = \bigcup_{a \in \Sigma} \phi_a(I)$.

In summary, on any instance I of R, the transduction T defines a DAG with domain $\phi_{\text{dom}}(I)$, edge-relation $\phi_e(I)$, ordering on siblings $\phi_<(I)$, and labels $\phi_a(I)$. Note that ϕ_{root}, ϕ_{fc}, and ϕ_{ns} are definable in FO from ϕ_e and $\phi_<$, but not in CQ.

As described previously, we see the DAG as a representation of a tree. We define $T(I)$ to be the tree obtained by unfolding the DAG. We remark that we only consider those nodes that are reachable through the edge relation from the root node. Unreachable nodes are discarded.

We say that an $\mathcal{L}$-transduction T is *fixed-depth* if there is an ℓ such that, for every input I, $T(I)$ is a tree of depth at most ℓ.

Logical second-order transductions. For a logic $\mathcal{L}$ and a natural number k, we define a *second-order $\mathcal{L}$-transduction of width k* as a tuple,

$$T = (\phi_{\text{dom}}(X), \phi_{\text{root}}(X), \phi_e(X; Y), \phi_{fc}(X, Y), \phi_{ns}(X, Y), \phi_<(X; Y; Z)), (\phi_a(X)_{a \in \Sigma}),$$

of $\mathcal{L}$-formulas over a schema R extended with (an unbounded number of) second order variables X, Y, Z, ..., of arity k. An $\mathcal{L}$-formula $\phi(X_1, \ldots, X_n)$ on an input structure I then defines the set $\phi(I) = \{(\bar{A}_1, \ldots, \bar{A}_n) \mid I \models \phi(\bar{A}_1, \ldots, \bar{A}_n)\}$, where each $\bar{A}_i$ has arity k. Similar to logical first-order transductions, the formulas in T satisfy the following constraints, ensuring that the output is a labeled ordered DAG. That is, T defines the DAG with domain $\phi_{\text{dom}}(I)$, edge-relation $\phi_e(I)$, ordering on siblings $\phi_<(I)$, and labels $\phi_a(I)$,

$\mathcal{C}$-transductions. A first-order $\mathcal{C}$-transduction of width k is defined to be a tuple of $\mathcal{C}$-TMs $(M_e, M_{\text{root}}, M_{fc}, M_{ns}, M_<, (M_a)_{a \in \Sigma})$ such that, for every I,

(a) $M_e(I) = \{(\bar{d}, \bar{d}') \mid (I, \bar{d}, \bar{d}') \text{ is accepted by } \mathrm{M_e}\}$ defines a DAG;

(b) $M_{\text{root}}(I) = \{\bar{d} \mid (I, \bar{d}) \text{ is accepted by } \mathrm{M_{root}}\}$ contains the root of the DAG;

(c) $M_<(I) = \{(\bar{d}, \bar{d}', \bar{d}'') \mid (I, \bar{d}, \bar{d}', \bar{d}'') \text{ is accepted by } \mathrm{M_<}\}$ and for all $\bar{d}, \bar{d}'$, $\bar{d}'' \in \text{adom}(I)^k$, when $(\bar{d}, \bar{d}', \bar{d}'') \in M_<(I)$, then $(\bar{d}, \bar{d}') \in M_e(I)$ and $(\bar{d}, \bar{d}'') \in M_e(I)$, and for each $\bar{d}$, $\{(\bar{d}', \bar{d}'') \mid (\bar{d}, \bar{d}', \bar{d}'') \in M_<\}$ is a total order;

(d) $M_{fc}(I)$, $M_{ns}(I)$ are the first-child and next-sibling relation induced by M_e, $M_<$;

(e) $M_a(I)$ are such defined that each node in the domain has precisely one label.

Second-order $\mathcal{C}$-transductions are defined in a similar way.

THEOREM 6.4.

