Expressiveness and complexity of XML publishing transducers

Citation for published version:

Digital Object Identifier (DOI):
10.1145/1412331.1412337

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Early version, also known as pre-print

Published in:
ACM Transactions on Database Systems

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Expressiveness and Complexity of XML Publishing Transducers

Wenfei Fan
Univ. of Edinburgh & Bell Labs
wenfei@inf.ed.ac.uk
and
Floris Geerts
Univ. of Edinburgh
fgeerts@inf.ed.ac.uk
and
Frank Neven
Hasselt University & Transnational Univ. of Limburg
frank.neven@uhasselt.be

A number of languages have been developed for specifying XML publishing, i.e., transformations of relational data into XML trees. These languages generally describe the behaviors of a middleware controller that builds an output tree iteratively, issuing queries to a relational source and expanding the tree with the query results at each step. To study the complexity and expressive power of XML publishing languages, this paper proposes a notion of publishing transducers 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, i.e., 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 \textit{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., \textit{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 [Information Systems]: Models and Principles—View definition languages

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: XML publishing, data exchange, transducer, complexity, expressiveness
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 $\tau$ 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 of Microsoft SQL Server 2005 [Microsoft 2005], DAD of 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”, i.e., 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 like 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]?

Example 1.1: Consider a registrar database $I_0$ of a relational schema $R_0$ consisting
of course\( (\text{cno}, \text{title}, \text{dept}) \), and \( \text{prereq}(\text{cno1}, \text{cno2}) \) (with keys underlined). The database maintains a \textit{course} relation and a relation \textit{prereq}, in which a tuple \((c_1, c_2)\) indicates that \(c_2\) is an \textit{immediate prerequisite} of \(c_1\). That is, relation \textit{prereq} gives the prerequisite hierarchy of the courses. The registrar office wants to export two XML views:

— XML view \( \tau_1 \) contains the list of all the \textit{CS courses} extracted from the database \( I_0 \). Under each \textit{course} are the \textit{cno} (number) and \textit{title} of the course, as well as its prerequisite hierarchy. As shown in Fig. 1(a), the depth of the \textit{course} sub-tree is determined by its prerequisite hierarchy.

— View \( \tau_2 \) is a tree of depth three, listing all the \textit{CS courses} as depicted in Fig. 1(b). Below each course \(c\) is a \textit{prereq} child, followed by the \textit{cno} and \textit{title} of \(c\); under \textit{prereq} is the list of all the \textit{cno’s} that appear in the prerequisite hierarchy of \(c\).

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

— XML view \( \tau_3 \) is a tree of depth two, listing all the \textit{courses} extracted from the database \( I_0 \) that do \textit{not} have dB as its immediate prerequisite. Under each \textit{course} element, its \textit{cno} and \textit{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 the 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 \textit{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 \textit{finite relation} versus those that store a single \textit{tuple};

— \( O \) (output): the types of tree nodes; in addition to \textit{normal} nodes that remain in the output tree, we may allow \textit{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 $\text{PT}(L, S, O)$ various classes of publishing transducers, where $L, S, O$ are logic, store and output parameters as specified above. As we will see later, different combinations of these parameters yield a spectrum of transducers with quite different expressive power and complexity.

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

**Characterization of existing XML publishing languages.** We examine several commercial languages and research proposals, and show that each of these languages can be embedded in some class of publishing transducers. For example, annotated xsd of Microsoft [Microsoft 2005] is a class of “nonrecursive” $\text{PT}(\text{CQ}, \text{tuple}, \text{normal})$, the for-xml constructs of Microsoft [Microsoft 2005] correspond to a class of nonrecursive $\text{PT}(\text{FO}, \text{tuple}, \text{normal})$, DBMS_XMLGEN of Oracle [Oracle ] can be expressed in $\text{PT}(\text{IFP}, \text{tuple}, \text{normal})$, and SQL/XML of IBM [IBM ] is a class of nonrecursive $\text{PT}(\text{IFP}, \text{tuple}, \text{normal})$. 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 $\text{PT}(L, S, O)$ there are existing publishing languages corresponding to them. For a few there are no corresponding commercial systems. For example, no commercial language corresponds to $\text{PT}(\text{IFP}, \text{relation}, \text{virtual})$. Our results, however, show that this class does not increase the expressive power over $\text{PT}(\text{FO}, \text{relation}, \text{virtual})$, 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 $\text{PTIME}$ to undecidable, for all the classes $\text{PT}(L, S, O)$ 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 $\text{PT}(L, S, O)$ 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 $\text{PT}(\text{IFP}, \text{relation}, \text{virtual})$ captures $\text{PSpace}$ and the smallest $\text{PT}(\text{CQ}, \text{tuple}, \text{normal})$ 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 of IBM [IBM ] is in FO and annotated xsd of Microsoft [Microsoft 2005] is in union 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 \( PT(\mathcal{L}, \text{tuple, virtual}) \) contain \( \mathcal{L} \)-transducers for \( \mathcal{L} \) ranging over CQ, FO and IFP, and that regular unranked tree languages are contained in \( PT(\text{FO, tuple, normal}) \) but not in \( PT(\text{CQ, relation, virtual}) \). Furthermore, we show the ability and inability of certain fragments of \( PT(\mathcal{L}, S, O) \) for defining DTDs and specialized DTDs of [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 \( PT(\text{IFP, relation, normal}) \) and \( PT(\text{FO, relation, normal}) \) are equivalent in the relational setting, whereas for tree generation, \( PT(\text{FO, relation, normal}) \) is properly contained in \( PT(\text{IFP, relation, normal}) \) but in contrast, \( PT(\text{FO, relation, virtual}) \) and \( PT(\text{IFP, relation, virtual}) \) 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 work differs from that lines of work in that we focus on (a) transformations from relational data to XML defined in terms of transducers with embedded relational queries, not 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., [Ludäschter 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 XSTM of [Ludäschter 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 paper 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 paper.

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

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

In particular we assume that $\Sigma$ contains a special *root tag*, denoted by $r$ unless specified otherwise, such that $\mathit{lab}(\varepsilon) = r$ and moreover, for any $v \in \mathit{dom}(t)$, $\mathit{lab}(v) \neq r$ if $v \neq \varepsilon$. To simplify the discussion we also assume a special tag, *text*, in $\Sigma$. Only leaf nodes can be labeled with *text*; they carry a string (PCDATA) and are referred to as *text* nodes. A node can have both *text* and non-*text* nodes as children.

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

We consider two classes of trees: for all $v \in \mathit{dom}(t)$, (a) either $\mathit{Reg}(v)$ stores a *finite relation* over $\mathbf{D}$, (b) or $\mathit{Reg}(v)$ 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 \( D \), 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 \( \land \) 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 \( \land \), disjunction \( \lor \), negation \( \neg \), and universal \( \forall \) and existential \( \exists \) quantifications. We also consider inflationary 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 \( \varphi \) and \(|\bar{x}| = k \), and \( t \) is a tuple of terms, where \(|t| = k \), then \( \mu^+_{\bar{x},\bar{a}}(\varphi(S,\bar{x}))(\bar{a}) \) is an IFP formula whose free variables are those in \( t \). Given an instance \( I \) of \( R \), \( I \models (\mu^+_{\bar{x},\bar{a}}(\varphi(S,\bar{x})))(\bar{a}) \) if \( \bar{a} \) is inflationary fixed point \( \mu^+ \) of the mapping \( F_\varphi : \mathcal{P}(D^k) \to \mathcal{P}(D^k) \). Here, \( \mathcal{P}(D^k) \) denotes the powerset of \( D^k \) 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 \( X \in \mathcal{P}(D^k) \). 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; \( \Sigma \) is a finite alphabet of tags; \( \Theta \) is a function from \( \Sigma \) to \( \mathbb{N} \) associating the arity of registers \( \text{Reg}_a \) to each tag \( a \); \( q_0 \) is the start state; and \( \delta \) 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 \( a \notin \{\text{text}\} \), then there is a unique rule of the form:

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

Here \( k \geq 0 \), and for \( i \in [1, k] \), \( (q_i, a_i) \in (Q \setminus \{q_0\}) \times (\Sigma \setminus \{r\}) \), and \( \phi_i(\bar{x}_i; \bar{y}_i) \in \mathcal{L} \) is a query from \( R \) and \( \text{Reg}_{a_i} \rightarrow \text{Reg}_{a_i} \), where \( \text{Reg}_a \) and \( \text{Reg}_{a_i} \) are a \( \Theta(a) \)-ary 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 \( \tau \). 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 \( a = \text{text} \), then \((q,a) \rightarrow \).

That is, its rule has an empty right-hand side (RHS).

In a nutshell, \( \tau \) is a deterministic transducer that generates a tree from a database \( I \) of schema \( R \) in a top-down fashion. Initially, \( \tau \) constructs a tree \( t \) consisting of a single node labeled \((q_0,r)\) with an empty storage. At each step, \( \tau \) expands \( t \) by simultaneously operating on the leaf nodes of \( t \). At each leaf \( u \) labeled \((q,a)\), \( \tau \) generates new nodes by finding the rule for \((q,a)\) from \( \delta \), issuing queries embedded in the rule to the relational database \( I \) and the register \( \text{Reg}_{\alpha}(u) \) 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 \( \text{Reg}_{\alpha} \) are produced as follows: The query \( \phi_j(x_i; y_i) \) extracts data from a database instance of \( R \) and from the parent register \( \text{Reg}_{\alpha} \). The result of the query is grouped by the distinct tuples corresponding to the variables in \( x_i \), 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 \( \text{Reg}_{\alpha} \). These \( a_i \) children are ordered based on an implicit ordering on the domain of data. If \( |x_i| = 0 \), then no grouping takes place and the query result is partitioned in one relation \( S_1 \) (i.e., \( m = 1 \)). If \( |y_i| = 0 \), then the result is grouped by the entire tuple and each \( S_j \) consists of single tuple only (i.e., \( |y_i| = 1 \), \( 1 \leq j \leq m \)). In general, there might be several sets \( S_j \) which might contain more than one tuple. When the result is grouped by the entire tuple (i.e., \( |y_i| = 0 \)), we refer to each register \( \text{Reg}_{\alpha} \) as a tuple register. Otherwise \( \text{Reg}_{\alpha} \) 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 \( \Sigma \)-tree, which is the output of \( \tau \).

**Transformations.** We now formally define the transformation induced by \( \tau \) from a database \( I \). As in [Alon et al. 2003], we assume an implicit ordering \( \leq \) on \( D \), which is just used to order the nodes in the output tree and, hence, get a unique output. We do not assume that the ordering is available to the query language \( \mathcal{L} \).

We extend \( \Sigma \)-trees with local storage by allowing nodes to be labeled with symbols from \( \Sigma \cup Q \times \Sigma \). We use \( \text{Tree}_{Q \times \Sigma} \) to denote the set of all such extended \( \Sigma \)-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 \( \text{Reg}_{\alpha}(u) = \emptyset \) (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 \( \Rightarrow_{\tau,f} \) as follows: \( \xi \Rightarrow_{\tau,f} \xi' \) iff there is a leaf \( u \) of \( \xi \) 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 \( \text{Reg}_{\alpha}(v) = \text{Reg}_{\alpha}(u) \), then \( \xi' \) is obtained from \( \xi \) by changing the label \((q,a)\) of \( u \) to \( a \); otherwise,
2. assume that the rule for \((q,a)\) is
   \[
   (q,a) \rightarrow (q_1,a_1,\phi_1(x_1; y_1)), \ldots, (q_k,a_k,\phi_k(x_k; y_k)).
   \]
   If \( k > 0 \), then \( \xi' \) is obtained from \( \xi \) 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 \( \{d_1, \ldots, d_n\} = \{d \mid I \cup \text{Reg}_{\alpha}(u) \models \exists y_j \phi_j(d; y_j)\} \) and \( d_1 \leq \cdots \leq d_n \) with \( \leq \) extended to tuples in the canonical way. Then \( f_j \) is a list of nodes \( v_1, \ldots, v_n \), where \( v_i \) is labeled with \((q_j,a_j)\) and its register \( \text{Reg}_{\alpha}(v_i) \) stores the relation \( \{d_i \times \{\bar{e} \mid I \cup \text{Reg}_{\alpha}(u) \models \phi_j(d_i; \bar{e})\}\} \), where
Reg\textsubscript{a} and Reg\textsubscript{a} denote the registers associated with the a-node u and the a\textsubscript{j}-node v, respectively. If all f\textsubscript{j}’s are empty, ξ’ is obtained from ξ by labeling u with a.

If k = 0, i.e., 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\textsubscript{a}(u) (assuming a function that maps relations over D to strings, based on the order ≤).

The second condition (2) states how to generate the children of the leaf u via a transduction rule. As remarked earlier, for each j ∈ [1, k], the a\textsubscript{j} children are grouped by the values d of the parameter x in the query Θ(y\textsubscript{j})(x; y\textsubscript{j}). That is, for each distinct d such that Θ(y\textsubscript{j})(d; y\textsubscript{j}) is nonempty, an a\textsubscript{j} child w is spawned from u, carrying the result of φ\textsubscript{j}(d; y\textsubscript{j}) in its local store Reg\textsubscript{a}(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\textsubscript{a}(v) of v. Since the subrooted tree at u is uniquely determined by q, a, Reg\textsubscript{a}(u) and Φ, 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, i.e., no children are spawned at u, if (a) the stop condition given above is satisfied; or (b) the query φ\textsubscript{j}(x; y\textsubscript{j}) turns out to be empty for all i ∈ [1, k] when it is evaluated on Φ and Reg\textsubscript{a}(u); in this case all the forests f\textsubscript{j} are empty; or (c) the RHS of the rule for (q, a) is empty, i.e., k = 0 in condition (2) above; this is particularly the case for a = text, as text nodes have no children. These conditions ensure the termination of the computation. Note that transduction at other leaf nodes may proceed after the transformation stops at u.

**Recursive vs. Nonrecursive transducers.** We define the dependency graph G\textsubscript{τ} of τ. For each (q, a) ∈ Q × Σ there is a unique node v(q, a) in G\textsubscript{τ}, and there is an edge from v(q, a) to v(q’, a’) iff (q’, a’) is on the RHS of the rule for (q, a). We say that the transducer τ is recursive iff there is a cycle in G\textsubscript{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 Φ rather than () for the empty sequence variables.

**Example 3.1:** The view shown in Fig. 1(a) can be defined by a publishing transducer τ\textsubscript{1} = (Q\textsubscript{1}, Σ\textsubscript{1}, Θ\textsubscript{1}, Φ\textsubscript{0}, δ\textsubscript{1}), where Q\textsubscript{1} = \{q\textsubscript{0}, q\textsubscript{1}\}, Σ\textsubscript{1} = \{db, course, prereq, cno, title, text\}, and the root tag is db; we associate six sets of registers Reg\textsubscript{db}, Reg\textsubscript{cno}, Reg\textsubscript{prereq} and Reg\textsubscript{title} with db, course, prereq, cno, title and text nodes, to which the arity-function Θ\textsubscript{1} assigns 0, 2, 1, 1, 1, respectively; finally, δ\textsubscript{1} is defined as follows:

\[
\begin{align*}
δ\textsubscript{1}(q\textsubscript{0}, db) & = (q,\text{ course, } φ\textsubscript{1}(cno, title; \emptyset)), \text{ where } \\
φ\textsubscript{1}(cno,title) & = \exists \text{ dept } (\text{ course}(cno, title, dept) \land \text{ dept } = \text{ 'CS'} ) \\
δ\textsubscript{1}(q, \text{ course}) & = (q,\text{ cno, } φ\textsubscript{1}\textsubscript{1}(\text{ cno; } \emptyset), (q,\text{ title, } φ\textsubscript{1}\textsubscript{2}(\text{ title; } \emptyset)), (q,\text{ prereq, } φ\textsubscript{1}\textsubscript{3}(\text{ cno; } \emptyset)), \text{ where } \\
φ\textsubscript{1}\textsubscript{1}(cno) & = \exists \text{ title } \text{ Reg}(cno, title), \text{ and } φ\textsubscript{1}\textsubscript{2}(title) = \exists \text{ cno } \text{ Reg}(cno, title) \\
δ\textsubscript{1}(q, \text{ prereq}) & = (q,\text{ course, } φ\textsubscript{1}(cno, title; \emptyset), \text{ where }
\end{align*}
\]
\[ \phi_3(c, t) = \exists c' d \ (\text{Reg}_p(c') \land \text{prereq}(c', c) \land \text{course}(c, t, d)) \]

\[ \delta_1(q, \text{cno}) = (q, \text{text}, \phi_4(\text{cno}; \emptyset)), \text{ where } \phi_4(c) = \text{Reg}_p(c) \]

/* similarly for \( \delta_1(q, \text{title}) */

\[ \delta_1(q, \text{text}) = . \]

/* empty RHS */

In each query \( \phi(x; y) \) in the rules, \(|y| = 0, \) i.e., \( y = \emptyset \). Thus \( \text{Reg}_d, \text{Reg}_p, \text{Reg}_q, \) and \( \text{Reg}_\text{text} \) are tuple registers. The semantics of \( \tau_1 \) is as follows: Given an instance \( I_0 \) of the schema \( R_0 \) described in Example 1.1, the publishing transducer \( \tau_1 \) 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 \( \phi_1 \) on \( I_0 \), and for each distinct tuple in the result, it spawns a course child \( v \) carrying the tuple in its register \( \text{Reg}_c(v) \). At node \( v \) it issues queries \( \phi_2^1 \) and \( \phi_2^2 \) on \( \text{Reg}_c(v) \), and spawns its \( \text{cno}, \text{title} \), and \( \text{prereq} \) children carrying the corresponding tuple in their registers.

At the \( \text{cno} \) child, it simply extracts the string value of \( \text{cno} \) and the transformation stops; similarly for \( \text{title} \). At the \( \text{prereq} \) child \( u \), it issues query \( \phi_3 \) against both \( I_0 \) and \( \text{Reg}_p(u) \); i.e., it extracts all immediate prerequisites of the course of node \( v \), for which the \( \text{cno} \) is stored in \( \text{Reg}_p(u) \). In other words, the \( \text{cno} \) information passed down from node \( v \) is used to determine the children of \( u \). For each distinct tuple in the result of \( \phi_3 \), it generates a course child of \( u \). The transformation continues until either it reaches some course for which there is no prerequisite, i.e., \( \phi_3 \) returns empty at its \( \text{prereq} \) child; or when 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 \( \Sigma \)-tree of the form depicted in Fig. 1(a).

Note that the transformation is data-driven: the number of children of a node and the depth of the XML tree are determined by the relational database \( I \). Note also that \( \tau_1 \) is recursive: \( G_{\tau_1} \) contains the cycle \( (v(q, \text{course}), v(q, \text{prereq})), (v(q, \text{prereq}), v(q, \text{course})) \).

