Representing and Querying XML with Incomplete Information

Serge Abiteboul*
INRIA-Rocquencourt
Serge.Abiteboul@inria.fr

Luc Segoufin
INRIA-Rocquencourt
Luc.Segoufin@inria.fr

Victor Vianu†
U.C. San Diego
vianu@cs.ucsd.edu

Abstract

We study the representation and querying of XML with incomplete information. We consider a simple model for XML data and their DTDs, a very simple query language, and a representation system for incomplete information in the spirit of the representations systems developed by Imielinski and Lipski for relational databases. In the scenario we consider, the incomplete information about an XML document is continuously enriched by successive queries to the document. We show that our representation system can represent partial information about the source document acquired by successive queries, and that it can be used to intelligently answer new queries. We also consider the impact on complexity of enriching our representation system or query language with additional features. The results suggest that our approach achieves a practically appealing balance between expressiveness and tractability. The research presented here was motivated by the Xyleme project at INRIA, whose objective it to develop a data warehouse for Web XML documents.

*This author supported in part by R.N.R.T.
†Contact author: vianu@cs.ucsd.edu, 858-534-6227 (phone) 858-534-7029 (fax), CSE 0114, UCSD, La Jolla, CA 92093-0114.
This author supported in part by the National Science Foundation under grant number IIS-9802288.

1 Introduction

In a warehouse for XML data – that we call an (XML) Webhouse – information is collected from Web sites and stored in a centralized fashion. In practice, the information held in a Webhouse is never complete. This is due to many reasons: limited storage capacity, the dynamic nature of Web data, expiration of data, etc. Thus, Webhouses have to deal with *incomplete information*. We view here the Webhouse as an incomplete repository of XML documents that is continuously enriched by exploration of Web sources, in response to queries or by crawling the Web. Documents may be entirely missing or may be partially available.

At any given time, the Webhouse contains a *representation* of incomplete information about XML documents satisfying given Document Type Definitions (DTDs). The incomplete information about the XML documents is *enriched* using answers to queries against remote documents.

When a query is posed against the Webhouse, two courses of action are possible:

(1) The first alternative is to answer the query as best possible using the incomplete information already available. Since the data is not entirely known, the answer is not always complete. We represent the answer using the same representation as the one we use to describe our incomplete knowledge of the sources. Two important variations are the *sure* and *possible* answer modalities, i.e., providing the pieces of information that *surely* hold in all possible answers, or similarly those that *possibly* hold.

(2) The second, similar to a mediator approach, is to seek from the sources the additional information needed to fully answer the query. In this case, we would like to use the incomplete information as a guide for determining what additional exploration of Web sources is needed by taking as much as possible advantage of the data already available.

This paper introduces *representations* for incomplete information, studies their incremental *maintenance* and addresses the issue of *answering queries* posed against an incomplete Webhouse.

The quest for simplicity and efficiency was the main motivating factor in our choice of model, and has led to many limitations. The most notable are (i) a very limited query language based on pattern matching and simple selection conditions on data values, (ii) the use of simplified DTDs that ignore the ordering of components in an element, and (iii) the assumption that XML elements have persistent identifiers. Despite its limitations, we believe that our framework captures a broad range of situations of practical interest. Our examples illustrate some of them.

The representation of incomplete information is quite natural. It uses partial XML trees to represent the data available, and typing information in the style of DTDs to represent the data that is still missing. The typing we use for the missing data is interesting in its own right and reminiscent of some extensions already proposed for DTDs. These include specifying ranges for some data value, e.g., *price* $\geq$ *100*, and a *specialization* mechanism allowing to define the type of a given element name depending on the context where it appears. We call such a representation an *incomplete tree*. As illustrated by our examples, incomplete trees exhibit in a user-friendly way the partial information available as well as the missing information, and can be itself naturally represented and browsed as an XML document.

We show that, given a simplified DTD satisfied by the input and a sequence of query/answer pairs on the input, the partial knowledge about the input can be represented by an incomplete tree which can be maintained incrementally in PTIME. Given a query and an incomplete tree, the set of possible answers to the query can again be represented by an incomplete tree computable in PTIME. In particular, this shows that incomplete trees form a strong representation system with respect to our queries. Furthermore, it can be checked in PTIME whether a given query can be fully answered with the data currently available. For the case when the available data is not sufficient, we provide a PTIME algorithm that uses the incomplete tree to determine which additional information is needed from the sources in order to fully answer the query, and provides a nonredundant set of queries for retrieving the information.