(1) *When $\mathcal{L}$ ranges over CQ, FO and IFP, every $\mathcal{L}$-transduction is definable in PT($\mathcal{L}$, tuple, virtual).*

(2) *When $\mathcal{L}$ ranges over FO and IFP, 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, normal) that is not definable as an FO-transduction.*

(4) *When $\mathcal{L}$ ranges over CQ, FO and IFP, 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(IFP, tuple, virtual) contain the second-order* PSPACE- *and first-order* PTIME-*transductions.*

PROOF. (1) We need to show that, given an $\mathcal{L}$-transduction T, there always exists a transducer τ_T in PT($\mathcal{L}$,tuple,virtual) such that $\tau(I) = T(I)$ for any instance I.

Let $T = (\phi_{\text{dom}}, \phi_{\text{root}}, \phi_e, \phi_<, \phi_{fc}, \phi_{fs}, (\phi_a)_{a\in\Sigma})$ be an $\mathcal{L}$-transduction of width k. Let $\Sigma = \{a_1, \ldots, a_n\}$. We then define the transducer $\tau_T = (Q, \Sigma, \Theta, q_0, \delta, \Sigma_e)$, where $Q = \{q_0, q, q_1, q_2\}$, $\Theta(a) = k$ for all a, and δ consists of the following rules:

$$(q_0, r) \to (q, a_1, \phi_{\text{root}}(\bar{x}) \wedge \phi_{a_1}(\bar{x})), \ldots, (q, a_n, \phi_{\text{root}}(\bar{x}) \wedge \phi_{a_n}(\bar{x})).$$

This rule takes the root node as defined by T and puts it with its associated label as a child of the (default) root node r. The next rule selects the first and second child of an already generated node using a virtual tag v:

$$(q, a) \to (q_1, v, \exists\bar{y}\, Reg(\bar{y}) \wedge \phi_{fs}(\bar{y}, \bar{x})), (q_2, v, \exists\bar{y}\exists\bar{z}\, Reg(\bar{y}) \wedge \phi_{fs}(\bar{y}, \bar{z}) \wedge \phi_{ns}(\bar{z}, \bar{x})).$$

The actual node corresponding to a first child with the correct label is produced by

$$(q_1, v) \to (q, a_1, Reg(\bar{x}) \wedge \phi_{a_1}(\bar{x})), \ldots, (q, a_n, Reg(\bar{x}) \wedge \phi_{a_n}(\bar{x})).$$

Finally, the rule $(q_2, v) \to (q, a_1, Reg(\bar{x}) \wedge \phi_{a_1}(\bar{x})), \ldots, (q, a_n, Reg(\bar{x}) \wedge \phi_{a_n}(\bar{x}))$, $(q_2, v, \exists\bar{y}\exists\bar{z}\, Reg(\bar{y}) \wedge \phi_{ns}(\bar{z}, \bar{x}))$ generates the node corresponding to a non-first-child and selects the following sibling. Observe that this rule is recursive. One can show that, for any instance I, $\tau(I)$ is equal to $T(I)$ rooted under an r-node.

(2) We show that, given a nonrecursive publishing transducer $\tau = (Q, \Sigma, \Theta, q_0, \delta, \Sigma_e)$ in PT$_{nr}$($\mathcal{L}$,tuple,virtual) for $\mathcal{L}$ either FO or IFP, there exists an $\mathcal{L}$-transduction T_τ such that, on any instance I, $\tau(I) = T_\tau(I)$. Moreover, the construction shows that T_τ can be assumed to be fixed-depth.

We may assume that we have constants for every state in Q and every label in Σ. Indeed, we can always simulate these by introducing registers with higher arity. Let k' be the largest arity of a register in τ and let $k = k' + 2$. We now construct a fixed-depth $\mathcal{L}$-transduction of width k as follows.

First, we observe that the first column and second column of a node (i.e., a k-tuple) will always refer to a state in Q and a label in Σ, respectively.