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

**Virtual versus normal nodes.** We also consider a class of publishing transducers with virtual nodes. Such a transducer is of the form \( \tau = (Q, \Sigma, \Theta, q_0, \delta, \Sigma_v) \), where \( \Sigma_v \) is a designated subset of \( \Sigma \), referred to as the virtual tags of \( \tau \); and \( Q, \Sigma, \Theta, q_0, \delta \) are the same as described in Definition 3.1. We require that \( \Sigma_v \) does not contain the root tag. On a relational database \( I \) the transducer \( \tau \) behaves the same as a normal transducer, except that the \( \Sigma \)-tree \( \tau(I) \) is obtained from the result \( \xi \) of the \( \tau \)-transformation on \( I \) as follows. First, the local registers and states are removed from \( \xi \). Second, for each node \( v \) in \( \text{dom}(\xi) \), if \( v \) is labeled with a tag in \( \Sigma_v \), we shortcut \( v \) by replacing \( v \) with \( \text{children}(v) \), i.e., treating \( \text{children}(v) \) as children of the parent of \( v \), and removing \( v \) from the tree. The process continues until no node in the tree is labeled with a tag in \( \Sigma_v \).
Example 3.2: Suppose that we want to define a publishing transducer for the XML view shown in Fig. 1(b), and that the query language \( \mathcal{L} \) is FO. One can show, via a simple argument using 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 \( \tau_2 \) with virtual nodes. Indeed, capitalizing on a virtual tag \( l \), we give some of the transduction rules \( \delta_2 \) of \( \tau_2 \) as follows:

\[
\delta_2(q_0, \text{db}) \text{ and } \delta_2(q, \text{course}) \text{ are the same as } \delta_1(q_0, \text{db}) \text{ and } \delta_1(q, \text{course}) \text{ in Example 3.1}
\]

\[
\delta_2(q, \text{prereq}) = (q, l, \varphi_1(\emptyset; \text{cno})), \text{ where } \varphi_1(c) = \exists c' \ (\text{Reg}_c(c') \land \text{prereq}(c', c))
\]

\[
\delta_2(q, l) = (q, l, \varphi_1(\emptyset; \text{cno})), \ (q, \text{cno}, \varphi_2(\text{cno}; \emptyset)), \text{ where } \varphi_2(c) = \text{Reg}_c(c) \lor \exists c' \ (\text{Reg}_c(c') \land \text{prereq}(c', c)), \ \varphi_2(c) = \varphi_1(c) \land \forall c' \ (\text{Reg}_c(c') \leftrightarrow \varphi_1(c'))
\]

In \( \varphi_1 \) and \( \varphi_2 \), \( |\vec{x}| = 0 \) and thus the result of \( \varphi_1 \) and \( \varphi_2 \) is put in a single relation, stored in the register \( \text{Reg}_v(l) \) of the \( l \) child \( v \). In contrast, \( |\vec{y}| = 0 \) in \( \varphi_2 \) 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 \( \text{cno} \) child is generated.

Intuitively, for each course \( c \) the transducer \( \tau_2 \) recursively finds \( \text{cno} \)'s in the prerequisite hierarchy of \( c \) and adds these \( \text{cno} \)'s to the relation \( \text{Reg}_v(l) \) 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 non-empty set \( \text{Reg}_v(l) \). For each \( \text{cno} \) in the set, a distinct \( \text{cno} \) node is created. Then, all the nodes labeled \( l \) are removed and those \( \text{cno} \) nodes become the children of \( c \). Thus \( \tau_2 \) induces the XML view of Fig. 1(b). \( \square \)

Fragments. We denote by \( \text{PT}(\mathcal{L}, S, O) \) various classes of publishing transducers. Here, \( \mathcal{L} \) indicates the relational query language in which queries embedded in the transducers are defined. We consider \( \mathcal{L} \) ranging over conjunctive queries with ‘\( \not= \)’ (CQ), first-order logic (FO) and (inflationary) fixpoint logic (IFP), all with equality ‘\( = \)’. Store \( S \) is either relation or tuple, indicating that the \( \Sigma \)-trees induced by the transducers are with relation or tuple stores, respectively. Observe that transducers with tuple stores are a special case of those with relation stores. More specifically, for any transducer \( \tau \) with tuple stores, \( |\vec{y}| = 0 \) in each query \( \phi_1(x; \vec{y}) \) in \( \tau \), as illustrated in Example 3.1. Output \( O \) is either normal or virtual, indicating whether a transducers allow virtual nodes or not. Thus \( \text{PT}(\text{IFP}, \text{reltionship, virtual}) \) is the largest class considered in this paper, which consists of transducers that are defined with fixpoint-logic queries and generate trees with relation stores and virtual nodes. In contrast, \( \text{PT}(\text{CQ}, \text{tuple, normal}) \) is the smallest.

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

For instance, the transducers \( \tau_1 \) and \( \tau_2 \) given in Examples 3.1 and 3.2 are in \( \text{PT}(\text{CQ}, \text{tuple, normal}) \) and \( \text{PT}(\text{FO}, \text{relation, virtual}) \), respectively (\( \tau_2 \) is also definable in \( \text{PT}_n(\text{IFP}, \text{tuple, normal}) \); we omit this definition for 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 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 $I_0$ 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 Fig. 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, i.e., 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 $\mathcal{PT}_{nr}(\text{FO}, \text{tuple}, \text{normal})$.

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 $\mathcal{PT}_{nr}(\text{CQ}, \text{tuple}, \text{normal})$.

**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 XML_ELEMENT, XML_ATTRIBUTE, XML_FOREST, XML_CONCAT, XML_AGG and XML_GEN. It extracts relational data in parallel with XML-element creation. Nested queries are used to generate a hierarchical XML tree, during which a node can pass information to its children via correlation. The tree has a fixed depth bounded by the level of query nesting, and has no virtual nodes. Although only non-recursive XML trees can be
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
<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> "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.

dbmsxmlgen.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 10g XML DB.

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 PTnr(IFP, tuple, normal). For instance, Figure 3 shows the view of Fig. 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 Fig. 2, e.g., can be expressed in terms of SQL MAPPING as shown in Fig. 4.

The latter embeds nested RDB NODE expressions in a DAD. The DAD is basically a tree template with a fixed depth, and those embedded expressions are essentially CQ queries for populating elements and attributes specified in the DAD.

Neither of these two allows virtual nodes. One can express DAD with SQL MAPPING in PTnr(IFP, tuple, normal), and RDB MAPPING in PTnr(CQ, tuple, normal).

Oracle 10g XML DB [Oracle]. Oracle supports SQL/XML as described above, and a PL/SQL package DBMS_XMLGEN. DBMS_XMLGEN extends SQL/XML by supporting the linear recursion construct CONNECT-BY (SQL’99), and is thus capable of defining recursive XML views. Given a relational source, an XML tree of an unbounded depth is generated top-down, along the same lines as nested SQL/XML queries. Information is passed from a node to its children via connect-by joins. For each tuple resulted from the joins, a child node is created, whose children are in turn created in the next iteration of the fixpoint computation. For example, Figure 5 shows a recursive XML view, containing the list of all courses; under each course c are the cno and title of c followed by the hierarchy of the prerequisite courses of c.
db → course

$course = \text{select} \ cno, \ title \ \text{from} \ \text{course}

course → cno, \ title, \ prereq

$prereq = \text{select} \ cno \ \text{from} \ \text{course}; \ \text{similarly for} \ \$cno \ \text{and} \ \$title

prereq → course

$course = \text{select} \ c.cno, \ c.title \ \text{from} \ \text{prereq} \ p, \ \$prereq \ cp, \ \text{course} \ c

\text{where} \ cp.cno = p.cno1 \ \text{and} \ p.cno2 = c.cno;

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

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]. It 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 $\alpha$, 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 $d_0$ and is required to conform to a DTD $d_0$. It list 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).

The classification is summarized in Table I, which, for each publishing language mentioned above, gives the “smallest” class of publishing transducers that can express all XML views definable in the language. Except 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 3.1 and 3.2 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
Table 1. Characterization of existing XML publishing languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Publishing transducers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft SQL Server 2005</td>
<td>PT_{IFP}(FO, tuple, normal)</td>
</tr>
<tr>
<td></td>
<td>PT_{IFP}(CQ, tuple, normal)</td>
</tr>
<tr>
<td>IBM DB2 XML Extender</td>
<td>PT_{IFP}(IFP, tuple, normal)</td>
</tr>
<tr>
<td></td>
<td>PT_{IFP}(CQ, tuple, normal)</td>
</tr>
<tr>
<td>Oracle 10g XML DB</td>
<td>PT_{IFP}(FO, tuple, normal)</td>
</tr>
<tr>
<td></td>
<td>PT(IFP, tuple, normal)</td>
</tr>
<tr>
<td>XPERANTO</td>
<td>PT_{IFP}(FO, tuple, normal)</td>
</tr>
<tr>
<td>TreeQL</td>
<td>PT_{IFP}(CQ, tuple, virtual)</td>
</tr>
<tr>
<td>ATG</td>
<td>PT(FO, relation, virtual)</td>
</tr>
</tbody>
</table>

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 \Sigma-tree t and a transducer \tau in this class, whether or not there exists an instance I such that t = \tau(I), i.e., \tau evaluated on I returns the tree t.

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

The equivalence problem for PT(\mathcal{L}, S, O) is to determine, given two transducers \tau_1 and \tau_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, i.e., the two transducers produce the same \Sigma-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 1. Let \tau be a publishing transducer in PT(\mathcal{L}, S, O). Let I be an instance.

(1) The \tau-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 \tau_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 \tau_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 2.** The membership, emptiness and equivalence problems are undecidable for \( \mathcal{PT}(\mathcal{L}, S, O) \) 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 \( \mathcal{PT}_{\text{nr}}(\text{FO, tuple, normal}) \). From this the theorem immediately follows.

We show the undecidability by reduction from the equivalence problem for relational \( \mathcal{FO} \) queries, which is to determine, given any \( \mathcal{FO} \) 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 (cf. [Abiteboul et al. 1995]).

Given any \( \mathcal{FO} \) queries \( Q_1, Q_2 \), we use \( \Delta Q \) to denote their symmetric difference \( (Q_1 \setminus Q_2) \cup (Q_2 \setminus Q_1) \). Obviously, \( Q_1 \equiv Q_2 \) iff \( \Delta Q(I) = \emptyset \) for all instances \( I \) of \( R \).

The membership problem. The reduction consists of a transducer \( \tau_0 \) in \( \mathcal{PT}_{\text{nr}}(\text{FO, tuple, normal}) \) and a tree \( t_0 \) such that \( t_0 \in \tau_0(R) \) iff \( Q_1 \not\equiv Q_2 \). 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 \( \delta_0 \) is given as follows, from which \( \Theta_0 \) is clear:

\[
\delta_0(q_0, x) = (q, a, \phi(x; \emptyset)), \quad \text{where } \phi(x; \emptyset) = \exists \bar{s}(\Delta Q(\bar{s}) \land x = 'c') \text{ and } |ar{s}| \text{ is the same as the arity of the result of } Q_1 \text{ and } Q_2;
\]

\[
\delta_0(q, a) = (q, a, \phi_0(x; \emptyset)), \quad \text{where } \phi_0(x; \emptyset) = (x = 'c') \land \neg(x = 'c'), \text{ i.e., it is a query that returns the empty set on any database instance.}
\]

Then obviously, if \( t_0 \) is in \( \tau_0(R) \) then there must exist an instance \( I \) of \( R \) such that \( \phi(I) \) is nonempty. Hence, \( \Delta Q(I) \) is nonempty, i.e., \( Q_1 \not\equiv Q_2 \). 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 \( \tau_1 \) in \( \mathcal{PT}_{\text{nr}}(\text{FO, tuple, normal}) \) over \( R \) such that \( \tau_1(R) \) consists of a single-node tree iff \( Q_1 \equiv Q_2 \). We define \( \tau_1 = (Q_1, \Sigma_1, \Theta_1, q_0, \delta_1) \) to be the same as \( \tau_0 \) except \( \delta_1 \). Here we define \( \delta_1(q_0, r) = (q, a, \phi(x; \emptyset)) \), where \( \phi(x; \emptyset) = \Delta Q(x) \), and \( \delta_1(q, a) \) to be the same as \( \delta_0(q, a) \). Then obviously, \( \tau_1(R) = \{ r \} \) iff \( \Delta Q(I) = \emptyset \) for all instances \( I \) of \( R \), i.e., \( Q_1 \equiv Q_2 \).

The equivalence problem. Given \( Q_1, Q_2 \), we construct \( \tau_2^1, \tau_2^2 \) in \( \mathcal{PT}_{\text{nr}}(\text{FO, tuple, normal}) \) over \( R \) such that for any instance \( I \) of \( R \), \( \tau_2^1(I) = \tau_2^2(I) \) iff \( Q_1 \equiv Q_2 \).

For \( i \in [1, 2] \) we define \( \tau_2^i \) to be the same as \( \tau_0 \) except \( \delta_2^i \), given as follows:

\[
\delta_2^1(q_0, r) = (q, a, \phi(x; \emptyset)), \quad \text{where } \phi(x; \emptyset) = Q_1(x);
\]

\[
\delta_2^1(q, a) = (q, text, \phi_1(x; \emptyset)), \quad \text{where } \phi_1(x; \emptyset) = \text{Rel}_a(x).
\]

Obviously, for each instance \( I \) of \( R \), \( \tau_2^1(I) = \tau_2^2(I) \) iff \( Q_1(I) = Q_2(I) \). This is because for each tuple in \( Q_1(I) \), 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 \( \tau_2^1 \equiv \tau_2^2 \) iff \( Q_1 \equiv Q_2 \). □

The situation gets slightly better when considering conjunctive queries.
Theorem 1. 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 \( \Sigma^p_2 \)-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 a bit long. In particular, for the membership problem we provide three proofs: one for the \( \Sigma^p_2 \)-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 5, 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 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 \( \tau \) in PT(CQ, S, normal) defined on a relational schema \( R \), consider the start rule \( (q_0, r) \rightarrow (q_1, a_1, \phi_1(x_1; y_1)), \ldots, (q_k, a_k, \phi_k(x_k; y_k)) \). It is obvious that \( \tau(R) \) contains a nontrivial tree iff one of the \( \phi_i \)'s 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 \( \phi_i \), based on the equalities in \( \phi_i \); and then check within each equivalence class whether it contains (i) two distinct constants, (ii) a constant \( c \) and variable \( x \) for which \( x \neq c \) is in \( \theta \), or (iii) two variables \( x \) and \( y \) for which \( x \neq y \) is in \( \theta \). One can show that none of these cases occurs iff \( \tau \) and therefore, \( \phi_i \), 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 (cf. [Papadimitriou 1994]). An instance of 3SAT is a well-formed Boolean formula \( \varphi = C_1 \land \cdots \land C_n \), 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 \( \ell_1^i \lor \ell_2^i \lor \ell_3^i \), where \( \ell_j^i \) is either \( x_k \in X \) or \( \neg x_k \). Given such a \( \varphi \), 3SAT is to determine the satisfiability of \( \varphi \).

Given \( \varphi \), we define a relational schema \( R \) and a transducer \( \tau_\varphi \) in PT(CQ, tuple, virtual) over \( R \) such that \( \varphi \) is satisfiable iff there exists an instance \( I \) of \( R \) such that \( \tau_\varphi(I) \) is non-empty. 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 \( \varphi \). 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 \( \delta_0 \) are given below.

\[(q_0, v) \rightarrow (q_1, v, \psi_0(x; y) = R(\bar{x})).\]

\[(q_{i-1}, v) \rightarrow (q_i, v, \psi_i(x_i; y) = \text{Reg}_v(x) \land (x_j = t_i^j[1] \land x_k = t_i^j[2] \land x_t = t_i^j[3])),\]

\[\ldots, (q_n, v, \psi_n(x; y) = \text{Reg}_v(x) \land (x_j = t_n^j[1] \land x_k = t_n^j[2] \land x_t = t_n^j[3])),\]

\[(q_0, v) \rightarrow (q_t, a, \psi_1 = \text{Reg}_v(\bar{x})).\]
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 iff the register of the current node \( u \) is a truth assignment that makes \( C_i \) true. For instance, suppose that \( C_i = \bar{x}_j \lor \bar{x}_k \lor \bar{x}_l \) and denote by \( T(C_i) \) the set of truth assignments of \( \bar{x}_j, \bar{x}_k \) and \( \bar{x}_l \) that make \( C_i \) true, i.e., \( T(C_i) = \{(1,0,0), (1,0,1), (1,1,0), (1,1,1), (0,0,1), (0,1,1), (0,0,0)\} \). Note that there are at most 8 tuples \( T(C_i) \), denoted by \( t_1^i, \ldots, t_8^i \). Then \( \psi_j^i \) checks whether \( t_j^i \) is in \( I_X \). If \( \text{Reg}(u) \) is such a truth assignment, then the rule spawns a \( v \)-child of \( u \) and copies the content of \( \text{Reg}(v) \) to \( \text{Reg}(u) \). Hence, \((q_n, v)\) is reached iff \( I_X \) is a truth assignment that satisfies all \( n \) clauses of \( \varphi \). 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 iff \( I_X \) contains a truth assignment satisfying \( \varphi \).

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

Let \( S_i \) be the relation schema of \( Q_i \). We assume w.l.o.g. that \( Q_n(\bar{x}) = \exists \bar{x}_1 \cdots \bar{x}_k \bigwedge_{i=1}^{k_1} R_1(\bar{x}_1) \land \cdots \land \bigwedge_{j=k_1+1}^{k} \alpha_j(\bar{x}_j) \land H_n(\bar{x}) \land C_n(\bar{x}, \bar{x}_1, \ldots, \bar{x}_k) \), 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 \( S_n \). \( H_n \) 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 \( Q^{n-1}(\bar{x}) \) for \( R_1(\bar{x}) \) in \( Q_n \), for \( i \in [1, k_1] \). We construct the \( \text{CQ} \) query \( Q_n(\bar{x}) \) and the conjunction of constraints \( H_n(\bar{x}) \) (whose purpose is explained below) inductively. For \( n = 1 \) we let \( Q_1(\bar{x}) = Q_1(\bar{x}) \) and let \( H_1(\bar{x}) \) be the completion of \( H_1(\bar{x}) \) w.r.t. \( C_1 \). That is, we complete \( H_1(\bar{x}) \) with all (in)-equalities on \( x \) inferred from \( H_1 \) and \( C_1 \). For \( n > 1 \), we define \( Q^n(\bar{x}) = \exists \bar{x}_1 \cdots \bar{x}_k \bigwedge_{i=1}^{k_1} (R_1(\bar{x}_1) \land H_{n-1}(\bar{x}_1)) \land \bigwedge_{j=k_1+1}^{k} \alpha_j(\bar{x}_j) \land H_n(\bar{x}) \land C_n(\bar{x}, \bar{x}_1, \ldots, \bar{x}_k) \) and let \( H_n(\bar{x}) \) be the completion of \( H_n(\bar{x}_i) \) w.r.t. \( C_n \) and \( H_{n-1}(\bar{x}_i) \), for \( i \in [1, k_1] \). Then we have the following.