Although incomplete trees can be incrementally maintained in PTIME, their size can grow exponentially in the overall sequence of query/answer pairs. We discuss several ways of dealing with the exponential blowup. We consider an extension to incomplete trees called *conjunctive*, that intuitively adds a form of alternation. With this extension, the size of incomplete trees is shown to remain polynomial with respect to the entire sequence of query/answer pairs. However, many of the manipulations needed in handling incomplete information now become exponential in the representation. For example, checking emptiness of conjunctive trees becomes NP-hard, whereas it is in PTIME for regular incomplete trees. As an alternative approach, we exhibit a restriction of the input DTD and the queries ensuring that incomplete trees remain polynomial in the overall sequence of query/answer pairs. Thus, all manipulations remain polynomial.

Regardless of the complexity-theoretic bound, we exhibit two approaches for dealing with cases when the incomplete tree grows too large to be practical. The first approach consists of asking a small set of additional queries chosen so as to provide precisely the critical information needed to eliminate some of the unknown information and shrink the incomplete tree. We prove that the queries can be chosen such that the incomplete tree remains of polynomial size with respect to the entire sequence of query/answer pairs and input DTD. This approach can be used heuristically whenever needed. The second approach is a heuristic for gracefully loosing some of the information represented in the incomplete tree, thus allowing to trade off accuracy against size in incomplete trees. Once again, we show that this approach can be used to keep the incomplete tree polynomial in the sequence of query/answer pairs.

We argue that our core model provides a practically appealing starting point for dealing with incomplete information in XML Webhouses. However, the model has many limitations. We discuss the impact of various extensions to the model, and show that even minor extensions lead to significant difficulties. The representation system may no longer exhibit in a user-friendly way the partial information available, or there may no longer be a strong representation system. Most seriously, various decision problems, such as whether a query can be answered given the information currently available, have very high complexity or become undecidable. Some of the extensions concern the query language: the extra features include optional and negative subtrees in query patterns, constructed answers, recursive path expressions, data joins, and powerful restructuring modeled by the k-pebble transducers of [MSV00]. We also discuss other extensions to the framework such as the persistent ids assumption and the issue of order.

The paper is organized as follows. Section 2 introduces our formal model for XML, DTD, the various types we use, as well as the representation system. Section 3 deals with the acquisition and the use of incomplete information, and with the various approaches to the exponential blowup of incomplete trees. Section 4 discusses extensions and associated complexity and undecidability results. The paper ends with brief conclusions. An appendix provides several examples, definitions, algorithms, and proof sketches.

Related work Incomplete information has been of interest early on in database systems [Cod75]. Much of the focus has been on searching for the "correct" semantics for queries applied to incomplete databases [Zan84, Rei86, Var86]. Usually, the semantics of incompleteness is approached from two perspectives, either a *closed world assumption* (CWA) or an *open world assumption* (OWA). Intuitively, CWA states that nothing holds unless explicitly stated in the incomplete database, whereas OWA states that anything not ruled out is possible. Interestingly, incomplete trees reconcile the two approaches by allowing a combination of the two semantics. They allow to describe with flexible precision the missing information, by stating that some facts are not in the document (CWA) but also that some data still ignored may exist (OWA).

A landmark paper [IL84] laid the formal groundwork for incomplete databases with nulls of the "unknown" kind, and introduced the notion of strong representation system. The representation system we use is in the spirit of the c-tables of [IL84], but addresses a tree model instead of the relational model. Most importantly, we use a more benign form of incompleteness, which is possible because our query language is more restricted than the relational algebra they consider (e.g., it has no data joins).

The complexity of handling incompleteness was studied in many works, e.g., [IL84, Var86, AKG91]. The program complexity of evaluation is usually higher by an exponential than the data complexity [Cos83, Var82]. This was first noted in [HLY80, MSY81], as part of the study of nulls in weak universal instances.

The Webhouse scenario that we consider here is in the spirit of data warehousing, see e.g., [LZW+97]. Our technique is reminiscent of techniques used in answering queries using views, a much studied problem [LMSS95, CKPS95, RSU95]. The use of a model based on incomplete information to study this problem is proposed in [AD98]. The management of incompleteness due to data expiration is studied in [GMLY98].