We now define the different formulas constituting the $\mathcal{L}$-transduction T_τ. We define ϕ_{dom} to be simply true. Moreover, for every $a \in \Sigma$, we let $\phi_a(x_1, x_2, \bar{x}) \equiv x_2 = a$. The edge relation is computed in two stages.
In the first step, we define:

$$\phi_e^1(x_1, x_2, \bar{x}; y_1, y_2, \bar{y}) \equiv \bigvee_{(q,a)\to(q_1,a_1,\phi_1),\ldots,(q_n,a_n,\phi_n)\in\delta} x_1 = q \wedge x_2 = a \wedge \left(\bigvee_{i=1}^{n} y_1 = q_1 \wedge y_2 = a_i \wedge \phi_i'(\bar{y})\right),$$

where each ϕ_i' is obtained from ϕ_i (i.e., the ith formula in the rule in δ under consideration) by replacing each $Reg(\bar{z})$ by $\bar{x} = \bar{z}$. It is easily verified that the formula ϕ_e^1 defines the correct DAG but with virtual nodes.

Therefore, in the second step, we define ϕ_e such that it skips all virtual nodes in the DAG defined by ϕ_e^1. For this we define:

$$\phi_e(x_1, x_2, \bar{x}; y_1, y_2, \bar{y}) \equiv \neg \bigwedge_{a\in\Sigma_e} \phi_a(x_1, x_2, \bar{x}) \wedge \neg \bigwedge_{a\in\Sigma_e} \phi_a(y_1, y_2, \bar{y}) \wedge \phi_e^1 * (x_1, x_2, \bar{x}; y_1, y_2, \bar{y})$$
$$\wedge \neg\exists\bar{z}(\phi_e^1 * (x_1, x_2, \bar{x}; \bar{z}) \wedge \phi_e^1 * (\bar{z}; y_1, y_2, \bar{y}) \wedge \neg \bigwedge_{a\in\Sigma_e} \phi_a(\bar{z})).$$

Here, $\phi_e^1 *$ denotes the transitive closure of ϕ_e^1, which is expressible in FO because the depth of the output tree depends on τ and not on the input structure (recall that τ is nonrecursive). Moreover, because τ is nonrecursive, there can be no path in the output tree where the same state-label pair appears twice. Hence no branch is short-circuited by the stop-condition.

It can be easily verified that for any instance I, $\tau(I) = T(I)$.

(3) We here provide an example of a recursive transducer in PT(FO, tuple, normal) that is not expressible as an FO-transduction. Let the schema consist of an edge relation E and two constants s and t. Let τ be the transducer that outputs the unfolding of E starting from s, and that stops when t or a duplicate node is reached. When t is reached, a b-labeled leaf node is returned. This can indeed be easily achieved by an PT(FO,tuple, normal)-transducer consisting of the following rules: $(q_0, r) \to (q, a, x = s)$, $(q, a) \to (q, a, \exists y\, Reg(y) \wedge E(y, x)), (q, b, \exists y\, Reg(y) \wedge y = t)$, and the rule for (q, b) has an empty RHS. However, suppose that this transformation is definable by an FO-transduction. Then the FO-sentence "there is a leaf with label b" expresses over E that there is a path from s to t, which is known not to be expressible in FO. Hence there exists no equivalent FO-transduction for τ.

(4) The containment of PT$_{nr}$(CQ,tuple,O) in fixed-depth $\mathcal{L}$-transduction can now be verified along the same lines as (2), since we do not need to express the stop condition for nonrecursive transducers, which is not definable in a monotone language. Because of this and (2), it is sufficient to show that, given a fixed-depth $\mathcal{L}$-transduction T, there exists an equivalent publishing transducer τ_T in PT$_{nr}$($\mathcal{L}$,tuple,O).

Let $T = (\phi_{\text{dom}}, \phi_{\text{root}}, \phi_e, \phi_<, \phi_{fc}, \phi_{ns}, (\phi_a)_{a\in\Sigma})$ be a fixed-depth $\mathcal{L}$-transduction. Let ℓ be the depth of the transduction. Assume that $\Sigma = \{a_1, \ldots, a_n\}$. We then define the transducer $\tau_T = (Q, \Sigma, \Theta, q_0, \delta)$, where $Q = \{q_0, q_1, \ldots, q_\ell\}$, $\Theta(a) = k$ for all a, and δ consists of the following rules. The start rule generates a_i nodes: $(q_0, r) \to (q, a_1, \phi_{\text{root}}(\bar{x}) \wedge \phi_{a_1}(\bar{x})), \cdots, (q, a_n, \phi_{\text{root}}(\bar{x}) \wedge \phi_{a_n}(\bar{x}))$. And for all $i \in [1, \ell]$