**Claim 1.** \( Q^n \) is satisfiable iff \( Q^n \) 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 \( t \), if \( t \in Q^n(I) \) then \( H_n(t) \) holds. The case \( n = 1 \) is trivial. Let \( n > 1 \). Note that \( Q^n \) is obtained from \( Q_n \) by replacing \( R_1(\bar{x}) \) by \( Q^{n-1}(\bar{x}) \). By the induction hypothesis \((a_{n-1})\) we can obtain an equivalent query by replacing \( R_1(\bar{x}) \) by \( Q^{n-1}(\bar{x}) \) \( \land H_{n-1}(\bar{x}) \). Hence, any \( t \in Q^n(I) \) satisfies all constraints inferred by \( H_n(\bar{x}), H_{n-1}(\bar{x}) \) and \( C_n \), and therefore \( H_n(t) \) 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 $J_i$ be the instances of $R_i$ $(i > 1)$ in $S_n$ (as given by $I$). By $(a_{n-1})$ we have that $Q^n(J) \neq \emptyset$, as desired.

Conversely, we show another property, denoted by $(b_n)$: if $Q_n$ and $Q^{n-1}$ are satisfiable and $H_n(t_i)$ 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 $t_i$ that satisfies $H_n(t_i)$ we can find $k$ source tuples $s_{i1}, \ldots, s_{ik}$ such that $C_n(t_i, s_{i1}, \ldots, s_{ik})$ holds, and moreover, $H_{n-1}(s_{i1})$ holds for $i \in [1, k_1]$ by the definition of $H_n$. 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 $\{s_{i1} | j \in [1, k_1], i \in [1, \ell]\} \subseteq Q^{n-1}(J_1)$. Let $J_i = \{s_{ij} | j \in [1, \ell]\}$ and $J = (J_1, \ldots, J_{\ell})$. 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 $Q_n$ is satisfiable, then so is $Q_n$, and there exist $\bar{t}_i$'s satisfying $H_n$; thus by $(b_n)$, $Q^n$ is satisfiable.

Note that $H_i$ 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 $O(|\bar{Q}_i|^2)$. Then from the proof of Theorem 1 (1), it follows that the satisfiability of $\bar{Q}_i$ can be decided in $\text{PTIME}$.

(2) The membership problem.

**PT(CQ, tuple, normal).** We first show that the problem is $\Sigma^p_2$-hard, and then give a $\Sigma^p_2$-algorithm for deciding the membership of PT(CQ, tuple, normal).

**Lower bound:** We show the $\Sigma^p_2$ lower bound by reduction from the $\exists \forall \neg \exists$-3SAT problem, which is known to be $\Sigma^p_2$-complete (cf. [Papadimitriou 1994]). The latter problem is to determine, given $\varphi = \exists \forall Z C_1 \land \cdots \land C_r$, whether or not $\varphi$ evaluates to true. Here $Y = \{y_1, \ldots, y_n\}$ and $Z = \{z_1, \ldots, z_k\}$, and $3Y$ is a shorthand for $\exists y_1 \ldots \exists y_n$; similarly for $\forall Z$. The clauses $C_1 \land \cdots \land C_r$ is an instance of $3\text{SAT}$ given above, in which each literal is either a variable in $Y \cup Z$ or a negation thereof.

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

(a) The relational schema $R$ consists of a unary relation $RC(B)$ and a ternary relations $\text{ROR}$. We shall use the instance $I_C = \{0, 1\}$ of $RC$ to construct the Cartesian product $\text{I}_R = \times_{i \in [1, n]} I_C$, to encode the existential quantification: there exists a tuple $t_\varphi \in \text{I}_R$, i.e., a truth assignment for $Y$, such that $t_\varphi$ satisfy $\varphi = \forall Z C_1 \land \cdots \land C_r$.

An instance of $\text{I}_\varphi$ of $\text{ROR}$ consists of $\{(0, 0, 0), (1, 0, 1), (0, 1, 0), (1, 1, 1)\}$ and encodes disjunction. This is needed since CQ does not allow disjunction.

(b) To define $\tau_\varphi$, observe that $\varphi = \exists Y \varphi_1 \land \cdots \land \varphi_j$ 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 $\varphi_j$'s are true. We express $\varphi_1 \land \cdots \land \varphi_j$ 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 $b$ of length $l$, let $\psi^{b}_j(Y) = \exists x_1, x_2, x_3, s \text{ROR}(x_1, x_2, x_3, s) \land \text{ROR}(s, x_3, 1) \land \bigwedge_{i=1}^{3} \theta_i^{b}(x_i)$, where $\theta_i^{b}(x_i) = y_p$ (resp. $\theta_i^{b}(x_i) \neq y_p$) if the $i$th literal in $C_j$ is $y_p \in Y$ (resp. $\bar{y}_p$) and $\theta_i^{b}(x_i) = b[i]$ otherwise. Let $\psi_j = \bigwedge_{b} Q_j^{b}$, where $b$ ranges over all possible truth assignments of the universally quantified variables in $C_j$. Since $l \leq 3$, there are at most 8 such assignments. Now we define $\psi(Y) = \bigwedge_{j=1}^{y} Q_j$. It is easily verified that $\psi(Y)$ is satisfiable iff $\varphi$ is true.

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

\[
(q_0, r) \to (q_1, b, \phi_1(x; \emptyset)), (q_1, c, \phi_3(x; \emptyset)), (q_1, d, \phi_3(x; \emptyset)),
\]

where

\[
\phi_1(x; \emptyset) \equiv RC(0) \land RC(1) \land R_\text{OR}(0, 0, 0) \land R_\text{OR}(0, 1, 1) \land R_\text{OR}(1, 1, 1) \land x = 1
\]

\[
\phi_2(x; \emptyset) \equiv RC(x) \land x \neq 0 \land x \neq 1,
\]

\[
\phi_3(x; \emptyset) \equiv \exists Y (\bigwedge_{j=1}^m RC(y_j) \land \psi(Y)) \land x = 1
\]

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 \(RC\), and that \(I_\text{OR}\) is contained in the instance of \(R_\text{OR}\) (which is not necessarily \(I_\text{OR}\)). The formula \(\phi_2(x; \emptyset)\) checks whether instances of \(RC\) have Boolean values only. By not including a \(c\) node in \(t_\varphi\), \(\phi_2\) and \(\phi_1\) assure that any instance of \(RC\) is precisely \([0, 1]\). The formula \(\phi_3(x; \emptyset)\) computes all truth assignment of \(\varphi\), and checks whether any of these satisfies \(\psi(Y)\). It is easy to verify that \(\varphi\) is true iff there exists an instance \(I\) of \(R\) such that \(\tau_\varphi(I) = t_\varphi\).

**Upper bound:** We next provide a \(\Sigma_2^p\) algorithm that, given a transducer \(\tau\) in \(PT(CQ, \text{tuple, normal})\) 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 \(\tau\) be a transducer in \(PT(CQ, \text{tuple, normal})\) 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 \(\tau\).

**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 \(R\). In case the schema consists of more than one relation, one can simulate these with a single relation, and change the \(CQ\) queries in \(\tau\) accordingly and obtain in this way an equivalent transducer, albeit on a different schema.

Let \(v\) be an arbitrary node in \(t\), and \((q, a) \to (q_1, a_1, \phi_1(x_1; \emptyset)), \ldots, (q_k, a_k, \phi_k(x_k; \emptyset))\) be the rule in \(\tau\) that is used to generate \(v\). More specifically, assume that \(v\) is generated by \(\phi_1\) and hence has label \(a_1\). Since \(\tau\) has a tuple store, each \(v\) is associated with a distinct tuple (stored in \(Reg_\alpha(v)\)) from the result of the \(CQ\) query \(\phi_1\) on \(I\) and \(Reg_\alpha(u)\), where \(u\) is the parent of \(v\) and \(Reg_\alpha(u)\) is the tuple register of \(u\).

If we express \(\phi_1\) as an SPC (selection, projection and Cartesian product) query in the normal form, it is clear that \(Reg_\alpha(v)\) comes from the Cartesian product of at most \(k\) tuples in \(I\), where \(k\) is the number of \(R\)’s (perhaps renamed) involved in \(\phi_1\) and thus is determined by the size of \(\phi_1\). 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 \(\tau\) are \(CQ\) queries with \(\varphi \neq \emptyset\) 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 \(\tau\).

Given a transducer \(\tau\) in \(PT(CQ, \text{tuple, normal})\) 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 \(U\) of \(I\) consist of the constants appearing in any \(CQ\) query embedded in the rules of \(\tau\) plus a set of \(K|t|\) other arbitrary constants. It is easily verified that for any other such domain \(U'\) and bijective mapping \(f : U \to U'\) such that \(f(a) = a\) for any constant \(a\) appearing in queries in \(\tau\), \(\tau(I) = \tau(f(I))\). Hence, we can choose \(U\) 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 $(q, a) \rightarrow (q_1, a_1, \phi_1(x;\emptyset)), \ldots, (q_k, a_k, \phi_k(x;\emptyset))$, which is unique as $\tau$ 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, i.e., 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 $\tau$ 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 non-deterministic polynomial time Turing machine with a NP-oracle. Hence, this algorithm is in NP$^\text{NP} = \Sigma_p^2$.

**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 2-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, $\text{E+TC}$, is undecidable over a schema having at least two non-nullary relation schemas, one of them being a function symbol. Although $\text{E+TC}$ 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 readers’ convenience we include the necessary definitions taken from [Spielmann 2000]. A deterministic finite 2-head automaton (or 2-head DFA for short) is a quintuple $A = (Q, \Sigma, \Delta, q_0, q_{acc})$ 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 $\Delta : Q \times \Sigma^* \times \Sigma \rightarrow Q \times \{0, +1\} \times \{0, +1\}$, where $\Sigma^* = \Sigma \cup \{\varepsilon\}$.

A configuration of $A$ is a triple $(q, w_1, w_2) \in Q \times \Sigma^* \times \Sigma^*$, representing that $A$ is in state $q$ and the first and second head of $A$ are positioned on the first symbol of $w_1$ and $w_2$, respectively. On an input string $w \in \Sigma^*$, $A$ starts from the initial configuration $(q_0, w, w)$; the successor configuration is defined as usual. The 2-head DFA $A$ accepts $w$ if it can reach a configuration $(q_{acc}, w_1, w_2)$ from the initial configuration for $w$; otherwise $A$ rejects $w$. The language accepted by $A$ is denoted by $L(A)$. The emptiness problem for 2-head DFA’s is to determine, given a 2-head DFA $A$, whether $L(A)$ is empty or not.

Given a 2-head DFA $A = (Q, \Sigma, \delta, q_0, q_{acc})$, we define a schema $R$, a transducer $\tau_A$ in PT(CQ, tuple, virtual) and a tree $t_A$ such that $t_A \in \tau_A(R)$ iff $L(A)$ 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 $\tau_A$ and the tree $t_A$ 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 $\tau_A$ and tree $t_A$ as follows. First, to assure that instances of $P$ and $\bar{P}$ are disjoint, we add the rule $(q_0, r) \rightarrow (q, a_1, \phi_1 = \exists x P(x) \land \neg \bar{P}(x))$ and by not including an $a$-child of the root in $t_A$. Second, we assure the properties (ii)-(iv) above on instances $I_F$ of $F$. This is achieved by adding $(q, a_2, \phi_2 = \exists y F(0, y)), (q, a_3, \phi_3(x, y; 0) = F(x, y) \land x = y)$ and $(q, a_4, \phi_4 = \exists x, y, z F(x, y) \land F(x, z) \land y \neq z)$ to the RHS of the start rule given above, and by adding a single $a_2$ and $a_3$-child to the root of $t_A$. We do not include an $a_2$-child of the root to $t_A$.

Then for $\tau_A$ and $t_A$ defined so far, $\tau_A(I) = t_A$ iff $I$ is a well-formed instance of $R$.

Before we continue with the definition of $\tau_A$ and $t_A$ we show, following [Spielmann 2000], how non-emptiness of $L(A)$ can be expressed in terms of an $E + TC$-formula over $R$. Consider a transition $\delta \in \Delta$ of the form $\delta = (q, i n_1, i n_2) \rightarrow (q', m o v e_1, m o v e_2)$. This can be encoded by means of the conjunctive query

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

where $\alpha_i(x) = \exists y F(x, y) \land x \neq y \land P(x)$ if $i_n = 1$; $\alpha_1(x) = \exists y F(x, y) \land x \neq y \land \bar{P}(x)$ if $i_n = 0$; and $\alpha_i(x) = F(x, y)$ if $i_n = \varepsilon$. Moreover, $\beta_1(x, y) = F(x, y)$ if $m o v e_1 = +1$ and $\beta_1(x, y) = x = y$ if $m o v e_1 = 0$. 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 $A$ makes a move (with head $i$) and $x = y$ otherwise. Then $\Phi = \exists y_1 \exists y_2 \exists q_1 \exists q_2 \exists x, y, x', y', z' \bigwedge_{\delta \in \Delta} \varphi_\delta(q_0, 0, 0, q_{acc}, y_1, y_2)$ is satisfiable iff $L(A) \neq \emptyset$.

The transducer $\tau_A$ simulates the transitive closure (TC) in $\Phi$ 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 $t_A$ we then can test whether $\Phi$ is satisfiable, or equivalently, whether $L(A) \neq \emptyset$.

To do so, we first initialize the TC-computation by extending the RHS of the start rule of $\tau_A$ with $(q, v, \kappa_0(q, x, y; \emptyset) = (q = q_0 \land x = 0 \land y = 0))$, 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 $\tau_A$ and define $\kappa_i(q, x, y; \emptyset) = \exists q', x', y' Reg_\delta(q', x', y') \land \varphi_\delta(q', x', y', q, x, y)$. That is, $\kappa_i$ encodes a valid transition (given by $\delta_i$) in the TC-computation starting from the configuration stored in the current register. We simulate $\Phi$ by simultaneously executing all valid transitions. For each $i \in [1, m]$ we define the rule:

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

As mentioned above, these rules create a (non-virtual) node $s$ iff the final configuration is encountered. We remark that since $A$ is deterministic, if the final configuration is encountered (and hence $\Phi$ is satisfiable) then this only happens once. Hence, $\tau_A$ 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 $t_A$ indicates whether $\Phi$ is satisfiable.
Taken together, it is easily verified that $t_A \in \tau_A(R)$ iff $L(A) \neq \emptyset$. 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 (cf. [Abiteboul et al. 1995]).

Given a relational algebra query $Q$ over a schema $S$, we define a relational schema $R$, a transducer $\tau_Q$ over $R$ in PT(CQ, relation, normal), and a tree $t_Q$ such that $t_Q \in \tau_Q(R)$ iff $Q$ is satisfiable. We define a non-recursive $\tau_Q$: 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 proof below.

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

(a) The schema $R$ consist of all base relations of $S$ and in addition, for each sub-query $Q'$ of $Q$ as given by $\text{parse}(Q)$, 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 sub-queries that return 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 $t_0 = (0,0,\ldots,0)$, \emph{i.e.}, 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 $\text{att}(R_i)$ to denote the set of attributes in a relation schema $R_i$ 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 $\tau_Q$ and tree $t_Q$ are defined such that $\tau_Q(I) = t_Q$ iff $I$ 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 $\tau_Q$ 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 $\tau_Q$ and $t_Q$ simultaneously. For each rule in $\tau_Q$ presented below, 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 (\emph{i.e.,} a sub-query) in $\text{parse}(Q)$, denoted by $p(a)$. The rule for nodes labeled $a$ is determined by the structure of the sub-query represented by the subtree rooted at $p(a)$. 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 $\tau_Q$ are given in relational algebra, but can be easily written in CQ. We define $\tau_Q$ and $t_Q$ top-down starting from the root $r$, inductively following a top-down traversal of $\text{parse}(Q)$, as follows.

(0) The start rule of $\tau_Q$ is $(q_0, r) \rightarrow (q, a_1, \phi_0^0(\emptyset; y_A, \bar{x}) = R_Q(y_A, \bar{x})), (q, b_1, \phi_0^0 = \pi_{\text{att}(R_Q)}(R_Q(1, \bar{x})))$. On instances $I$ of $R$, this rule creates an $a_1$-node $w_1$ such that
its relation register $R_{\text{reg}_1}(w_1)$ stores $I_Q(I_0)$, i.e., 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 non-emptiness of $Q(I_S)$.

We next give the rule for $(q, a_1)$, where $a_1$ corresponds to $p(a_1)$ in parse($Q$), which indicates a sub-query $Q'$ of $Q$. From the inductive construction we have that for any $a_2$-node $v$ generated and included in $t_Q$, $R_{\text{reg}_2}(v)$ stores the relation $R_{Q'}$. We define the rule for $(q, a_1)$ 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 $Q_{\text{prod}} = \pi_{\text{att}(Q_1 \times Q_2)} \sigma_{A=A'}(R_{Q_1} \times \rho_{A=A'}(R_{Q_2}))$, where $A$ is the special attribute mentioned above. We define the rule for $(q, a_1)$ to be $(q, a_1) \rightarrow (q, a_1) \cap \{Q_1(y, x) = R_{Q_1}(y, x), Q_2(y, x) = R_{Q_2}(y, x)\}$. 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 $a_i$-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 iff $I_{Q'} \neq Q_{\text{prod}}(I_{Q_1}, I_{Q_2})$, 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.

(2, 3, 4) We omit the construction for the simple cases when $Q'$ is a selection, projection or renaming sub-query, due to the space constraint.

(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 $Q_+ = \pi_{\text{att}(R_{Q_1})} R_{Q_1+Q_2}, Q_1 = \pi_{\text{att}(R_{Q_1})} \sigma_{A=1}(R_{Q_1+Q_2})$ and $Q_2 = \pi_{\text{att}(R_{Q_1})} \sigma_{A=0}(R_{Q_1+Q_2})$, respectively, where $A$ is the first attribute in $R_{Q_1+Q_2}$ that holds tags $1$ or $\theta$. Then as in case (1), we can assure the following in the rule for $(q, a_1)$. 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 $R_{Q_1\cap Q_2}$. For an instance $I$ of $R$, $I_{Q_1\cap Q_2}$ is to store $I_{Q_1} \cap I_{Q_2}$, and $I_{Q_1\cap Q_2}$ is to store $\{0\} \times I_{Q_1\cap Q_2} \cup \{1\} \times I_{Q'}$. Intuitively, we inspect set difference by checking whether $I_{Q_1} \cap I_{Q_2} = \emptyset$ and $I_{Q'} \cup I_{Q_1\cap Q_2} = I_{Q_1}$. To do this, in terms of $R_{Q_1\cap Q_2}$, we express $R_{Q'}$ and $Q_{\text{diff}} = \pi_{\text{att}(R_{Q_1})} \sigma_{A=1}(R_{Q_1\cap Q_2})$ and $Q_1 = \pi_{\text{att}(R_{Q_2})} R_{Q_2 \cap Q_2}$, respectively. Furthermore, we express $R_{Q_1\cap Q_2}$ as both $Q_1 = R_{Q_1} \cap R_{Q_2}$ and $Q_2 = \pi_{\text{att}(R_{Q_2})} \sigma_{A=0}(R_{Q_1\cap Q_2})$. We also define $Q_0 = \sigma_{A=0}(R_{Q_1} \cap R_{Q_2})$. Then as in case (1), we can assure the following in the rule for $(q, a_1)$. First, $I_{Q'} = Q_{\text{diff}}(I_{Q_2 \cap Q_2})$ and $I_{Q_1\cap Q_2}$ 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 $I_{Q_1\cap Q_2}$. We omit the details for the 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 $(q_0, a_i) \rightarrow (q_1, a_i+1, b_i = R_S(0, 0, \ldots, 0))$, $(q_0, a_i+2, b_i(0; x) = R_S(1, x))$, $(q_1, d_i, b_i(0; A; \emptyset) = \exists_x \pi_A R_S(yA, x))$. 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 $\tau_Q(I) = t_Q$ iff $Q$ is satisfiable. First, suppose that $Q$ is satisfiable, let $I_Q$ be an instance of $S$ such that $Q(I_Q)$ is nonempty. Then we obtain an instance $I$ of $R$ by letting $R_{Q'} = \{(1) \times Q(I_Q)\} \cup \{(0, 0, \ldots, 0)\}$ for any sub-query $Q'$ of $Q$ given by $\text{parse}(Q)$. Clearly, $\tau_Q(I) = t_Q$. Conversely, by the construction above 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 $t_Q$ together with the presence of a $b_1$-child of the root of $t_Q$, $Q$ is satisfiable.