Extensions of DTDs with specialization have been considered under various names and in various contexts, e.g., in [BM99, CDSS98, PV00]. We use the specialization mechanism in our representation system.

Perhaps closest to our work is the investigation in [KNS99], that studies incomplete information in semistructured data. However, their framework and results are quite different.

2 Formal Framework

We next present our core framework for Webhouses with incomplete information. We define in turn our model of XML documents and simplified DTDs, queries, and the representation system for incomplete information. We use as a running example the *catalog example* in Figure 1.

Data trees Our formal model abstracts XML documents as labeled trees. Our abstraction simplifies real XML documents in several ways, some of which are minor and others more substantial. For example, the model does not distinguish between attributes and subelements, a distinction often considered cosmetic. A more significant simplification is that out trees are unordered, whereas XML documents are ordered. We will discuss the issue of order in Section 4.

We use the following: an infinite set $\mathcal{N}$ of *nodes*; a finite set Σ of *element names* (labels); and a set $\mathbb{Q}$ of *data values*. We denote element names by $a, b, c...$, nodes by n, data values by v, possibly with sub and superscripts. We denote sets of labels by $A, B, C, ...$ etc. For simplicity, we assume that the set $\mathbb{Q}$ of data values is the rational numbers (any set equipped with a dense linear order would do). Our simplified model for XML is defined next.

Definition: *A* (data) tree *over Σ is a triple $\langle t, \lambda, \nu \rangle$, where: (1) t is a finite rooted tree with nodes from $\mathcal{N}$; (2) λ, the labeling function, associates a label in Σ to each node in t; and*
(3) ν, the data value mapping, assigns a value in $\mathbb{Q}$ to each node in t.

Data trees are denoted by $T, T', \ldots$ A *prefix* of $\langle t, \lambda, \nu \rangle$ is a data tree $\langle t', \lambda', \nu' \rangle$ where t' is a subtree of t containing the root, and λ', ν' are the restrictions of λ, ν to the nodes of t'. Note that two prefixes of the same tree share nodes in $\mathcal{N}$.

Tree types In XML, the structure of valid documents is described by DTDs. We use here a simplified version of DTDs that we call *tree type*. A tree type specifies, for each element name a, the set of element names allowed for children of nodes labeled a, together with some multiplicity constraint. We also specify a root name, a restriction that can easily be removed.

Consider the alphabet Σ of labels. We use the auxiliary notion of *multiplicity atom* to describe the children that nodes labeled a may have. A multiplicity atom is an expression $a_1^{\omega_1} \cdots a_k^{\omega_k}$ where the a_i are distinct labels in Σ and each ω_i is a symbol in $\{\star, +, ?, 1\}$.

Definition: *A* tree type τ *(over alphabet Σ) is a triple (Σ, r, τ) where $r \in \Sigma$ is a particular label called the* root, *and τ associates to each $a \in \Sigma$ a multiplicity atom $\tau(a)$ called the* type of a.

Satisfaction of a tree type (Σ, r, τ) by a data tree t, denoted $t \models \tau$, is defined in the obvious way as follows. The root of t is labelled r and for each node n in t labeled a, if $\tau(a)$ contains $a_i^{w_i}$, the number of children of n labelled a_i is restricted as follows:

$w_i = 1$	:	exactly one child is labelled a_i;
$w_i = ?$	:	at most one child is labelled a_i;
$w_i = +$	:	at least one child is labelled a_i.
$w_i = \star$	:	no restriction;

The set of trees satisfying τ is denoted by $rep(\tau)$.

We usually denote a tree type by τ when r is understood. In examples, we specify tree types as in the following example:

root: *catalog*
$$
\begin{array}{lll}
\textit{catalog} & \rightarrow & \textit{product}^{+} \\
\textit{product} & \rightarrow & \textit{name price cat picture}^{\star} \\
\textit{cat} & \rightarrow & \textit{subcat}
\end{array}
$$

Observe that we ommit the 1 in exponents, e.g., we write *name* for $name^1$. This tree type is represented graphically in the catalog example in tree form, with multiplicities placed on edges. See Figure 1 (a).

Queries We define a simple query language that selects prefixes of input trees. Although very limited, we claim that this is often sufficient in practice. The query basically browses the input tree down to a certain depth starting from the root, by reading nodes with specified element names and possibly selection conditions on data values. All nodes involved in the pattern are extracted (so there is no projection), as well as subtrees of specified leaves. The pattern may also specify the non-existence of nodes with a given label. We call such a query a *prefix-selection query* (ps-query). For instance, Queries 1 and 2 in Figure 1 (b,c) are ps-queries. Query 1 finds the name, price and subcategories of electronics products with price less than \$200. Query 2 finds the name and pictures of all cameras whose picture appears in the catalog.