and $a \in \Sigma$, we define the rule $(q_i, a) \rightarrow (q_{i+1}, a_1, \phi_1(\bar{x})), \cdots, (q_{i+1}, a_n, \phi_n(\bar{x}))$, where $\phi_i(\bar{x})$ is $\exists \bar{y}(Reg(\bar{y})) \wedge \phi_e(\bar{y}, \bar{x}) \wedge \phi_{a_i}(\bar{x})$. It is not hard to see that, for every I, $\tau(I)$ equals $T(I)$ rooted under an r-symbol when disregarding the order of siblings.

(5) Let $T = (M_{\text{dom}}, M_{\text{root}}, M_e, M_{fc}, M_{ns}, M_<, (M_a)_{a\in\Sigma})$ be a PSPACE-transduction. Since we assume that the domain is ordered, for every PSPACE Turing machine, $\mathcal{M}$, there exists a Partial Fixed Point (PFP) sentence $\xi_{\mathcal{M}}$ over the relational schema extended with symbols X and Y such that (I, X, Y) is accepted by $\mathcal{M}$ iff $(I, X, Y) \models \xi_{\mathcal{M}}$ [Flum and Ebbinghaus 1999]. Every such sentence can be written as $\exists \bar{x} \text{PFP}(\chi)\bar{x}$, where χ is first-order and total, that is, it always reaches a fixpoint [Flum and Ebbinghaus 1999]. Although registers in transducers in PT(FO,relation,virtual) can only contain a single relation, it is easy to encode a finite number of them with one relation. For instance, all tuples where the first column contains a specific constant, correspond to one relation. Thus we may assume several registers. We can now simulate in PT(FO,relation,virtual) a formula $\exists \bar{x} \text{PFP}_Z(\chi)\bar{x}$, where χ is over X, Y, the relational schema and the recursion variable Z. Indeed, we start with $Z = \emptyset$, and we associate with each iteration step of χ a transition in the transducer that outputs a virtual node. When a fixpoint is reached, we test whether it is nonempty.

More specifically, the start rule $(q_0, r) \rightarrow (q, v, \phi(\emptyset, \bar{y}))$ tries to find the relation Y for which $Y \in M_{\text{root}}$. It hence enumerates every relation Y and tests whether $Y \in M_{\text{root}}$ by simulating the PFP-sentence. If so, the current node is output with the corresponding nonvirtual label. Relations (of arity k) can be enumerated by considering each relation as a number between 0 and 2^{n^k}. When a node has been output with a nonvirtual label and register content X, the transducer tests, for every relation Y, whether $(X, Y) \in M_{fc}$. Similarly for all next-sibling relations.

The proof that PT(IFP,tuple,virtual) contains the first-order PTIME-transductions on ordered structures is similar, exploiting the correspondence of IFP to PTIME. □

Regular tree languages. Recall that a DTD d over Σ is a mapping from Σ-symbols to regular expressions over Σ. A Σ-tree t conforms to d if and only if for each a-node v in t, the list of labels of the children of v is a string in $d(a)$. It is known that DTDs define the set of local tree languages, Relax NG corresponds to the regular tree languages, and XML Schema lies in between [Martens et al. 2006]. The class of regular tree languages is conveniently abstracted by specialized or extended DTDs [Papakonstantinou and Vianu 2000], also referred to as generalized DTDs [Maneth and Neven 1999]. An *extended* DTD D over Σ is a triple (Σ', d, μ), where $\Sigma \subseteq \Sigma'$, μ 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, $t = \mu(t')$, where μ is canonically extended from labels to trees. We denote by $L(D)$ the set of all Σ-trees conforming to D. Another characterization of the class of unranked regular tree languages is in terms of the MSO definable tree languages (e.g., [Neven and Schwentick 2002]). A tree language L 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 $L = \tau(R)$.