(3) The equivalence problem. It suffices to show that the equivalence problem for $\text{PT}(\text{CQ, tuple, normal})$ 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 $\text{reg}_1$, $\text{reg}_2$, and is programmed by a numbered sequence $I_0, I_1, \ldots, I_t$ 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 $\text{reg}_1$ and $\text{reg}_2$ containing $m$ and $n$, respectively.

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

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

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

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

Assume w.l.o.g. that the initial ID is $id_0 = (0, 0, 0)$ and that the final ID is $id_f = (f, 0, 0)$, i.e., 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 2RM’s to the complement of the equivalence problem. Given a 2RM $M$, we construct a relational schema $R$ and two transducers $\tau_1$ and $\tau_2$ over $R$ in $\text{PT}(\text{CQ, normal, tuple})$ such that there exists an instance $I$ of $R$ such that $\tau_1(I) \neq \tau_2(I)$ if $M$ is halting.

(a) The schema $R$ is a 6-ary relation with attributes $\text{prev}$ (for previous), $\text{next}$ (for next), $\text{cs}$ (for current state), $\text{reg}_1$ (for register 1), $\text{reg}_2$ (for register 2), and $\text{ns}$ (for next state). Intuitively, an instance $I$ of $R$ consists of tuples $t$ where $t[\text{cs}, \text{reg}_1, \text{reg}_2]$ encodes an ID of $M$, $t[\text{ns}]$ encodes the next state of $M$; and $t[\text{prev}]$ and $t[\text{next}]$ provide an ordering on the tuples in $t$. An instance $I$ of $R$ is said to be well-formed if $\text{prev}$ determines $\text{next}$ and vice versa, i.e., for any $t_1, t_2 \in I$, if $t_1[\text{prev}] = t_2[\text{prev}]$
then \( t_{1[\text{next}]} = t_{2[\text{next}]} \), and vice versa; if this holds we say that \( \text{prev} \) is a key for \( \text{next} \); similarly, \( \text{next} \) is a key for \( \text{prev} \). Any well-formed instance \( I \) of \( R \) necessarily contains a unique sequence \( \sigma_I \) 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 \( \tau_1 \) and \( \tau_2 \) over \( R \) such that, when applied to a well-formed instance \( I \) of \( R \), they behave almost the same and both verify whether \( \sigma_I \) forms a valid run of \( M \). At each step, both transducers spawn an \( \alpha \)-node if the transition between the two consecutive tuples in \( \sigma_I \) is valid. If either \( \sigma_I \) does not form a valid run or \( \sigma_I \) forms a valid run but the halting state is not reached, both \( \tau_1 \) and \( \tau_2 \) simply stop. If \( \sigma_I \) is a valid run leading to a halting state, \( \tau_1 \) and \( \tau_2 \) exhibit a different behavior. More specifically, while \( \tau_1 \) creates an extra \( \alpha \)-node, \( \tau_2 \) 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 \( \alpha \)-node more than \( \tau_2(I) \).

To accommodate instances of \( R \) that are not well-formed, we modify \( \tau_1 \) and \( \tau_2 \). When a halting state is reached by \( \tau_1 \) and \( \tau_2 \), \( \tau_1 \) generates an extra \( \alpha \)-node (apart from the one it already created) iff neither \( \text{prev} \) is a key for \( \text{next} \) nor is \( \text{next} \) a key for \( \text{prev} \). In contrast, \( \tau_2 \) will generate an \( \alpha \)-node if \( \text{prev} \) is not a key for \( \text{next} \), and another \( \alpha \)-node if \( \text{next} \) is not a key for \( \text{prev} \). This suffices. Indeed, consider the following three scenarios when a halting state is encountered: (i) \( \text{prev} \) is a key for \( \text{next} \) and vice versa; in this case \( \tau_1 \) generates a single \( \alpha \)-node (because the halting state is encountered), while \( \tau_2 \) does not generate anything (since both are keys); (ii) \( \text{prev} \) is a key for \( \text{next} \) but not conversely (the symmetric case is analogous); then \( \tau_1 \) will generate a single \( \alpha \)-node and so does \( \tau_2 \); and finally (iii) \( \text{prev} \) is not a key for \( \text{next} \) and \( \text{next} \) is not a key for \( \text{prev} \); in this case, both \( \tau_1 \) and \( \tau_2 \) generate two \( \alpha \)-nodes. These are expressible as rules (with \( \neq \) in particular) in PT(CQ, tuple, normal).

Hence, \( \tau_1 \) and \( \tau_2 \) 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 \( \tau_1 \) and \( \tau_2 \). The transducer \( \tau_1 \) consists of the following rules:

\[
(q,r) \rightarrow (q_1,a,\phi_0(a_1,a_2,i,m,n,j;\emptyset) = R(a_1,a_2,i,m,n,j) \land a_1 = 0 \land i = 0 \land m = 0 \land n = 0 \land \exists z_1 z_2 z_3 R(0,0,j,z_1,z_2,z_3))
\]

\[
(q_1,a) \rightarrow \text{add to } \text{register}_1, \text{add to } \text{register}_2, \text{subtract from } \text{register}_1, \text{subtract from } \text{register}_2,
\]

\[
(q_3,a,\phi_{\text{halt}} = \exists a_1,a_2,i,m,n,j \text{ Reg}(a_1,a_2,i,m,n,j) \land i = f \land m = 0 \land n = 0 \land j = f), (q_4,a,\phi_{\text{halt+nokeys}} = \phi_{\text{halt}} \land \phi_{\text{nokey}} \land \phi_{\text{nokey}}),
\]

where \( \phi_{\text{nokey}} = \exists a_1,a_2,b_1,b_2,\bar{x},\bar{a} R(a_1,a_2,\bar{x}) \land R(b_1,b_2,\bar{x}) \land a_1 = b_1 \land a_2 \neq b_2 \), and similarly, \( \phi_{\text{nokey}} = \exists a_1,a_2,b_1,b_2,\bar{x},\bar{a} R(a_1,a_2,\bar{x}) \land R(b_1,b_2,\bar{x}) \land a_2 = b_2 \land a_1 \neq b_1 \).

The transducer \( \tau_2 \) consists of the same set of rules, except that \( (q_4,a,\phi_{\text{halt+nokeys}}) \) is replaced by \( (q_4,a,\phi_{\text{halt+nokeys}}) \), \( (q_4,a,\phi_{\text{nokey}}) \).

We now explain how to simulate the addition and subtraction. The register contents of \( M \) are stored in the \text{reg1} and \text{reg2} attributes of \( R \). The order induced by the key constraints on the \text{prev} and \text{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 \text{register1} 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 \text{rg1} is defined such that there exists a tuple in \( I \) of the form \((m,m',\bar{x})\). Similarly, when \text{register1} needs to be decremented, \( m' \) is defined such
that there is a tuple in I of the form (m’, m, x) (provided that m ≠ 0).

Let A₁ (resp. D₁) be the set of states that correspond to additions (resp. subtractions) of register. We explain “add to register” in the rule for (q₁, a) in τ₁ and τ₂. For each i ∈ A₁, the instruction Iᵢ is fixed (s₁, rg₁, s₂); we add (q₁, a, a₁) where α₁(a₁, a₂, s₁, m, n, s₂; 0) = ∃b₁, b₂, s’₁, m’, s’₂ Reg(b₁, b₂, s’₁, m’, s’₂) ∧ s’₁ = i
∧ R(a₁, a₂, s₁, m, n, s₂) ∧ a₁ = b₂ ∧ s₁ = s’₂ ∧ n = m’ ∧
(∃c₁, c₂, s”₁, m”, n”, s”₂ R(c₁, c₂, s”₁, m”, n”, s”₂) ∧ m’ = c₁ ∧ m = c₂).

Intuitively, for an instance I of R, it verifies whether for all states in A₁ (resp. A₂) 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 are encoded similarly.

The “subtract from register”-part in the rule for (q₁, a) in τ₁ and τ₂ is encoded in a similar way, and is omitted due to the lack of space. The difference here is that for each instruction Iᵢ = (s₁, rgᵢ, j, s₂) ∈ D₁, we need to add rules for (q₁, a, σᵢ=0) and (q₁, a, σᵢ=0) to separate the case when rgᵢ ≠ 0 from that when rgᵢ = 0.

One can verify that τ₁ ⊢ τ₂ if M does not halt. Thus the equivalence problem for PT(CQ, S, O) is undecidable.

\[ \square \]

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, XMG, XMLGEN) and PT(FI, relation, virtual) (ATG). Table I shows that, however, most of these languages are non-recursive: PTᵣ(IFP, tuple, normal) (SQL, XML), PTᵣ(IFP, tuple, normal) (FOR-XML), PTᵣ(CQ, tuple, normal) (annotated XSD, RDB, TreeQL). Each of these non-recursive classes is treated below.

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

\[ \text{Proposition 3. For publishing transducers } \tau \text{ in PTᵣ(IFP, tuple, O), the worst-case data complexity for } \tau \text{-transformations is in PTIME (for O normal or virtual).} \]

\[ \text{Proof. For any transducer } \tau \text{ in PTᵣ(IFP, tuple, O) over a relational schema R and for any instance I of R, the depth of the } \Sigma \text{-tree } \tau(I) \text{ induced by } \tau \text{-transformation on I is bounded by a fixed } k, \text{ which is determined by } |\tau|. \text{ From the proof of Proposition 1 it follows that the size of } \tau(I) \text{ is bounded by } O(p(|I|)^k), \text{ where } p \text{ is a polynomial. Since each query evaluated during the transformation takes at most PTIME in } |I|, \text{ it takes at most PTIME in } |I| \text{ in total to generate } \tau(I). \] 

\[ \square \]

The decision problems also become simpler, to an extent.

\[ \text{Theorem 2. (1) The emptiness, membership and equivalence problems are undecidable for PTᵣ(L, tuple, normal) for } L = \{ \text{FO, IFP} \}. \text{ (2) The emptiness problem is in PTIME for PTᵣ(CQ, tuple, normal), and is NP-complete for PTᵣ(CQ, tuple, virtual). (3) The membership problem for PTᵣ(CQ, tuple, O) is } \Sigma_2^p \text{-complete. (4) The equivalence problem for PTᵣ(CQ, tuple, O) is } \Pi_2^p \text{-complete.} \]

\[ \text{Proof. PTᵣ(IFP, tuple, normal) and PTᵣ(IFP, tuple, normal). It suffices to prove the undecidability for PTᵣ(IFP, tuple, normal). The proof of Propo-} \]

\[ \]
position 2 remains intact for PT_{nr}(FO, tuple, normal). Indeed, all transducers constructed in that proof are non-recursive (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 1(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 1(1)) uses a non-recursive 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 \( \Sigma^p_2 \)-lower bound follows from the proof for PT(CQ, tuple, normal), which uses a non-recursive transducer (Theorem 1(2)).

We now extend the \( \Sigma^p_2 \)-algorithm for PT(CQ, tuple, normal) to accommodate virtual nodes. We first establish a small model property. Given any \( \tau \) 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 \( \tau \) (see Claim 2), and \( D \) is the depth of \( \tau \), i.e., the length of the longest path in its dependency graph \( G_{\tau} \), which is a DAG since \( \tau \) 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 \( \Sigma^p_2 \)-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 \( (D \times |t|)^3 \) many tuples, one for the register of each node in \( t' \), for all nodes in \( t' \). (4) Extend step 3 of the \( \Sigma^p_2 \) algorithm given for Theorem 1(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 \( \Sigma^p_2 \)-upper bound follows.

The equivalence problem. It suffices to show that the problem is \( \Pi^p_3 \)-hard for PT_{nr}(CQ, tuple, normal), and then give \( \Pi^p_3 \) 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 1(3)).

Lower bound: The proof is by reduction from the \( \forall^* \exists^* \forall^* \cdot \text{3sat} \)-problem, which is \( \Pi^p_3 \)-complete (cf. Papadimitriou 1994). The latter problem is determined, given \( \varphi = \forall X \exists Y \forall Z \ C_1 \land \cdots \land C_r \), whether or not \( \varphi \) evaluates to true. Here \( \exists Y \forall Z \ C_1 \land \cdots \land C_r \) is an instance of \( \exists^* \forall^* \cdot \text{3sat} \) problem described in the proof of Theorem 1(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 \( \varphi \), we define a schema \( R \) and transducers \( \tau_1 \) and \( \tau_2 \) over \( R \) in PT_{nr}(CQ, tuple, normal) such that for all instance \( I \) of \( R \), \( \tau_1(I) = \tau_2(I) \) iff \( \varphi \) 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 1(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 $\tau_1$ and $\tau_2$ 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 $R$, i.e., (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 $\tau_1$ in the following steps ($\tau_2$ is defined using $\tau_1$ later on).

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

\[
(q_0, r) \rightarrow (q_1, a, \phi_0(\bar{x}; \emptyset) \equiv R_X(\bar{x})); \quad (q_1, a) \rightarrow (q_{k+1}, a, \phi_0(\bar{x}; \emptyset)), \quad (q_{k+1}, a, \phi_1(\bar{x}; \emptyset)),
\]

where for $i \in [1, m - 1]$ and $j \in \{0, 1\}$, $\phi_i(\bar{x}; \emptyset) = \text{Reg}(X) \land (x_i = j)$, and $\text{Reg}$ denotes the register. These enforcing 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 $\tau_1$, where $\phi_{m+1}(\bar{x}; \emptyset) \equiv \text{Reg}(X) \land \phi(\bar{x}; \emptyset) \land \phi_1(\bar{x}; \emptyset)$. Here $\phi(X; \emptyset) = \exists Y \forall Z C_1 \land \cdots \land C_r$; and $\phi_1$ is also given in that proof, assuring that instances of $R_X$ and $R_C$ are well-formed.

(c) We define $\tau_2$ to be the same as $\tau_1$ except that $(q_{m+1}, b) \rightarrow (q_{m+2}, c, \phi'_{m+1}(\bar{x}; \emptyset))$ where $\phi'_{m+1}(\bar{x}; \emptyset) = \text{Reg}(X)$. 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 $R_{OR}$, $\phi$ evaluates to true iff $\tau_1(I_X, I_C, I_{OR}) = \tau_2(I_X, I_C, I_{OR})$ 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 $I_{OR} \subseteq I'_{OR}$.

Then by the definition of $\tau_1$ and $\tau_2$, if $\tau_1(I_X, I_C, I_{OR}) = \tau_2(I_X, I_C, I_{OR})$, then $\tau_1(I_X, I'_C, I'_{OR}) = \tau_2(I_X, I'_C, I'_{OR})$ for any instance $I_X$ of $R_X$, since CQ queries are monotonic. Thus if $\tau_1(I_X, I_C, I_{OR}) = \tau_2(I_X, I_C, I_{OR})$ when $I_X$ ranges over all instances of $R_X$, then $\tau_1 \equiv \tau_2$. Conversely, if $\tau_1 \equiv \tau_2$, then $\tau_1(I_X, I_C, I_{OR}) = \tau_2(I_X, I_C, I_{OR})$ when $I_X$ ranges over all truth assignments of $X$. Putting (i) and (ii) together, we have that $\phi$ evaluates to true iff $\tau_1 \equiv \tau_2$. This completes the proof of the $\Pi^p_2$-lower bound.

**Upper bound:** We first provide a $\Pi^p_2$-algorithm for checking the equivalence of transducers in $\text{PT}_{\text{nr}}(\text{CQ}, \text{tuple}, \text{normal})$, and then extend it to $\text{PT}_{\text{nr}}(\text{CQ}, \text{tuple}, \text{virtual})$.

**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 $\Pi^p_2$-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 w.l.o.g. that $R$ consists of a single $k$-ary relation and that $R$ consists of 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 non-distinguished 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 $E_Q$ 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 $E_Q$. 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 $E_Q$ as follows: $([z_1], [z_2]) \in F_Q$ 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 non-value 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 $X_Q^{\text{nc}}$ the set of variables $x$ in $X_Q$ for which $[x]$ is constant; $X_Q^{\text{nc}}$ denotes $X_Q \setminus X_Q^{\text{nc}}$. 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 $X_Q^{\prime}$ of $X_Q$ in $\text{ptime}$. We define the reduced version of $Q$, denoted by $Q^\prime$, as the query that consists of $X_Q^{\prime}, Y_Q^{\prime} = Y_Q, K_Q$, i.e., the subset of $K_Q$ that consists of (non-)values of $[y]$ with $y \in X_Q^{\prime} \cup Y_Q^{\prime}$. $R_Q$ in which all variables in $x \in X_Q \setminus X_Q^{\prime}$ are replaced by the variable $x' \in X_Q^{\prime}$ such that $x \in [x']$ or is the value of $[x]$, and $L_Q$ is modified similarly. We say that $Q_1$ and $Q_2$ are $\equiv$-equivalent, denoted by $Q_1 \equiv \equiv Q_2$, if $Q_1^{\prime} \equiv Q_2^{\prime}$. The $\equiv$ relation is needed for characterizing transducer equivalence as indicated below, and can be tested in terms of $\equiv$.

**Claim 3.** $Q_1 \equiv \equiv Q_2$ iff for all instances $I$, $|Q_1(I)| = |Q_2(I)|$.

**Proof.** Assume that $Q_1^{\prime} \equiv Q_2^{\prime}$ and let $I$ be an instance of $R$. Then, for any $t \in Q_1(I)$ there exists a unique completion $t'$ such that $t \in Q_1(I)$. Indeed, the remaining attributes are either a constant or are equal to attributes already appearing in $t$. Hence, $|Q_1(I)| = |Q_1(I)| = |Q_2(I)| = |Q_2(I)|$, from which the result follows.

Conversely, assume that for all $I$, $|Q_1(I)| = |Q_2(I)|$ (or equivalently, $|Q_1(I)| = |Q_2(I)|$). We then show that $Q_1^{\prime} \equiv Q_2^{\prime}$. We first show that $|X_Q^{\prime}| = |X_Q^{\prime}|$. Assume that $|X_Q^{\prime}| < |X_Q^{\prime}|$ (similarly for $>$). Let $p_1$ be a valuation of $X_Q^{\prime} \cup Y_Q^{\prime}$ that is order-preserving w.r.t. $Q_1^{\prime}$, i.e., for all $z_1 \theta z_2 \in L_Q^{\prime}, \rho(z_1) \theta \rho(z_2)$ holds. Let $p_2$ be any order-preserving valuation that agrees with $p_1$ on $X_Q^{\prime}$. Denote by $I_{p_1}$ be the instance $\{(\rho(z_1), \ldots, \rho(z_k)) \mid R(z_1, \ldots, z_k) \in R_Q^{\prime}\}$; similarly for $I_{p_2}$. Since $X_Q$ is reduced, $Q_1^{\prime}(I_{p_1}) = Q_2^{\prime}(I_{p_2})$. However, since $X_Q^{\prime}$ is reduced too, there must exist a variable in $X_Q^{\prime}$ that is not constant and not equal to any other variable in $X_Q^{\prime}$ and that correspond to a variable in $Y_Q^{\prime}$. Hence $Q_1^{\prime}(I_{p_1}) \neq Q_2^{\prime}(I_{p_2})$. Let $I = I_{p_1} \cup I_{p_2}$ we then get (by monotonicity) that $Q_1^{\prime}(I) = Q_1^{\prime}(I_{p_1})$ while $Q_1^{\prime}(I)$ strictly contains $Q_2^{\prime}(I_{p_2})$. Thus $|Q_1^{\prime}(I)| < |Q_2^{\prime}(I)|$, contradiction. Hence, $|X_Q^{\prime}| = |X_Q^{\prime}|$.

Next, we use the characterization of the equivalence of CQ-queries (without $\neq$) given in [Klug 1988]. That is, $Q_1^{\prime} \subseteq Q_2^{\prime}$ if for each valuation $\rho$ of $X_Q^{\prime} \cup Y_Q^{\prime}$ that is order-preserving w.r.t. $Q_1^{\prime}$ it is the case that $\rho(s_Q^{\prime}) \in Q_2^{\prime}(I_{\rho})$. We show that $Q_1^{\prime} \subseteq Q_2^{\prime}$ (similarly for $Q_2^{\prime} \subseteq Q_1^{\prime}$) for $i \in [1, n]$. Here, the $z_i^{\prime}$’s denote the variables in the $i$th occurrence of $R$ in $Q_1^{\prime}$. Now, for a given $\rho$, we have that $\rho(s_Q^{\prime}) \in Q_1^{\prime}(I_{\rho})$ and therefore $Q_2^{\prime}(I_{\rho})$ is nonempty (because $|Q_1^{\prime}(I_{\rho})| = |Q_2^{\prime}(I_{\rho})|$). Let $s \in Q_2^{\prime}(I_{\rho})$ and assume that $s$ is obtained by the combination of $n$ tuples $s_1, \ldots, s_n$ such that $s_i = t_{\pi(i)}$ for some mapping $\pi : [1, n] \rightarrow [1, n]$. Let $h$ be the mapping from $X_Q^{\prime} \cup Y_Q^{\prime}$ to $X_Q^{\prime} \cup Y_Q^{\prime}$ defined by $h((z_j^\prime)^i) = z_j^{\pi(i)}$. Here, the primed variables correspond to variables in $Q_2^{\prime}$. Clearly, $h$ maps an occurrence of $R$ in $Q_2^{\prime}$ to an occurrence of $R$ in $Q_1^{\prime}$. We now argue that
\( h \) must also map the summary of \( Q'_2 \) to that of \( Q'_1 \). From this, it follows that the valuation \( \rho \circ h \) of \( XQ'_2 \cup YQ'_2 \) gets \( \rho(sQ'_1) \) in \( Q_2(I_\rho) \). Since the above argument holds for any \( \rho \), we then may conclude that \( Q'_1 \subseteq Q'_2 \), as desired.

It remains to show that \( h(sQ'_2) = sQ'_1 \). Suppose otherwise, then either there exists a variable \( x' \in sQ'_2 \) such that \( [h(x')] \) does not contain any variable from \( sQ'_1 \); or \( [h(x')] \) has more than one variable in common with \( sQ'_1 \). Observe that since \( Q'_2 \) is reduced, \( [h(x')] \) cannot have a value. Therefore, valuations can assign an arbitrary value to the variables in \( [h(x')] \). Hence, another argument as above shows the existence of an instance \( I \) such that \( |Q'_1(I)| < |Q'_2(I)| \). This contradicts our assumption and therefore \( h(sQ'_2) = sQ'_1 \). □

We now continue towards the characterization of equivalence. We introduce some notations first. Let \((q, a)\) be a state-label pair in \( \tau \) and let \((q, a) \rightarrow (q_1, a_1, \psi_1), \ldots, (q_k, a_k, \psi_k)\) be the corresponding rule in \( \tau \). We partition indices \([1, k]\) and associate \( S_r(q, a) = \{S_1, S_2, \ldots, S_t\} \) 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 \( G_\tau \) be the dependency graph of \( \tau \). We define the type of a node \((v, q, a)\) in \( G_\tau \) as the list \([lab(S_1), \ldots, lab(S_t)]\). Let \( \rho \) be a path in \( G_\tau \) starting from the root node of \( G_\tau \). Denote by \( Q_\rho \) the composition of all queries along \( \rho \); assume that \( \rho \) ends in \( v(q, a) \). We denote by \( Q'_\rho \) the CQ query obtained by composing \( \psi_i \), i.e., the \( i \)-th query in the RHS of the rule for \((q, a)\) in \( \tau \), with \( Q_\rho \). We assume that \( G_\tau \) does not contain nodes that are not reachable from the root \( \tau \), and moreover, that for each path \( \rho \) in \( G_\tau \), \( Q_\rho \) is satisfiable (note that the latter can be tested in a similar way as in the proof of Theorem 1 \((1)\)). Two graphs \( G_{\tau_1} \) and \( G_{\tau_2} \) 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 \( \cong \) and Claim 3 to union of CQ queries. We then have the following characterization of transducer equivalence:

**Claim 4.** For any two publishing transducers \( \tau_1 \) and \( \tau_2 \) in \( PTrans(CQ, \text{tuple, normal}) \) we have that \( \tau_1 \cong \tau_2 \) iff \((i)\) \( G_{\tau_1} \cong G_{\tau_2} \) and \((ii)\) for all paths \( \rho \) in \( G_{\tau_1} \), each \( S_i \in S_r(q, a) \), \( I_\rho \subseteq \bigcup_{j \in S_i} Q^0_{\rho(j)} \rightarrow \bigcup_{j \in I_\rho} Q^0_{h(\rho)^{-1}(j)} \) in case \( \rho \) ends in \( v(q, a) \) with a not equal to \( \text{text} \); and \( \bigcup_{j \in S_i} Q^0_{\rho(j)} \subseteq \bigcup_{j \in I_\rho} Q^0_{h(\rho)^{-1}(j)} \) otherwise. (Here, \( h \) is a homomorphism between \( G_{\tau_1} \) and \( G_{\tau_2} \).)

**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 \( \text{children}(v) \) is different from \( \text{children}(w) \), where \( w \) is the node in \( \tau_2(I) \) corresponding to \( v \). Assume that \( \tau_1 \) (resp. \( \tau_2 \)) is in state \((q, a)\) (resp. \((q', a)\)) when reaching \( v \) (resp. \( w \)). Such an instance \( I \) exists if either \( (q, a) \) and \( (q', a) \) have different types; or they have the same type but some labels appear with a different multiplicity in \( \text{children}(v) \) and \( \text{children}(w) \). 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 \( Q_\rho \) along paths \( \rho \) in \( G_{\tau_1} \) and \( G_{\tau_2} \) 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 \( \tau_1 \) and \( \tau_2 \) disagree, again by Claim 3. □
As will be shown in Theorem 3(1), composed CQ queries can be rewritten as a
program in nonrecursive \textsf{LinDatalog}, in \textsf{PTime}. A \textsf{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 \textsf{IDB}
predicate (i.e., relation name), and we allow some \( p_i \) to be \( \neq \) (see e.g., [Abiteboul et al. 1995]). Let \( Q_1^\rho \cup \cdots \cup Q_n^\rho \) be a union of conjunctive queries as in Claim 4.
It is possible to encode each composed query \( Q_\rho^i \equiv Q_n^\rho \circ \cdots \circ Q_1^\rho \) in \textsf{LinDatalog}
because in each \( Q_1^\rho \), although there may be multiple occurrences of \textsf{Reg}, all of these
indicate the same single tuple since the transducer is in \textsf{PT}_{\text{nr}}(\textsf{CQ}, \textsf{tuple}, O); as a
result, for each \( Q_1^\rho \) we can define an \textsf{IDB} predicate \( p_j^\rho \), such that \( p_j^\rho \) has a unique
rule and in the \textsf{rhs} of the rule there is a unique \textsf{IDB} predicate \( p_j^\rho \), which encodes
\( Q_{1-1}^\rho \). As a consequence, we obtain a \textsf{LinDatalog} program \( \Pi^\rho \) for each \( Q_\rho^i \), the
combination of these in a single non-recursive program \( \Pi^\rho \) encodes the union of the
\( Q_\rho^i \)'s.

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

Furthermore, a straightforward induction on \( n \) suffices to show the following
claim. We remark that if \( \Pi \) is not linear or not deterministic, the claim does not
hold since some \( Q_1 \) may appear in \( Q_n \) exponentially many times.

\textbf{Claim 5.} For each nonrecursive deterministic \textsf{LinDatalog} program \( \Pi^\rho \), a CQ
query \( Q \) can be computed in \( O(|\Pi|) \) time such that \( Q \) and \( \Pi^\rho \) are equivalent.

It remains to show that for given transducers \( \tau_1 \) and \( \tau_2 \) we can check the conditions
in Claim 4 in \( \Pi_\rho^i \). We give an algorithm for testing that \( \tau_1 \) and \( \tau_2 \) are \textit{not}
equivalent, as follows. (a) Guess a mapping \( h \) from \( \Gamma_1 \) to \( \Gamma_2 \). (b) Check whether \( h \) and \( h^{-1} \) make \( \Gamma_1 \cong \Gamma_2 \). If not, reject. (c) If \( \Gamma_1 \cong \Gamma_2 \), guess a path \( \rho \) from
the root of \( \Gamma_1 \), and compute nonrecursive \textsf{LinDatalog} programs \( \Pi_1^\rho \) encoding the
union of \( Q_\rho^i \)'s, and \( \Pi_2^\rho \) for \( h(\rho) \) in \( \Gamma_2 \), as described in Claim 4. Note that \( \Pi_1^\rho \)'s and
\( \Pi_2^\rho \)'s can be computed in \textsf{PTime}. (d) Check whether all \( \Pi_1^\rho \) and \( \Pi_2^\rho \) are (c-) equivalent as described in Claim 4. One can easily verify that \( \Pi_1^\rho \) and Claim 3 can be
extended to nonrecursive \textsf{LinDatalog}. By the definition of \( \equiv_c \), (c-)equivalence can be checked in terms of equivalence \( \equiv \), and thus below we shall focus on \( \equiv \) only.
Provided that \( \Pi_1^\rho \) and \( \Pi_2^\rho \) are not equivalent for some \( i \), then we can conclude that
\( \tau_1 \) and \( \tau_2 \) are not equivalent. If step (d) is in \( \Sigma_2^P \), we can decide whether \( \tau_1 \) and \( \tau_2 \) are not equivalent in \( \Sigma_2^P = NP \Sigma_2^P \), and thus its complement, testing the equivalence of
two transducers in \textsf{PT}_{\text{nr}}(\textsf{CQ}, \textsf{tuple}, normal), is in \( \Pi_\rho^i \).

For step (d), it is easy to verify that \( \Pi_1^\rho \) and \( \Pi_2^\rho \) are not equivalent iff either there
exists a deterministic sub-query \( (\Pi_1^\rho)' \) of \( \Pi_1^\rho \) that is not contained in \( \Pi_2^\rho \), or vice versa.
Leveraging this, in step (d) we guess a deterministic sub-query \( (\Pi_2^\rho)' \) of \( \Pi_2^\rho \)
and check whether \( (\Pi_2^\rho)' \) is not contained in \( \Pi_2^\rho \), and similarly with the roles of \( \Pi_1^\rho \)
and \( \Pi_2^\rho \) reversed. By Claim 5, \( (\Pi_1^\rho)' \) is \textsf{PTime} definable as a CQ query \( Q \). The next
claim shows that deciding whether \( Q \) is not contained in \( \Pi_2^\rho \) is in \( \Sigma_2^P \), as desired.

\textbf{Claim 6.} It is in \( \Pi_\rho^i \) time to determine, given a CQ query \( Q \) and a nonrecursive
\textsf{LinDatalog} program \( \Pi \), whether or not \( Q \) is contained in \( \Pi \).
Proof. Proposition 2.10 of [van der Meyden 1997] shows that the combined complexity of model checking for indefinite order databases and CQ queries, i.e., checking whether an indefinite order database is a model of a CQ query, is \textit{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 \textsc{LinDatalog} programs is \textit{PTIME} reducible to model checking for indefinite order databases and nonrecursive \textsc{LinDatalog}. We then show that the combined complexity of the latter is in \( \Pi^p_2 \).

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

We next give a \( \Pi^p_2 \)-time model-checking algorithm. Given any indefinite order database \( D \) and nonrecursive \textsc{LinDatalog} program \( \Pi \), the algorithm guesses a minimal model \( M \) of \( D \), and checks whether \( M \) does not satisfy \( \Pi \) (see [van der Meyden 1997] for discussions of minimal models). One can verify that checking whether or not \( M \) satisfies \( \Pi \) can be done in \textit{NP}. Thus the complement of the model checking problem is in \( \Sigma^p_2 \). Hence the combined complexity for model checking of indefinite order databases and nonrecursive \textsc{LinDatalog} programs is in \( \Pi^p_2 \). \( \square \)

\textbf{PT}_{mr}(CQ, tuple, virtual). Given two transducers \( \tau_1, \tau_2 \) in \( \text{PT}_{mr}(CQ, \text{tuple, virtual}) \), we construct in \textit{PTIME} equivalent \( \tau_1', \tau_2' \) without virtual nodes such that the \( \Pi^p_2 \)-algorithm for the equivalence problem of \( \text{PT}_{mr}(CQ, \text{tuple, normal}) \) can be used.

Given a transducer \( \tau \) in \( \text{PT}_{mr}(CQ, \text{tuple, virtual}) \), we define an equivalent \( \tau' \) such that it contains normal output nodes only but some of its queries may be in nonrecursive \textsc{LinDatalog} rather than CQ. As above, let \( G_\tau \) be the dependency graph of \( \tau \) 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 \( G'_\tau \) by removing virtual nodes from \( G_\tau \) as follows. For any two normal nodes \( n_1 = v(q, a), n_2 = v(q', a') \) in \( G_\tau \), let \( G(n_1, n_2) \) be the largest connected sub-graph of \( G_\tau \) 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 \( \tau \) is nonrecursive, \( G(n_1, n_2) \) is a rooted \( \text{DAG} \) 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 \( G'_\tau \) is obtained from \( G_\tau \) by substituting a new edge \( (n_1, n_2) \) for \( G(n_1, n_2) \) in \( G_\tau \) as long as \( G(n_1, n_2) \) is nonempty, when \( (n_1, n_2) \) range over all pairs of normal nodes in \( G_\tau \). 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 3(1) gives a method to define the query in nonrecursive \textsc{LinDatalog}. Then from \( G'_\tau \) we can derive transducer \( \tau' \) as follows: for each \( v(q, a) \), define the rule \( (q, a) \rightarrow (q_1, a_1, \psi_1) \ldots (q_k, a_k, \psi_k) \), where \((v(q_1, a_1), \ldots, v(q_k, a_k))\) is the list of \textit{children}(\(v(q, a)\)), as ordered in \( G_\tau \), and \( \psi \) is
the query associated with the edge from \(v(q,a)\) to \(v(q_i,a_i)\). It is straightforward to show that \(\tau\) and \(\tau'\) are equivalent, and that \(\tau'\) can be constructed in \(\text{PTIME}\).

We now obtain an algorithm for testing the equivalence of \(\tau_1\) and \(\tau_2\) as follows. First, construct equivalence \(\tau'_1\) and \(\tau'_2\) without virtual nodes, as described above, in \(\text{PTIME}\). Then check the equivalence of \(\tau'_1\) and \(\tau'_2\) by using the \(\Pi^p_1\) algorithm for \(\text{PT}_{nr}(\text{CQ}, \text{tuple}, \text{normal})\) given above. It suffices to observe that the algorithm for \(\text{PT}_{nr}(\text{CQ}, \text{tuple}, \text{normal})\) trivially extends to transducers in which some queries are in non-recursive \(\text{LinDatalog}\) rather than \(\text{CQ}\).

6. EXPRESSIVENESS OF PUBLISHING TRANSDUCERS

In this section, we characterize the expressive power of publishing transducers in terms of relations-to-relation mappings (\textit{i.e.}, relational query languages) and relations-to-tree mappings (\textit{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 \(\tau\). 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 \(\tau\)-transformation on \(I\) yields a final tree \(\xi\) with local storage in \(\text{Tree}_{Q \times \Sigma}\), from which the output \(\Sigma\)-tree \(\tau(I)\) is obtained by removing local stores and states (recall from Section 3). The output relation induced by \(\tau\) on \(I\), denoted by \(R_{\tau}(I)\), is then defined to be the union of the registers \(\text{Reg}_{a_i}(v)\) for all nodes \(v\) labeled \(a_o\) in \(\xi\). Therefore, we refer to \(\tau\) as a \textit{relational query} when \(\tau\) is viewed as a mapping from instances \(I\) to \(R_{\tau}(I)\). When \(\tau\) is viewed as a relation-to-tree mapping, we refer to \(\tau\) as a \textit{tree generating mapping}.

We want to compare the expressiveness of one class \(A = \text{PT}(L_1,S_1,O_1)\) with that of another class \(B = \text{PT}(L_2,S_2,O_2)\) both as a tree generation and a relational query language. We say that \(A\) is \textit{contained in} \(B\) as a tree/relational query language, denoted by \(A \subseteq B\), if for any \(\tau_1\) in \(A\) defined for a relational schema \(R\), there exists \(\tau_2\) in \(B\) for the same \(R\) such that they define the same tree/relational query. The two classes are said to be \textit{equivalent} in expressive power, denoted by \(A = B\), if \(A \subseteq B\) and \(B \subseteq A\). We say that \(A\) is \textit{properly contained in} \(B\), denoted by \(A \subset B\), if \(A \subseteq B\) but not \(A = B\). These notions extend to comparing \(\text{PT}(L,S,O)\) versus other tree generating formalisms, and versus relational query languages.