The next result tells us that PT(FO, S, virtual) is capable of defining all extended DTDs, and thus all regular unranked and MSO definable tree languages. In contrast, PT(CQ, S, O) does not have sufficient expressive power to define even DTDs.

THEOREM 6.5. *Every extended DTD over Σ is definable in PT(FO, tuple, virtual). There exist DTDs that are not definable in PT(CQ, relation, virtual).*

PROOF. The proof is referred to the Appendix. □

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. The results of Theorem 6.2 and Proposition 6.2 for PT(IFP, tuple, normal) and PT(FO, relation, virtual) also provide insight for the expressive power of DBMS_XMLGEN and ATG, respectively. The following result settles the issue for PT$_{nr}$(IFP, tuple, normal) (SQL_mapping,SQL/XML), PT$_{nr}$(FO, tuple, normal) (FOR-XML, XPERANTO), and PT$_{nr}$(CQ, tuple, O) (annotated XSD, RDB_mapping, TreeQL).

Denote by UCQ union of conjunctive queries extended with '$\neq$'.

PROPOSITION 6.6. *When treated as relational query languages, (1) PT$_{nr}$(CQ, tuple, O) = UCQ; (2) PT$_{nr}$(FO, tuple, O) = FO; and (3) PT$_{nr}$(IFP, tuple, O) = IFP;*

PROOF. The proof is referred to the Appendix. □

Tree generation. The proof for Proposition 6.3(1, 2) remains intact for nonrecursive transducers. As a result, PT$_{nr}$(CQ, tuple, normal) $\subset$ PT$_{nr}$(FO, tuple, normal) $\subset$ PT$_{nr}$(IFP, tuple, normal) and PT$_{nr}$(CQ, tuple, normal) $\subset$ PT$_{nr}$(CQ, tuple, virtual). Theorem 6.4 tells us that, over unordered trees, fixed-depth FO-transduction (resp. IFP-transduction) is equivalent to PT$_{nr}$(FO, tuple, O) (resp. PT$_{nr}$(IFP, 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. It is easily verified that specialized DTDs are definable in ATG [Bohannon et al. 2004].

Proposition 6.3(6) states that PT(FO, relation, virtual)=PT(IFP, relation, virtual). From a practical point of view, this implies that one does not need the linear recursion of SQL'99 to define XML views expressible in PT(IFP, relation, virtual).

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 have provided (a) a complete picture of the membership, equivalence and emptiness problems, (b) a comprehensive expressiveness analysis in terms of

Table II. Complexity of Decision Problems (S: relation or tuple; O: normal or virtual)

Fragments	Equivalence	Emptiness	Membership
PT(FP, S, O) (Prop. 5.2)	undecidable	undecidable	undecidable
PT(FO, S, O) (Prop. 5.2)	undecidable	undecidable	undecidable
PT(CQ, tuple, normal) (Th. 5.3)	undecidable	PTIME	Σ_2^p-complete
PT(CQ, relation, normal) (Th. 5.3)	undecidable	PTIME	undecidable
PT(CQ, S, virtual) (Th. 5.3)	undecidable	NP-complete	undecidable
PT$_{nr}$(FO, tuple, normal) (Th. 5.5)	undecidable	undecidable	undecidable
PT$_{nr}$(CQ, tuple, normal) (Th. 5.5)	Π_3^p-complete	PTIME	Σ_2^p-complete
PT$_{nr}$(CQ, tuple, virtual) (Th. 5.5)	Π_3^p-complete	NP-complete	Σ_2^p-complete

Table III. Expressive Power Characterized in Terms of Relational Query Languages

Fragments	Complexity class/Language
PT(IFP, relation, O) (Th. 6.1(4))	PSPACE
PT(FO, relation, O) (Th. 6.1(4))	PSPACE
PT(IFP, tuple, O) (Th. 6.1(5))	IFP, PTIME (ordered database)
PT(FO, tuple, O) (Th. 6.1(3))	LinDatalog(FO), NLOGSPACE (ordered)
PT(CQ, tuple, O) (Th. 6.1(2))	LinDatalog
PT$_{nr}$(IFP, tuple, O) (Prop. 6.6(3))	IFP
PT$_{nr}$(FO, tuple, O) (Prop. 6.6(2))	FO
PT$_{nr}$(CQ, tuple, O) (Prop. 6.6(1))	UCQ