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

\textbf{Outline}. We study the expressive power of all the classes \(\text{PT}(L,S,O)\) defined in Section 3 with respect to relational query and tree generation languages, in Sections 6.2 and 6.3, respectively. We then investigate the expressive power of
existing XML publishing languages in Section 6.4. The results in this section hold irrespective of whether the queries in $\mathcal{L}$ have explicit access to the order $\leq$ on the domain $D$, unless explicitly stated otherwise.

6.2 Expressiveness in Terms of Relational Queries

We start by treating 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 $\text{LinDatalog}$, which we have seen in the proof of Theorem 2. 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 $\text{ans}$ containing the result of the query expressed by the program. Another fragment is $\text{LinDatalog(FO)}$ (see e.g., [Grädel 1992]) which extends $\text{LinDatalog}$ by allowing $p_i$ to be an arbitrary FO-formula over the EDB predicates. In [Grädel 1992] it is shown that $\text{LinDatalog(FO)}$ captures $\text{nlogspace}$ over ordered databases.

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

**Theorem 3.** When treated as relational query languages,

1. $\text{PT}(\mathcal{L}, S, \text{virtual}) = \text{PT}(\mathcal{L}, S, \text{normal}),$
2. $\text{PT}(\text{CQ}, \text{tuple}, O) = \text{LinDatalog},$
3. $\text{PT}(\text{FO}, \text{tuple}, O) = \text{LinDatalog(FO)} \subset \text{nlogspace}.$
4. $\text{PT}(\text{FO}, \text{relation}, O)$ and $\text{PT}(\text{IFP}, \text{relation}, O)$ capture $\text{pspace}.$
5. $\text{PT}(\text{IFP}, \text{tuple}, O) = \text{IFP}.$

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

**Proof.** The proof is referred to the Appendix. \(\square\)

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

**Proposition 4.** When treated as relational query languages,

1. $\text{PT}(\text{CQ}, \text{tuple}, O) \subset \text{PT}(\text{FO}, \text{tuple}, O)$
2. $\subset \text{PT}(\text{IFP}, \text{tuple}, O)$
3. $\subset \text{PT}(\text{FO}, \text{relation}, O)$
4. $\subset \text{PT}(\text{FO}, \text{relation}, O),$
5. $\text{PT}(\text{CQ}, \text{tuple}, O) \subset \text{PT}(\text{CQ}, \text{relation}, O)$
6. $\subset \text{PT}(\text{FO}, \text{relation}, O),$
7. $\text{PT}(\text{CQ}, \text{relation}, O) \not\subset \text{PT}(\text{FO}, \text{tuple}, O),$
The containment in statement (2) is proper if \( \text{NLOGSPACE} \neq \text{PTIME} \).

**Proof.** The proof is referred to the Appendix. \( \Box \)

### 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 4, Proposition 5 below shows that when it comes to tree generation, virtual nodes do add expressive power to publishing transducers. Moreover, if \( L \subset L' \), then \( \text{PT}(L', S, \text{normal}) \) properly contains \( \text{PT}(L, S, \text{normal}) \) whereas in the relational query setting, \( \text{PT} \text{(IFP, relation, normal)} = \text{PT} \text{(FO, relation, normal)} \). The other results in Proposition 5 are comparable to their counterparts in Theorem 4. In particular, it shows that \( \text{PT} \text{(FO, relation, virtual)} = \text{PT} \text{(IFP, relation, virtual)} \).

**Proposition 5.** For tree generation,

1. \( \text{PT}(L, S, \text{normal}) \subset \text{PT}(L, S, \text{virtual}) \)
2. \( \text{PT}(L, S, \text{normal}) \subset \text{PT}(L', S, \text{normal}) \) if \( L \subset L' \)
3. \( \text{PT}(CQ, \text{tuple, virtual}) \subset \text{PT}(FO, \text{tuple, virtual}) \)
4. \( \text{PT}(CQ, \text{relation, virtual}) \subset \text{PT}(FO, \text{relation, virtual}) \)
5. \( \text{PT}(CQ, \text{tuple, virtual}) \subset \text{PT}(FO, \text{relation, virtual}) \)
6. \( \text{PT}(IFP, \text{tuple, virtual}) = \text{PT}(IFP, \text{relation, virtual}) \)
7. \( \text{PT}(L, \text{tuple, } O) \subset \text{PT}(L, \text{relation, } O) \)
8. \( \text{PT}(L, \text{relation, normal}) \not\subset \text{PT}(L', \text{tuple, virtual}) \)
9. \( \text{PT}(L, \text{tuple, virtual}) \not\subset \text{PT}(L', \text{relation, normal}) \) with \( L' \subset L \)
10. \( \text{PT}(CQ, \text{tuple, virtual}) \not\subset \text{PT}(CQ, \text{relation, normal}) \)
11. \( \text{PT}(FO, \text{tuple, virtual}) \not\subset \text{PT}(FO, \text{relation, normal}) \)

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

**Proof.** (1) The inclusion is immediate. To show proper containment it suffices to show that there is a transducer \( \tau \) in \( \text{PT}(CQ, \text{tuple, virtual}) \) not expressible in \( \text{PT}(IFP, \text{relation, normal}) \). Consider the transducer \( \tau_1 \) in the proof of Proposition 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 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 \( \text{PT}(IFP, \text{relation, normal}) \) can only output polynomially many nodes (in the size of the input graph) below the root. Therefore, it follows that in fact \( \text{PT}(CQ, \text{tuple, virtual}) \not\subset \text{PT}(IFP, \text{relation, normal}) \).

(2) Let \( q \) be a Boolean query in \( L' \setminus L \). We then simply define a transducer \( \tau_q \) which produces a tree \( r(a) \), i.e., 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, \( \tau_q \) is not expressible in \( \var{L}, S; \text{normal} \) as \( q \) is not definable in \( \var{L}' \).

(3) Let \( q \) be a Boolean query definable in \( \var{FO} \) but not in \( \var{LIN DATALOG} \). Such a query can always be found as \( \var{LIN DATALOG} \) only defines monotone queries. Define a transducer \( \tau_q \) 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 \( \tau \) in \( \var{PT}(\var{CQ}, \text{tuple, virtual}) \) equivalent to \( q \). Then modify \( \tau \) into a transducer \( \tau' \) 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 non-virtual. Then, \( \tau' \) defines \( q \) which means by Theorem 3(1) that \( q \) is definable in \( \var{LIN DATALOG} \). Contradiction.

(4) We show that \( \var{PT}(\var{FO}, \text{tuple, virtual}) \subset \var{PT}(\var{IFP}, \text{tuple, virtual}) \) if \( \var{PTIME} \neq \var{NLOGSPACE} \). Let \( q \) be a Boolean IFP query over ordered databases (a PTIME query) not in \( \var{NLOGSPACE} \). Consider again the transducer \( \tau_q \) which produces a tree \( r(a) \) if \( q \) evaluates to true on the input database, and it produces a trivial tree \( r \) otherwise. Assume there is an equivalent transducer in \( \var{PT}(\var{FO}, \text{tuple, virtual}) \), 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 3(3).

(5) Let \( \tau \) be a transducer in \( \var{PT}(\var{FO}, \text{relation, virtual}) \) defining a Boolean relational query \( q \) which is not in \( \var{PT}(\var{CQ}, \text{relation, virtual}) \). By Theorem 4(1) and Proposition 4(6) such a transducer exists. Modify \( \tau \) into \( \tau' \) as follows. Change the arity of the register of the output label \( a_o \) from nullary to unary such that \( R_{\var{CQ}}(I) \) is non-empty iff \( q(I) \) is true. Modify right-hand sides of the rules such that whenever an \( a_o \) is output with non-empty register, a \( b \)-labeled first-child is output as well. The latter is clearly definable in \( \var{FO} \). Here \( b \) is a new label. Denote by \( q' \) the relation-to-tree query expressed by \( \tau' \). We now argue that \( q' \) cannot be defined in \( \var{PT}(\var{CQ}, \text{relation, virtual}) \). Assume there is such a transducer \( \tau'' \). Then, \( \tau'' \) and \( \tau' \) define the same relation-to-tree mappings. But, as the non-emptiness of the register of an \( a_o \)-labeled nodes is encoded in the tree, both \( \tau'' \) and \( \tau' \) also define the same relational query \( q \) which leads to contradiction.

(6) To simulate a transducer in \( \var{PT}(\var{IFP}, \text{relation, virtual}) \) by one in \( \var{PT}(\var{FO}, \text{relation, virtual}) \) 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 \( \var{PT}(\var{L}, \text{relation}, O) \) are capable of generating trees of exponential depth even when \( \var{L} = \var{CQ} \) (Proposition 1(4)) whereas transducers in \( \var{PT}(\var{L}, \text{tuple}, O) \) can only induce trees of polynomial depth (Proposition 1(2)).

(8) Proof is similar to (7).

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

(10–11) Take \( R \) as a binary relation and let \( s \) and \( t \) be two constants. Consider the publishing transducer \( \tau \) in \( \var{PT}(\var{CQ}, \text{tuple, virtual}) \) defined by the rules: \( \delta(q_0, r) \rightarrow (q, v, R(s, x)) \) and \( \delta(q, v) \rightarrow (q, v, \exists y \var{Reg}_o(y) \wedge R(y, x)) \) or \( (q, a, \exists y \var{Reg}_o(y) \wedge y = t) \).
Here, $v$ is a virtual label while $a$ is not. Then, $\tau$ outputs the tree $\tau(a \cdots a)$ where $k$ is the number of simple paths from $s$ to $t$. As it cannot be checked in CQ or FO whether there is a (simple) path from $s$ to $t$, $\tau$ 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 $\phi_e, \phi_<$ and $(\phi_a)_{a \in \Sigma}$ such that on every $R$-structure $I$, $\phi_e(I)$, $\phi_<(I)$ and $\phi_a(I)$ define the edge relation, the ordering on the siblings, and the $a$-labeled nodes of the tree, respectively. To express transformations of exponential size increase (like publishing transducers can), $\phi_e(I)$ defines a DAG, and we consider its unfolding as a tree when making a comparison with publishing transducers. First-order (resp. second-order) transductions are those where nodes of the output tree are $k$-ary tuples (resp. $k$-ary relations) over the input structure, for some fixed $k$. 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 $\phi_e$, $\phi_<$ 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 below.

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 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 $\text{dom}(I)$ 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 $\{d \mid I \models \phi(d)\}$.

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

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

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 exactly one element and this element is the root of the DAG;
(c) for all $d, d', d'' \in \text{dom}(I)^k$, if $I \models \phi_<(d, d', d'')$ then $(d, d') \in \phi_e(I), (d, d'') \in \phi_e(I)$, and $\phi_<(d, \bar{y}, \bar{z})$ is a total order, i.e., $\phi_<$ defines an ordering on the children of each $\bar{x}$;
(d) $\phi_{\text{fc}}(I)$ and $\phi_{\text{ats}}(I)$ are the first-child and next-sibling relations induced by $\phi_e, \phi_<$;
(e) for every $a$ and $a'$ in $\Sigma$ 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; i.e., 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 $\phi_{\text{root}}$, $\phi_{fc}$, and $\phi_{ns}$ are definable in FO from $\phi_{e}$ and $\phi_{<}$, but not in CQ.

As described above, 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 $\ell$ such that for every input $I$, $T(I)$ is a tree of depth at most $\ell$.

**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 $X, Y, Z, \ldots$, of arity $k$. An $\mathcal{L}$-formula $\phi(X_1, \ldots, X_n)$ on an input structure $I$ then defines the set $\phi(I) = \{(A_1, \ldots, A_n) \mid I \models \phi(A_1, \ldots, A_n)\}$, where each $A_i$ 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 $\phi_{a}(I)$.

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

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

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

(c) $M_{<}(I) = \{(d, d', d'') \mid (I, d, d', d'') \text{ is accepted by } M_{<}\}$ and for all $d, d', d'' \in \text{dom}(I)^k$, when $(d, d', d'') \in M_{<}(I)$ then $(d, d') \in M_{e}(I)$ and $(d, d'') \in M_{e}(I)$, and for each $d$, $\{(d', d'') \mid (d, d', d'') \in M_{<}\}$ 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 $\mathcal{L}$-transductions are defined in a similar way.

**Theorem 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$_{\text{nr}}$($\mathcal{L}$, tuple, virtual) is definable as a fixed-depth $\mathcal{L}$-transduction.

(3) There is a recursive transducer in PT($\mathcal{L}$, 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$_{\text{nr}}$($\mathcal{L}$, tuple, $O$).

(5) Over ordered input structures, PT($\mathcal{L}$, relation, virtual) and PT($\mathcal{L}$, 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 $\tau_{T}$ in PT($\mathcal{L}$, tuple, virtual) such that $\tau_{T}(I) = T(I)$ for any instance $I$.

Let $T = (\phi_{\text{dom}}, \phi_{\text{root}}, \phi_{e}, \phi_{<}, \phi_{fc}, \phi_{ns}, (\phi_{a})_{a \in \Sigma})$ 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 $\delta$ consists of the following rules:
The above 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 $v$:

$$(q, a) \rightarrow (q, a_1, \phi_{\text{root}}(\bar{x}) \land \phi_{a_1}(\bar{x})), \ldots, (q, a_n, \phi_{\text{root}}(\bar{x}) \land \phi_{a_n}(\bar{x})).$$

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

$$(q, v) \rightarrow (q, a_1, \phi_{\text{root}}(\bar{x}) \land \phi_{a_1}(\bar{x})), \ldots, (q, a_n, \phi_{\text{root}}(\bar{x}) \land \phi_{a_n}(\bar{x})).$$

Finally, the rule

$$(q_1, v) \rightarrow (q_1, a_1, \phi_v(y) \land \phi_{a_1}(\bar{x})), (q_1, a_2, \phi_v(y) \land \phi_{a_2}(\bar{x})), \ldots, (q_1, a_n, \phi_v(y) \land \phi_{a_n}(\bar{x})).$$

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

We may assume that we have constants for every state in $Q$ and every label in $\Sigma$. Indeed, we can always simulate these by introducing registers with higher arity. Let $k'$ be the largest arity of a register in $\tau$ 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 $\Sigma$, respectively.

We now define the different formulas constituting the $\mathcal{L}$-transduction $T_\tau$. We define $\phi_{\text{dom}}$ to be simply $\text{true}$. 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

$$\phi^1_i(x_1, x_2, \bar{x}; y_1, y_2, ar{y}) \equiv \bigvee_{(q, a) \rightarrow (q_1, a_1, \bar{y}), \ldots, (q_n, a_n, \bar{y}) \in \delta} x_1 = q \land x_2 = a \land (\bigwedge_{i=1}^n y_1 = q_1 \land y_2 = a_i \land \phi^1_i(y)).$$

where each $\phi^1_i$ is obtained from $\phi_i$ (i.e., the $i$th formula in the rule in $\delta$ under consideration) by replacing each $\text{Reg}(\bar{x})$ by $\bar{x} = \bar{z}$. It is easily verified that the formula $\phi^1_i$ defines the correct DAG but with virtual nodes.

Therefore, in the second step, we define $\phi_e$ such that it skips all virtual nodes in the DAG defined by $\phi^1_e$. For this we define

$$\phi_e(x_1, x_2, \bar{x}; y_1, y_2, ar{y}) \equiv \bigwedge_{a \in \Sigma_a} \phi_a(x_1, x_2, \bar{x}) \land \bigwedge_{a \in \Sigma_a} \phi_a(y_1, y_2, ar{y}) \land \phi^1_e(x_1, x_2, \bar{x}; y_1, y_2, ar{y}) \land \exists \exists \exists \phi^1_e(x_1, x_2, \bar{x}; \bar{z}) \land \phi^1_e(\bar{z}; y_1, y_2, \bar{y}) \land \bigwedge_{a \in \Sigma_a} \phi_a(\bar{z}).$$

Here, $\phi^1_e$ denotes the transitive closure of $\phi^1_e$ which is expressible in FO because the depth of the output tree depends on $\tau$ and not on the input structure (recall that $\tau$ is non-recursive). Moreover, because $\tau$ is non-recursive, 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 $I$, $\tau(I) = T(I)$. 

\[ \text{(1)} \]
(3) We here provide an example of a recursive transducer in \( \text{PT}(\text{FO, tuple, normal}) \) that is not expressible as an FO-transduction. Let the schema consist of an edge relation \( E \) and two constants \( s \) and \( t \). Let \( \tau \) 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 \( \text{PT}(\text{FO, tuple, normal}) \)-transducer consisting of the following rules:

\[
(q_0, r) \rightarrow (q, a, x = s), \quad (q, a) \rightarrow (q, a, \exists y \text{Reg}(y) \land E(y, x)), 
\quad (q, b, \exists y \text{Reg}(y) \land y = t),
\]

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 \( \tau \).

(4) The containment of \( \text{PT}_{nr}(\text{CQ, tuple, O}) \) 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 \( \tau_T \) in \( \text{PT}_{nr}(\mathcal{L}, \text{tuple}, \text{O}) \).

Let \( T = (\phi_{\text{dom}}, \phi_{\text{root}}, \phi_c, \phi_{<}, \phi_{\text{f}, c}, \phi_{\text{n}, c}, (\phi_a)_{a \in \Sigma}) \) be a fixed-depth \( \mathcal{L} \)-transduction. Let \( \ell \) 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 \( Q = \{q_0, q_1, \ldots, q_{\ell}\} \) and \( \Theta(a) = k \) for all \( a \), and \( \delta \) consists of the following rules. The start rule generates \( a \) nodes:

\[
(q_0, r) \rightarrow (q, a_1, \phi_{\text{root}}(\bar{x}) \land \phi_{a_1}(\bar{x})), \ldots, (q, a_n, \phi_{\text{root}}(\bar{x}) \land \phi_{a_n}(\bar{x})).
\]

And for all \( i \in [1, \ell] \) and \( a \in \Sigma \), we define the rule \( (q_i, \text{a}) \rightarrow (q_{i+1}, a_1, \phi_1(\bar{x})), \ldots, (q_{i+1}, a_n, \phi_n(\bar{x})) \), where \( \phi_i(\bar{x}) = \exists y (\text{Reg}(y)) \land \phi_c(y, \bar{x}) \land \phi_{a_i}(\bar{x}) \). 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 \( T = (M_{\text{dom}}, M_{\text{root}}, M_c, M_{\text{f}, c}, M_{\text{n}, c}, M_{<}, (M_a)_{a \in \Sigma}) \) be a \( \text{PSpace} \)-transduction. Since we assume that the domain is ordered, for every \( \text{PSpace} \) Turing machine \( M \) there exists a Partial Fixed Point (pfp) sentence \( \xi_M \) over the relational schema extended with symbols \( X \) and \( Y \) such that \( (I, X, Y) \models \xi_M \) if and only if \( \xi_M \) [Flum and Ebbinghaus 1999]. Every such sentence can be written as \( \exists \ell \text{FPFP}(\chi) \bar{x} \) where \( \chi \) is first-order and total, i.e., it always reaches a fixpoint [Flum and Ebbinghaus 1999]. Although registers in transducers in \( \text{PT}(\text{FO, relation, virtual}) \) 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 \( \text{PT}(\text{FO, relation, virtual}) \) a formula \( \exists \ell \text{FPFP}_2(\chi) \bar{x} \) where \( \chi \) 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 \( \chi \) 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 non-virtual label. Relations (of arity \( k \)) can be enumerated by considering each relation as a number between \( 0 \) and \( 2^k \). When a node has been output with a non-virtual label and register content \( X \), the transducer tests for every relation \( Y \) whether \( (X, Y) \in M_{\text{f}, c} \). Similarly for all next-sibling relations.

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

Regular tree languages. Recall that a DTD $d$ over $\Sigma$ is a mapping from $\Sigma$-symbols to regular expressions over $\Sigma$. A $\Sigma$-tree $t$ conforms to $d$ 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 $\Sigma$ is a triple $(\Sigma', d, \mu)$, where $\Sigma \subseteq \Sigma'$, $\mu$ is a mapping $\Sigma' \mapsto \Sigma$, and $d$ is a DTD over $\Sigma'$. A $\Sigma$-tree $t$ conforms to $D$ if there exists a $\Sigma'$-tree $t'$ that satisfies $d$ and moreover, $t = \mu(t')$, where $\mu$ is canonically extended from labels to trees. We denote by $L(D)$ the set of all $\Sigma$-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 $\tau$ 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 5.** Every extended DTD over $\Sigma$ is definable in $PT(FO, tuple, virtual)$. There exist DTDs that are not definable in $PT(CQ, relation, virtual)$.

**Proof.** We first show that for any normalized DTD $d$ there exists a transducer $\tau$ in $PT(FO,tuple,normal)$ such that $L(d) = \tau(R)$. We then modify this construction for arbitrary DTDs by allowing for virtual nodes in the transducer. We conclude by showing that a slight modification entails the result for extended DTDs as well. As a result $PT(FO,tuple,virtual)$ contains the class of regular tree languages.

A normalized DTD $d$ is a DTD in which we only have the following kinds of rules: $d(a) = b_1, \ldots, b_k$ (i.e., concatenation), $d(a) = b_1 + \cdots + b_k$ (i.e., disjunction), and $d(a) = b^*$ (i.e., Kleene star). Clearly, by introducing new virtual element labels any DTD can be transformed into a normalized one.

Given a normalized DTD $d$ we obtain an equivalent publishing transducer $\tau_d$ in $PT(FO,tuple,normal)$ as follows. We define a 4-ary relation $R$ consisting of tuples of the form $(v_1, a, v_2, b)$, with the meaning that there is an edge between the $a$-labeled node $v_1$ and the $b$-labeled node $v_2$. We assume a special constant $\text{root}$. It is now straightforward to define an FO-sentence $\phi_d$ that checks whether the graph rooted at $\text{root}$ is a tree and whether it conforms to $d$. Note that in addition to the tree rooted in $\text{root}$, $R$ can also contain other graphs disjoint from that tree. If an instance does not satisfy $\phi_d$ then the transducer will output a constant tree in $L(d)$. Otherwise, it will use the edge relation in $R$ to construct the encoded tree rooted at $\text{root}$. We omit the concrete construction of $\tau_d$. To output the original (i.e., not necessarily normalized) DTD it suffices to make all newly introduced labels virtual.

Given an extended DTD ($\Sigma', d, \mu$), the transducer then first tests whether the relation $R$ encodes a $\Sigma'$-tree in $L(d)$ as before. But if so, it outputs the tree with labels projected on $\Sigma$.

We next exhibit an example of a DTD not definable in $PT(CQ,relation,virtual)$. 

The proof exploits the monotonicity of CQ. Consider the DTD $d$ with $P(a) = b_1 + b_2$ and $P(b_i) = \varepsilon$ for $i = 1, 2$. Suppose by contradiction that there exists a $\tau$ in $PT(CQ, \text{relation}, \text{virtual})$ defining $L(d)$. Then, it is easy to argue that the transition rules in $\tau$ must contain transition rules of the form: $(q, a) \rightarrow (q_1, b_1, \varphi_1(\vec{x}_1; \vec{y}_1))$, $(q_2, b_2, \varphi_2(\vec{x}_2; \vec{y}_2))$ where $\varphi_1$ and $\varphi_2$ are CQ queries. Consider the following two trees in $L(d)$: $t_1 = a(b_1)$ and $t_2 = a(b_2)$. By assumption, there exist relational instances $I$ and $I'$ such that $\tau_\tau(I_1) = t_1$ and $\tau_\tau(I_2) = t_2$. Since $\varphi_1$ and $\varphi_2$ are monotonic, $\tau_\tau(I_1 \cup I_2)$ entails a tree where both a $b_1$ and a $b_2$ child occur below the root $a$. This obviously violates the DTD $d$ and contradicts our assumption. We remark that CQ can be replaced by any monotone query language.

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 4 and Proposition 4 for $PT(IFP, \text{tuple}, \text{normal})$ and $PT(FO, \text{relation}, \text{virtual})$ also provide insight for the expressive power of $\text{DBMSXMLGEN}$ and ATG, respectively. The result below settles the issue for $PT_{nr}(IFP, \text{tuple}, \text{normal})$ (SQL_mapping, SQL/XML), $PT_{nr}(FO, \text{tuple}, \text{normal})$ (FOR-XML, XPERANTO) and $PT_{nr}(CQ, \text{tuple}, O)$ (annotated XSD, RDB_mapping, TreeQL).

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

**Proposition 6.** When treated as relational query languages, (1) $PT_{nr}(CQ, \text{tuple}, O) = UCQ$; (2) $PT_{nr}(FO, \text{tuple}, O) = FO$; and (3) $PT_{nr}(IFP, \text{tuple}, O) = IFP$.

**Proof.** The proof is referred to the Appendix.

Tree generation. The proof for Proposition 5(1, 2) remains intact for nonrecursive transducers. As a result, $PT_{nr}(CQ, \text{tuple}, \text{normal}) \subset PT_{nr}(FO, \text{tuple}, \text{normal}) \subset PT_{nr}(IFP, \text{tuple}, \text{normal}) \subset PT_{nr}(CQ, \text{tuple}, \text{virtual})$. Theorem 4 tells us that over unordered trees fixed-depth FO-transduction (resp. IFP-transduction) is equivalent to $PT_{nr}(FO, \text{tuple}, O)$ (resp. $PT_{nr}(IFP, \text{tuple}, O)$).

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 5(6) states that $PT(FO, \text{relation}, \text{virtual}) = PT(IFP, \text{relation}, \text{virtual})$. 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 $PT(L, S, O)$ and nonrecursive $PT_{nr}(L, S, O)$ characterizing existing publishing languages, we have provided (a) a complete picture of the membership, equivalence and emptiness problems, (b) a comprehensive expressiveness analysis in terms of both querying and tree
generating power, as well as a number of separation and equivalence results. We expect these results will help the users decide what publishing languages to use, and database vendors develop or improve commercial XML publishing languages.

The main results for the static analyses and 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 \( L \), 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. W.r.t. expressiveness, we leave the following as open questions: the relationship between PT(CQ, relation, \( O \)) and PT(IFP, tuple, \( O \)) (where \( O = \{ \text{normal, virtual} \} \)) w.r.t. relational expressiveness; and PT(IFP,tuple,virtual) versus PT(IFP,relation,normal) w.r.t. their tree generation power.

Further, the relationship between publishing transducers and XML-to-XML transformation languages such as, e.g., 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 the ACM Digital Library by visiting the following URL: http://www.acm.org/pubs/citations/journals/tods/20YY-V-N/p1-url.

REFERENCES


A. PROOF OF PROPOSITION 1

Proposition 1. Let \( \tau \) be a publishing transducer in \( PT(L, S, O) \). Let \( I \) be an instance.

(1) The \( \tau \)-transformation on \( I \) always terminates and returns a unique \( \Sigma \)-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 \( L \) is CQ, FO or IFP, and \( O \) is normal or virtual.

(3) There is a publishing transducer \( \tau_1 \) in \( PT(CQ, \text{tuple}, \text{normal}) \) and a family of instances \( (I_n)_{n \in \mathbb{N}} \), such that the size of each \( I_n \) is \( O(n) \) and the size of \( \tau_1(I_n) \) is at least \( 2^n \).

(4) There is a publishing transducer \( \tau_2 \) in \( PT(CQ, \text{relation}, \text{normal}) \) and a family of instances \( (J_n)_{n \in \mathbb{N}} \), such that the size of each \( J_n \) is \( O(n) \) and the size of \( \tau_2(J_n) \) is at least \( 2^{2^n} \).

Proof. (1) We first show that the \( \tau \)-transformation on \( I \) always terminates and returns a unique \( \Sigma \)-tree \( \tau(I) \). Assume, by contradiction, that the \( \tau \)-transformation on \( I \) induces an infinite tree \( \xi \) in \( \text{Tree}_{Q \times \Sigma} \). Then there exists an infinite path \( \rho \) from the root of \( \xi \), on which each node \( v \) is labeled with \( (q, a) \in Q \times \Sigma \), carrying a local store \( \text{Reg}_a(v) \). Since \( \text{Reg}_a(v) \) is a relation over the data in \( I \) and constants in \( \tau \), there are at most exponentially many such relations. Furthermore, both \( Q \) and \( \Sigma \) are finite. Hence on path \( \rho \) there must exist two nodes \( u \) and \( v \) such that \( u \) is a
A binary relation $R$ that specifies the edges of a graph. That is, $R(a,b)$ indicates that there is an edge from $a$ to $b$. We define $\tau_I$ for $R$ such that given an instance $I$ of $R$, it expands the graph $G$ specified by $I$ into a tree following a "top-down" process. More specifically, let $\tau_I = (Q_1, \Sigma_1, \Theta_1, q_0, \delta_1)$, where $Q_1 = \{q_0, q_1\}$, $\Sigma_1 = \{\tau, a\}$, and $\delta_1$ is given below. A unary tuple register $Reg_a$ is associated with each $a$-element. The transition rules are given as follow:

\[
\begin{align*}
\delta_1(q_0, r) &\rightarrow (q, a, \phi(x;\emptyset)), \quad \text{where } \phi(x;\emptyset) = \exists y R(x,y); \text{ and}
\delta_1(q, a) &\rightarrow (q, a, \phi_1(x;\emptyset)), \quad \text{where } \phi_1(x;\emptyset) = \exists y Reg_a(y) \wedge R(y,x).
\end{align*}
\]

Initially, $\delta_1(q_0, r)$ creates a distinct $a$-child $u$ for root$(t)$ for each vertex $g$ in $G$ that has outgoing edges, storing $g$ in the register $Reg_a(u)$. Then $\delta_1(q, a)$ expands the tree as follows: for each current leaf node $u$, each vertex $v \in Reg_a(u)$, and each vertex $w$ with $(v, w) \in G$, $\tau_1$ spawns a distinct $a$-child of $u$ carrying $w$ in its register. Since $\tau_1$ is recursively defined, the process continues until the graph can no longer be expanded due to the stop condition. Let $L_n$ be the graph \{$(a_0, b_1^n)$, $(a_0, b_2)$, $(b_0^n, a_1)$, $(b_0^n, a_1)$, \ldots, $(b_0^n, a_n)$, $(b_2, a_1)$, $(b_2, a_1)$\}. That is, a "chain-of-diamond" shape graph with $4n$ edges. Then $\tau_1(L_n)$ is of size at least $2^n$ for each $n \in \mathbb{N}$.

(4) Intuitively, $\tau_2$ mimics an $n$-digit binary counter by generating chains, at each step incrementing the counter by 1 and making a duplicate copy of the chain. This
will yield a tree of depth $2^\alpha$ and size at least $2^{2\alpha}$. More specifically, we use a schema $R$ that consists of three relations:

(a) $\text{counter}(k, d, c)$ that gives the initial value of the counter, where $k$ indicates the $k$-th digit of the counter (with 0 the least significant), $d$ in $\{0, 1\}$ is the value of the digit, and $c$ is the carry from the last addition for the next digit (in $\{0, 1\}$). Define $C_n$ as the instance $\{(0, 0, 1), (1, 0, 0), \ldots, (n - 1, 0, 0)\}$.

(b) $\text{add}(d_1, d_2, d_3, d, c)$ that simulates a full adder circuit: $d_1 + d_2 + d_3$ yields $d$ with carry $c$. Define $A_n$ as $\{(0, 0, 0, 0, 0), (0, 0, 1, 1, 0), (0, 1, 0, 1, 0), (0, 1, 1, 0, 1), (1, 0, 0, 1, 0), (1, 1, 1, 0, 1), (1, 1, 1, 1, 1)\}$.

(c) $\text{next}(k, k')$ that is used to move from one digit to the next (mod $n$). Define $N_n$ as $\{(0, 1), \ldots, (n - 2, n - 1), (n - 1, 0)\}

So, for each $n$, $J_n$ is the instance $(C_n, A_n, N_n)$.

On the schema $R$ we define $\tau_2 = (Q_2, \Sigma_2, \Theta_2, q_0, \delta_2)$ in PT(CQ, relation, $O$), where $Q_2 = \{q_0, q\}$, $\Sigma_2 = \{r, a\}$, and $\delta_2$ associates a relation register $\text{Reg}_n(k, d, c)$ with each $a$-node, where $k, d, c$ have the same meaning as in $\text{counter}(k, d, c)$, as follows:

$$\delta_2(q_0, r) \rightarrow (q, a, \phi(\emptyset; k, d, c)); \quad (q, a, \phi(\emptyset; k, d, c)); \quad \text{where } \phi(\emptyset; k, d, c) = \text{counter}(k, d, c).$$

$$\delta_2(q, a) \rightarrow (q, a, \phi_1(\emptyset; k, d, c)); \quad (q, a, \phi_1(\emptyset; k, d, c)); \quad \text{where } \phi_1(\emptyset; k, d, c) = \exists d_1, c_1, k', d_2, c_2, (\text{Reg}_n(k, d_1, c_1) \land \text{Reg}_n(k', d_2, c_2) \land \text{next}(k', k) \land \text{counter}(k, d_3, c_3) \land \text{add}(d_1, d_2, d_3, d, c))$$

Initially, $\delta_2(q_0, r)$ creates two $a$-children of the root, each carrying a copy of $\text{counter}$ in its register. Then via $\phi_1$, $\delta_2(q, a)$ increments the counter by 1. More specifically, for the $k$-th digit $(k, d_1, c_1) \in \text{Reg}_n$, $(k', d_2, c_2)$ is the $(k + 1)$th digit and $c_2$ is the carry from the last round of addition. Note that for the initial value $(k, d_3, c_3)$ of the $k$-digit in the $\text{counter}$, $c_3 = 0$ if $k > 0$ and $c_3 = 1$ only if $k = 0$. The query $\phi_1$ sets the $k$-th digit to $(k, d, c)$, such that $d_1 + c_2 + c_3$ comes up to $d$ with carry $c$. Since $c_3$ is 0 for any $k > 0$, it does not affect $d_1 + c_2 + c_3$, but it adds 1 when $k = 0$. Thus $\phi_1$ adjusts the $k$-th digit by using the carry from the last round, and adds 1 to the 0th digit. In this way each $a$-element chain starting from the root increments the counter, 1 at each time, from 1 to $2^n$. Furthermore, each node has two copies of the chains below it. Putting these together, we have that $\tau_2(J_n)$ is of size at least $2^{2n}$.

B. PROOFS FOR SECTION 6.2

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 $\text{LINDATALOG}$, which we have seen in the proof of Theorem 2. It consists of datalog programs in which each rule is of the form: $p(x) \leftarrow p_1(x_1), \ldots, p_n(x_n)$, 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 $\text{ans}$ containing the result of the query expressed by the program. Another fragment is $\text{LINDATALOG(FO)}$ (see e.g., [Grädel 1992]) which extends $\text{LINDATALOG}$ 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.

B.1 Proof of Theorem 3

**Theorem 3.** When treated as relational query languages,

1. \( \text{PT}(\mathcal{L}, S, \text{virtual}) = \text{PT}(\mathcal{L}, S, \text{normal}) \),
2. \( \text{PT}(\text{CQ, tuple, } O) = \text{LINDATALOG} \),
3. \( \text{PT}(\text{FO, tuple, } O) = \text{LINDATALOG}(\text{FO}) \subset \text{nlogspace} \).
4. \( \text{PT}(\text{FO, relation, } O) \) and \( \text{PT}(\text{IFP, relation, } O) \) capture pspace.
5. \( \text{PT}(\text{IFP, tuple, } O) = \text{IFP} \).

Here the output \( O \) can be either virtual or normal.

**Proof.** (1) The statement is immediate as the relational query defined by a transducer remains the same irrespective of the removal of nodes with virtual tags.

(2) First, we establish the inclusion \( \text{PT}(\text{CQ, tuple, normal}) \subset \text{LINDATALOG} \). Consider a given publishing transducer \( \tau = (Q, \Sigma, \Theta, q_0, \delta) \), where \( Q = \{q_0, \ldots, q_n\} \) and \( \Sigma = \{a_0, a_1, \ldots, a_m\} \); here \( a_0 \) is the designated output tag. We assume the presence of two constants 0 and 1 which are interpreted by different elements in the domain. W.l.o.g., we may assume that all registers have the same arity, say \( k \). While we can easily encode the constants appearing in \( Q \) and \( \Sigma \), for ease of exposition, we simply assume that we have the constants in \( Q \) and \( \Sigma \) at our disposal.

Given \( \tau \), we define a \( (4 + 2k) \)-ary relation \( S \) w.r.t. the instance \( I \) as follows: the tuple \( (q, a, d, q', a', d') \in S \) iff there exists a rule in \( \delta \) of the form \( (q, a) \rightarrow \ldots (q', a', \phi(x; \theta)), \ldots \) such that \( (I \cup \{\text{Reg}_a = d\}) \models \phi(d') \). Here, \( d \) and \( d' \) are in \( (\text{adom}(I) \cup \text{adom}(\tau))^k \), where \( \text{adom}(I) \) denotes the active domain of \( I \), and \( \text{adom}(\tau) \) is the union of the constants appearing in the queries in rules of \( \delta \). Clearly, a tuple \( \bar{u} \) now belongs to output relation \( R_{e}(I) \) of \( \tau \) when there is a path in \( S \) from the root (i.e., state is \( q_0 \) and label is \( r \)) to \( (q, a_0, \bar{u}) \) for some state \( q \). Actually, we are only interested in simple paths, that is, paths without cycles. However, in this setting, whenever there is a path between two nodes there is also a simple one as we can always remove cycles. We now construct a LinDatalog-program \( \Pi_{e} \) that defines the same relation as \( \tau \) on any instance \( I \). Essentially, \( \Pi_{e} \) computes (i) all tuples in the transitive closure of \( S \) that are reachable from the root (as explained above); and (ii) extracts those that correspond to states that are tagged with \( a_0 \).