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 relational querying power are summarized in Tables II and III, respectively, annotated with corresponding theorems and conditions (e.g., ordered). These tables show that different combinations of logic $\mathcal{L}$, store S, and output O, and 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 concerns, when treated as a relational query language, whether or not PT(CQ, relation, O) captures some relational query language (e.g.,a fragment of DATALOG). Another interesting topic is the typechecking problem for publishing transducers. Our preliminary results show that, while this is undecidable in general, there are interesting decidable cases. This issue deserves a full treatment of its own. With respect to expressiveness, we leave the following as open questions: the relationship between PT(CQ, relation, O) and PT(IFP, tuple, O) (where O = {normal, virtual}) with respect to relational expressiveness; and PT(IFP,tuple,vitrual) versus PT(IFP,relation,normal) with respect to their tree generation power.

Further, the relationship between publishing transducers and XML-to-XML transformation languages such as, for example, XSLT, is fully unexplored. In this setting, a relational database could be regarded as an XML document using a "canonical encoding." Finally, in contrast to XML publishing that deals with a single source, XML integration extracts data from multiple distributed relational sources and builds an XML tree with the extracted data. A new challenge of XML integration is introduced by dependencies on the data extracted from different sources. We plan to investigate two-way and nondeterministic publishing transducers for studying the expressive power and complexity of XML integration languages being used in practice.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in he ACM Digital Library.

ACKNOWLEDGMENT

We thank Michael Benedikt, Christoph Koch and Leonid Libkin for helpful discussions.

REFERENCES

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley.

ALON, N., MILO, T., NEVEN, F., SUCIU, D., AND VIANU, V.. 2003. Typechecking XML views of relational databases. *ACM Trans. Comput. Logic 4*, 3, 315–354.

ARENAS, M. AND LIBKIN, L. 2005. XML data exchange: consistency and query answering. In *Proceedings of the 24th ACM SIGMOD Symposium on Principles of Database Systems* (*PODS*). ACM, New York, NY, 13–24.

BENEDIKT, M., CHAN, C., FAN, W., RASTOGI, R., ZHENG, S., AND ZHOU, A. 2002. DTD-directed publishing with attribute translation grammars. In *Proceedings of the 28th International Conference on Very Large Data Bases* (*VLDB*). Morgan Kaufmann, San Francisco, CA, 838–849.

BENEDIKT, M. AND KOCH, C. 2006. Interpreting tree-to-tree queries. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming* (*ICALP*). Lecture Notes in Computer Science, vol. 4052, Springer, Berlin, Heidelberg, New York, 552–564.

BOHANNON, P., CHOI, B., AND FAN, W. 2004. Incremental evaluation of schema-directed XML publishing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). ACM, New York, NY, 503–514.

BÖRGER, E., GRÄDEL, E., AND GUREVICH, Y. 1997. *The Classical Decision Problem*. Springer.

COURCELLE, B. 1994. Monadic second-order definable graph transductions: a survey. *Theor. Comput. Sci. 126*, 1, 53–75.

DANTSIN, E., EITER, T., GOTTLOB, G., AND VORONKOV, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33*, 3, 374–425.

FAGIN, R., KOLAITIS, P., AND POPA, L. 2005. Data exchange: getting to the core. *ACM Trans. Database Syst. 30*, 1, 174–210.

FAN, W., GEERTS, F., AND NEVEN, F. 2007. Expressiveness and complexity of XML publishing transducers. In *Proceedings of the 26th ACM SIGMOD Symposium on Principles of Database Systems* (PODS). ACM, New York, NY, 83–92.