More specifically, assume that the initial rule of \( \tau \) is of the form \( (q_0, r) \rightarrow (q_1, a_1, \phi_1(\bar{x}; \theta)), \ldots, (q_m, a_m, \phi_m(\bar{x}; \theta)). \) Then, the program \( \Pi_{e} \) consists of the following initial rules. For each \( i \in [1, m] \) we have \( T(x, y, z, x', y', z') \leftarrow x = q_0 \land y = r \land z = z' \land x' = q_i \land y' = a_i \land \phi_i(z') \). Here, \( \phi_i(z') \) is simply the \( i \)-th CQ query in the initial rule of \( \tau \). We may assume that \( \phi_i \) is a CQ query solely over the relations in the schema of \( I \). Indeed, if it addresses \( \text{Reg}_e \) then we simply omit the corresponding initial rule in \( \Pi_{e} \) since it will always return \( \emptyset \) (recall that \( \text{Reg}_e = \emptyset \) by default).

Furthermore, for every rule \( (q, a) \rightarrow (q_1, a_1, \phi_1(\bar{x}; \theta)), \ldots, (q_m, a_m, \phi_m(\bar{x}; \theta)) \), \( \Pi_{e} \) contains the following linear recursive rules: for \( i \in [1, m] \)

\[
T(x, y, z, x', y', z') \leftarrow \exists x'', y'', z'' T(x, y, z, x'', y'', z'') \land x'' = q \land y'' = a \land x' = q_i \land y' = a_i \land \phi_i(z')
\]
where \( \phi'_i \) is obtained from \( \phi_i \) by replacing each occurrence of \( \text{Reg}_a(\bar{u}) \) by \( \bar{u} = \bar{u}' \).

Finally, the answer query \( \text{ans} \) is defined as \( \text{ans}(\bar{z}') \leftarrow \exists \bar{x}. x'T(q_0, r, \bar{z}, \bar{z}', a_o, \bar{z}') \).

It remains to show \( \text{LinDataLog} \subseteq \text{PT}(\text{CQ,tuple,normal}) \). Since we have assumed the existence of two different constants we can encode all IDB predicates by a single one by increasing the arity. For instance, the binary relation \( S \) can be extended to the 4-ary relation \( S \) as follows: \( R_1(x, y) \equiv S(x, y, 0, 1) \), \( R_2(x, y) \equiv S(x, y, 1, 0) \), and \( R_3(x, y) \equiv S(x, y, 1, 1) \). Then, it is not hard to see that any program \( \Pi \) in \( \text{LinDataLog} \) can be written in the following normal form, possibly by adding some equality conditions on the variables:

\[
S(\bar{y}) \leftarrow \phi_i(\bar{x}, \bar{y}) \quad \text{and} \quad S(\bar{y}) \leftarrow S(\bar{z}_j), \psi_j(\bar{u}_j, \bar{z}_j, \bar{y}), \quad \text{and} \quad \text{ans}(\bar{y}') \leftarrow \alpha(\bar{w}, \bar{y}')
\]

where \( \phi_i \), for \( i \in [1, m] \) and \( \psi_j \), for \( j \in [1, n] \) are CQ queries over the EDB relations. In contrast \( \alpha \) is a CQ query that contains a single occurrence of the IDB relation \( S \) and arbitrarily many occurrences of EDB relations.

Given a \( \text{LinDataLog} \)-program \( \Pi \) in normal form, we now define an equivalent publishing transducer \( \tau_{\Pi} \) in \( \text{PT}(\text{CQ, tuple, O}) \). More specifically, let \( \tau_{\Pi} = (Q, \Sigma, \Theta, q, \delta) \) where \( Q = \{q_0, q_1\} \), \( \Sigma = \{r, a_o\} \cup \{a_{ij} \mid i \in [1, m], j \in [1, n]\} \), \( \delta \) consists of the following rules from which the arity function \( \Theta \) can easily be determined (i.e., \( \Theta(r) = 0 \), \( \Theta(a_{ij}) = |\bar{y}'| \) and \( \Theta(\bar{a}_{ij}) = |\bar{y}| \) for \( i \in [1, m] \), \( j \in [1, n] \):

\[
\delta(q_0, r) = (q_1, a_{11}, \psi_{11}(\bar{y}; 0)), \ldots, (q_1, a_{1n}, \psi_{1n}(\bar{y}; 0)), \ldots, (q_1, a_{mn}, \psi_{mn}(\bar{y}; 0)), \ldots
\]

\[
\delta(q_1, a_{ij}) = (q_1, a_{11}, \psi_{11}(\bar{y}; 0)), \ldots, (q_1, a_{1n}, \psi_{1n}(\bar{y}; 0)), \ldots, (q_1, a_{mn}, \psi_{mn}(\bar{y}; 0)), \ldots
\]

where \( \psi_{kj}(\bar{y}) = \exists \bar{x}. xT(q_0, r, \bar{z}, \bar{z}'', a_o, \bar{z}') \).

Clearly, \( R_{\tau}(I) \) equals the result of \( \Pi \) computed on \( I \). We therefore may conclude the equivalence of \( \text{PT}(\text{CQ,tuple,normal}) \) and \( \text{LinDataLog} \).

(3) The proof of equivalence is similar to the proof of (2). As \( \text{LinDataLog}(\text{FO}) \) captures \( \text{nlogspace} \) on ordered structures, \( \text{PT(FO,tuple,O)} \) is in \( \text{nlogspace} \). As the class of all databases with an even domain is definable in \( \text{nlogspace} \) but not in \( \text{IPF} \) [Libkin 2004; Flum and Ebbinghaus 1999], and \( \text{PT(FO,tuple,O)} \) can be defined in \( \text{IPF} \) (see (5) below), it follows that \( \text{PT(FO,tuple,O)} \subseteq \text{nlogspace} \) is strict.

(4) Let \( \tau = (Q, \Sigma, \Theta, q_0, \delta) \) be in \( \text{PT(IEF, relation, output)} \). We first show how to solve the recognition problem in \( \text{NPSPACE} \) (recall that \( \text{NPSPACE} = \text{PSPACE} \) [Papadimitriou 1994]). The main underlying idea is the following. To check whether a tuple \( \bar{u} \) belongs to \( R_{\tau}(I) \), it suffices to guess a path in the tree \( \tau(I) \) leading to an \( a_e \)-labeled node containing \( \bar{u} \) in its register. It is therefore not necessary to construct the complete output tree. For ease of exposition, we assume that all registers have the same arity. As before, the latter can easily be achieved extending to the largest arity and padding.

During computation the tape contains the tuple \((q, a, R)\). Here, \( q \), \( a \), and \( R \) are the current state, label, and contents of the local store of the current node, respectively. On input \( I \), the algorithm starts by writing \((q_0, r, \emptyset)\) on its tape, that
is, the left-hand side of its start rule together with the current contents of the local store, which is empty at the beginning. The algorithm now operates as follows:

1. remove the triple \((q,a,R)\) from the tape;
2. if \(a = a_n\) and \(\bar{u} \in R\), then accept;
3. otherwise let \((q,a) \rightarrow (q_1,a_1,\phi_1(\bar{x}_1;\bar{y}_1)),\ldots,(q_k,a_k,\phi_k(\bar{x}_k;\bar{y}_k))\) be a rule in \(\delta\). Non-deterministically pick an \(i \in [1,k]\) and write \((\bar{q}_i,a_i,\phi_i(I,R))\) on the tape. Here, \(\phi_i(I,R)\) is the result of evaluating \(\phi_i\) over \(I\) with the local store interpreted by \(R\). Note that the latter can be done in \(\text{PSpace}\) [Libkin 2004]. Go to step (1).

It remains to argue the correctness of the algorithm. Suppose the algorithm accepts. Then we need to show that there is a node in the output tree labeled \(a_n\) with \(\bar{u}\) in its register. Assume the algorithm guesses the tuples \((q_1,a_1,R_1),\ldots,(q_n,a_n,R_n)\) with \(a_n = a_o\). A problem is that this sequence can not appear as such in the branch of the output tree as there can be \(i\) and \(j\) with \(i < j\) and \(q_i = q_j, a_i = a_j,\) and \(R_i = R_j\) which stops the computation of the transducer at step \(j\). However, in this case the sequence \((q_1,a_1,R_1),\ldots,(q_i,a_i,R_i),(q_{j+1},a_{j+1},R_{j+1}),\ldots,(q_n,a_n,R_n)\) would also lead to acceptance of the algorithm. We can repeat this argument and remove all duplicate triples ending up with a sequence without repeated triples which is realizable in a branch in the output tree. Conversely, when \(\bar{u}\) is in the relation computed by \(\tau\), there is a branch in the output tree leading to an \(a_l\)-labeled node containing \(\bar{u}\). The algorithm can now guess the sequence of triples leading to this node. In addition, the algorithm uses only space polynomial in the size of \(I\).

We next sketch how every \(\text{PSPACE}\) computable query can be defined by a transducer in \(\text{PT(FO,relation),O}\). First, it is instrumental to be more precise about what it means for a recognition problem to belong to a complexity class for unordered databases. Indeed, every encoding of the database on the input tape of a Turing Machine induces an ordering. Actually, it is more accustomed to first determine an ordering of the elements of the database and then use this ordering in obtaining a unique encoding. A \(\text{PSPACE}\) Turing Machine accepting a recognition problem is therefore independent of the ordering at hand and is order-invariant (cf. Ebbinghaus and Flum [Flum and Ebbinghaus 1999], Definition 7.5.12 and preceding discussion). The outline of the proof is as follows. We show that a transducer can compute a total order on the domain of the input database. Then, we make use of the known result that on ordered databases, partial fixpoint logic captures \(\text{PSPACE}\) [Vardi 1982]. The simulation of partial fixpoint logic with a transducer is straightforward and is therefore omitted. By the discussion above, it does not matter which ordering we use. Actually, the transducer does not define a single order but, instead, defines all possible orderings in parallel. Intuitively, the transducer starts by making every element the smallest element of a particular ordering. Then it continues by adding new elements which are always larger than elements already present in the ordering. We use a 3-ary relation, in which the last two columns contain the ordering relation, and the first column is used to pick a new element from the domain. The following formula then computes all orderings in parallel:

\[
\varphi(x;y_1,y_2) \equiv \psi_1 \land (\psi_2 \lor \exists z_1 \exists z_2 (\psi_3 \lor \psi_4)), \quad \text{where} \quad \psi_1 \equiv \neg \exists z_1 \exists z_2 \text{Reg}(z_1,z_2,x),
\]

\[
\psi_2 \equiv y_1 = x \land y_2 = x, \quad \psi_3 \equiv \text{Reg}(z_1,y_1,y_2), \quad \psi_4 \equiv y_2 = x \land \text{Reg}(z_1,y_1,z_2).
\]

The formula says the following: \((\psi_1)\) \(x\) does not occur in the ordering up to now;
otherwise. The transducer consists of the rules: 

(ψ_2) x is less than or equal to itself; (ψ_3) the new ordering extends the previous one; and (ψ_4) all elements in the ordering are smaller than the new element x. At this point, every leaf node contains an ordering of the domain and now starts simulating the partial fixpoint logic formula corresponding to the given \( \text{pspace} \) TM. As the machine is order-independent, every branch ends up with the same relation.

(5) Clearly, PT(IFP,tuple,O) can simulate IFP. The proof of the other direction is similar to the proof of (2). Indeed, it only needs to be observed that PT(IFP,tuple,O) can be defined in LinDatalog(IFP) (the extension of LinDatalog(FO)) that allows arbitrary IFP-formulas over \( \text{FO} \) predicates in the body of the rules) which in turn can be defined in IFP itself. The latter holds as LinDatalog(IFP) is essentially the transitive closure of an IFP-definable relation and transitive closure can easily be defined in IFP itself. □

B.2 Proof of Proposition 4

**Proposition 4.** When treated as relational query languages,

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

The containment in statement (2) is proper if \( \text{nlogspace} \neq \text{ptime} \).

**Proof.**

1. This follows from the inclusion of LinDatalog in LinDatalog(FO) and the equivalence of those to PT(CQ, tuple, O) and PT(FO, tuple, O), respectively (Theorem 3(2,3)). Strictness follows as LinDatalog can only express monotone queries.

2. This holds as PT(FO, tuple, O) equals LinDatalog(FO) (Theorem 3(3)) and PT(IFP, tuple, O) is IFP (Theorem 3(5)), and LinDatalog(FO) \( \subseteq \text{IFP} \). The containment is proper if \( \text{ptime} \neq \text{nlogspace} \) since on ordered structures PT(FO, tuple, O) captures \( \text{nlogspace} \) whereas PT(IFP, tuple, O) captures \( \text{ptime} \).

3,4 Statements (3) and (4) follow as PT(IFP, tuple, O) is IFP (Theorem 3(5)) and both of PT(FO, relation, O) and PT(IFP, relation, O) equal \( \text{pspace} \) (Theorem 3(4)), respectively. In addition, the containment is proper since on unordered databases, the parity query even is expressible in \( \text{pspace} \) but it is not definable in IFP.

5. It suffices to give a publishing transducer \( \tau \) in PT(CQ, relation, O) such that there exists no \( \tau' \) in PT(FO, tuple, O) that is equivalent to \( \tau \) when they are considered relational queries. The query expressed by \( \tau \) is over a directed graph \( E \) with three distinguished nodes \( c_1, c_2 \) and \( c_3 \). It computes the pair \((c_1, c_3)\) whenever there is a path from \( c_1 \) to \( c_3 \) which passes through \( c_2 \) and returns the empty set otherwise. The transducer consists of the rules:

\[
\begin{align*}
\delta(y_0, r) &= (q, a, \phi(\emptyset; y_1, y_2)), \quad \text{where } \phi(\emptyset; y_1, y_2) = E(y_1, y_2) \land y_1 = c_1, \\
\delta(q, a) &= (q, a, \phi_1(\emptyset; y_1, y_2)), \quad (q, a, o, \phi_2(\emptyset; y_1, y_2)), \quad \text{where } \\
&\phi_1(\emptyset; y_1, y_2) = \exists y (\text{Reg}(y_1, y) \land E(y, y_2)),
\end{align*}
\]

(\( \text{Reg} \) is the regular predicate over the relation \( E \).)
\[ \phi_2(y_1, y_2) = \text{Reg}(c_1, c_2) \land \text{Reg}(c_2, c_3) \land y_1 = c_1 \land y_2 = c_3 \]

We next argue that the above query cannot be expressed in \text{LinDatalog}(\text{FO}) and therefore not in \text{PT}(\text{CQ, tuple, } O) or \text{PT}(\text{FO, tuple, } O). The proof is similar to the proof of Theorem 7 in [Grädel 1992] showing that graph connectivity is not in \text{LinDatalog}(\text{FO}) via an Ehrenfeucht-Fraïssé game (EF-game) for transitive closure logic. The initial databases \(I_1\) and \(I_2\) are different but the rest of the argument is the same. Therefore, we only explain how to construct these and refer for the concrete EF-game to [Grädel 1992]. Assume that the query is definable in \text{LinDatalog}(\text{FO}) and let \(\xi\) be the corresponding transitive closure logic formula. Let \(k\) be the arity of \(\xi\) plus the quantifier-rank. Let \(p(\ell)\) be a chain of length \(\ell\).

Let \(I_1\) be the graph \(c_1p(2^{k+2})c_2p(2^{k+2})c_3p(2^{k+2})\) that is a chain starting with \(c_1\) followed by a path of length \(2^{k+2}\) to \(c_2\), followed in turn by a path of length \(2^{k+2}\) to \(c_3\) ended by a final path of length \(2^{k+2}\). Similarly, let \(I_2\) be the graph obtained from \(I_1\) by exchanging \(c_2\) and \(c_3\), that is, \(I_2 = c_1p(2^{k+2})c_3p(2^{k+2})c_2p(2^{k+2})\). Note that \(I_1\) satisfies the query whereas \(I_2\) does not.

(6) The inclusion is immediate. To obtain strictness it suffices to show that every query expressed by a transducer \(\tau\) in \text{PT}(\text{CQ, relation, } O) is monotone, whereas \(\text{FO}\) is not. That is, whenever \(I_1\) is an extension of \(I_0\), meaning that every tuple in a relation in \(I_0\) also belongs to the corresponding relation in \(I_1\), \(R_\tau(I_0) \subseteq R_\tau(I_1)\). By induction on the depth of the output tree, it can be shown that there is a mapping \(\mu\) from nodes in \(\tau(I_0)\) to nodes in \(\tau(I_1)\), such that for each node \(u \in \tau(I_0)\), \(u\) and \(\mu(u)\) carry the same label and \(\text{Reg}(u) \subseteq \text{Reg}(\mu(u))\). It then follows that \(R_\tau(I_0) \subseteq R_\tau(I_1)\).

(7) The proof of (5) already gives a publishing transducer that is in \text{PT}(\text{CQ, relation, } O) but is not expressible in \text{PT}(\text{FO, tuple, } O). \qed

C. PROOF OF PROPOSITION 6

PROPOSITION 6. When treated as relational query languages, (1) \(\text{PT}_{nr}(\text{CQ, tuple, } O) = \text{UCQ}\); (2) \(\text{PT}_{nr}(\text{FO, tuple, } O) = \text{FO}\); and (3) \(\text{PT}_{nr}(\text{IFP, tuple, } O) = \text{IFP}\);

PROOF. (1) For any UCQ query \(\varphi = \varphi_1 \cup \ldots \cup \varphi_n\), where for \(i \in [1, n]\), \(\varphi_i\) is a CQ query without union, we can define an equivalent transducer \(\tau_{\varphi_i}\) in \(\text{PT}_{nr}(\text{CQ, tuple, normal})\) with a single rule \((q_0, r) \rightarrow (q, a_0, \varphi_i(q; \emptyset)), \ldots, (q, a_0, \varphi_n(q; \emptyset))\).

Conversely, for each transducer \(\tau\) in \(\text{PT}_{nr}(\text{CQ, tuple, virtual})\) we show that it is equivalent to a UCQ query. Let \(G_\tau\) be the dependency graph of \(\tau\) as defined in Section 3. We denote by \(P(a_o)\) all paths in \(G_\tau\) starting from \(v(q_0, r)\) and leading to a node \(v(q, a_o)\) for some \(q \in Q\), where \(a_o\) is the designated output tag. For each path \(\rho \in P(a_o)\), we define CQ query \(\lambda(\rho)\) that is the composition of the CQ queries along \(\rho\), along the same lines as in the proof of Theorem 2. Then for any instance \(I\), the relation induced by \(\tau\) on \(I\) is equal to \(\bigcup_{\rho \in P(a_o)} \lambda(\rho)\).

(2) The inclusion \(\text{FO} \subseteq \text{PT}_{nr}(\text{FO, tuple, normal})\) is straightforward. Similar to case (1) we can verify that \(\text{PT}_{nr}(\text{FO, tuple, virtual}) \subseteq \text{FO}\). More specifically, given a transducer \(\tau\) in \(\text{PT}_{nr}(\text{FO, tuple, virtual})\) with designated output tag \(a_o\), we obtain that \(\tau\) and the FO-query \(\bigcup_{\rho \in P(a_o)} \lambda(\rho)\) define the same relation on any instance \(I\).

(3) The proofs of the inclusions \(\text{IFP} \subseteq \text{PT}_{nr}(\text{IFP, tuple, normal})\) and \(\text{PT}_{nr}(\text{IFP, tu-}
ple, virtual) \subseteq \text{IFP} are completely analogous to those of case (1). \qed