FERNANDEZ, M., KADIYSKA, Y., SUCIU, D., MORISHIMA, A., AND TAN, W. C. 2002. SilkRoute: a framework for publishing relational data in XML. *ACM Trans. Database Syst. 27*, 4, 438–493.

FLUM, J. AND EBBINGHAUS, H. 1999. *Finite Model Theory*, 2nd ed. Springer.

GÉCSEG, F. AND STEINBY, M. 1996. Tree languages. In *Handbook of Formal Languages*. Vol. 3. Springer.

GRÄDEL, E. 1992. On Transitive Closure Logic. In *Proceedings of the 5th Workshop on Computer Science Logic* (*CSL*). 149–163.

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

KLUG, A. 1988. On conjunctive queries containing inequalities. *J. ACM 35*, 1, 146–160.

KRISHNAMURTHY, R., KAUSHIK, R., AND NAUGHTON, J. 2003. XML-SQL query translation literature: the state of the art and open problems. In *Proceedings of the 1st International XML Database*

Symposium (*XSym*). Lecture Notes in Computer Science, vol. 626, Springer, Berlin, Heidelberg, New York, 1–18.

LIBKIN, L. 2003. Expressive power of SQL. *Theor. Comput. Sci. 296*, 3, 379–404.

LIBKIN, L. 2004. *Elements of Finite Model Theory*. Springer.

LUDÄSCHER, B., MUKHOPADHYAY, P., AND PAPAKONSTANTINOU, Y. 2002. A transducer-based XML query processor. In *Proceedings of the 28th International Conference on Very Large Data Bases* (*VLDB*). Morgan Kaufmann, San Francisco, CA, 227–238.

MANETH, S. AND NEVEN, F. 1999. Structured document transformations based on XSL. In *Proceedings of the 7th International Workshop on Database Programming Languages* (*DBPL*). Lecture Notes in Computer Science, vol. 1949, Springer, Berlin, Heidelberg, New York, 80–98.

MARTENS, W., NEVEN, F., SCHWENTICK, T., AND BEX, G. J. 2006. Expressiveness and complexity of XML Schema. *ACM Trans. Database Syst. 31*, 3, 770–813.

MELTON, J. AND SIMON, A. 1993. *Understanding the New SQL: A Complete Guide*. Morgan Kaufman.

MICROSOFT. 2005. XML support in microsoft SQL server 2005. *msdn.microsoft.com/library/en-us/dnsql90/html/sql2k5xml.asp/*.

MILO, T., SUCIU, D., AND VIANU, V. 2003. Typechecking for XML transformers. *J. Comput. Syst. Sci. 66*, 1, 66–97.

NEVEN, F. 2002. On the power of walking for querying tree-structured data. In *Proceedings of the 21st ACM SIGMOD Symposium on Principles of Database Systems* (*PODS*). ACM, New York, NY, 77–84.

NEVEN, F. AND SCHWENTICK, T. 2002. Query automata over finite trees. *Theor. Comput. Sci. 275*, 1–2, 633–674.

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

PAPADIMITRIOU, C. H. 1994. *Computational Complexity*. Addison Wesley.

PAPAKONSTANTINOU, Y. AND VIANU, V. 2000. DTD inference for view of XML data. In *Proceedings of the 19th ACM SIGMOD Symposium on Principles of Database Systems* (*PODS*). ACM, New York, NY, 35–46.

SHANMUGASUNDARAM, J., SHEKITA, E., BARR, R., CAREY, M., PIRAHESH, B. L. H., AND REINWALD, B. 2001. Efficiently publishing relational data as XML documents. *VLDB J. 10*, 2–3, 133–154.

SPIELMANN, M. 2000. Abstract state machines: verification problems and complexity. Ph.D. thesis, RWTH Aachen.

VAN DER MEYDEN, R. 1997. The complexity of querying indefinite data about linearly ordered domains. *J. Comput. Syst. Sci. 54*, 1, 113–135.

VARDI, M. Y. 1982. The complexity of relational query languages (extended abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing* (*STOC*). ACM, New York, NY, 137–146.

Received September 2007; revised August 2008; accepted August 2008