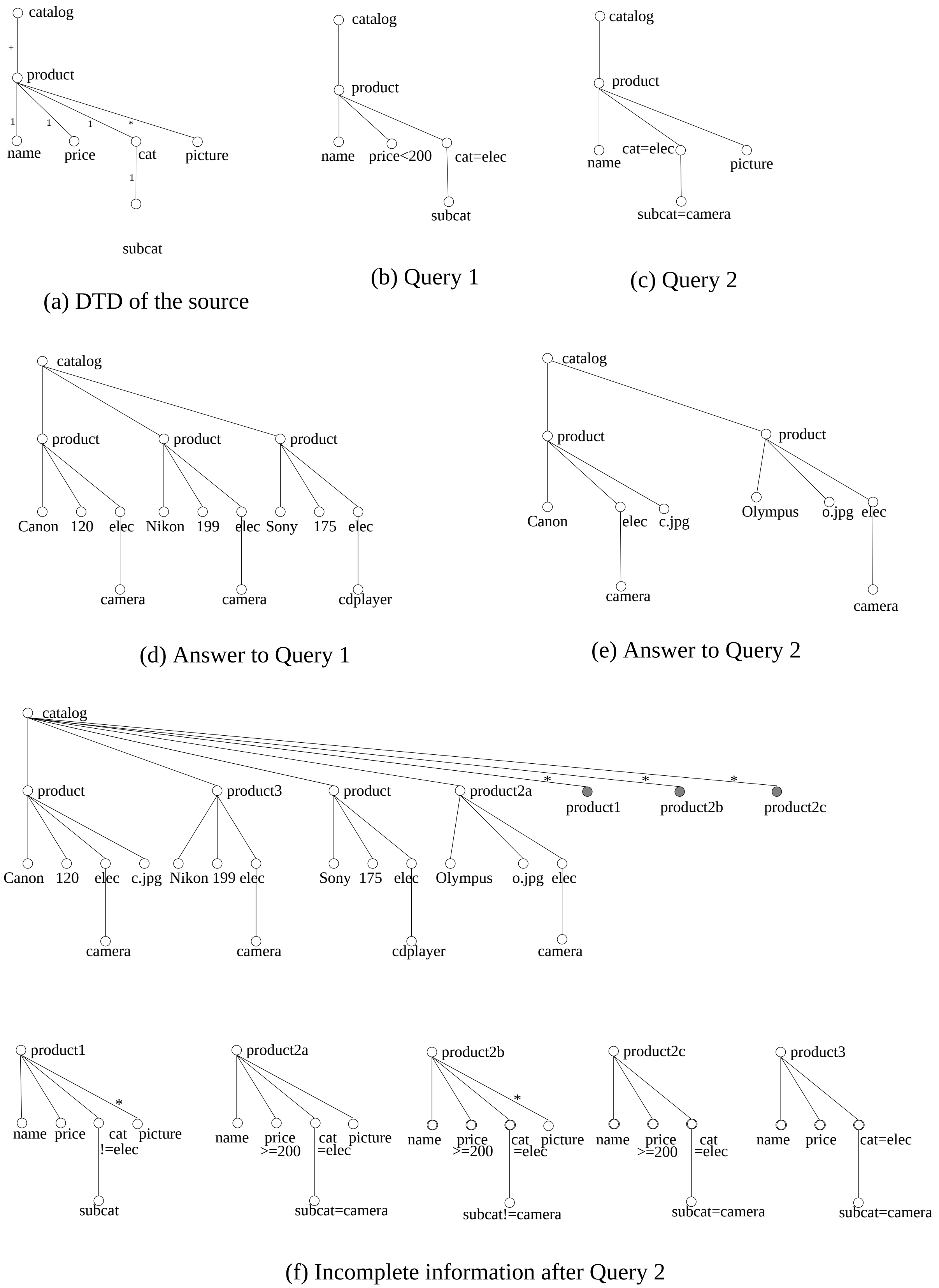
More formally, a ps-query is a labeled tree $\langle t, \lambda, \text{cond} \rangle$ where:

- t is a rooted tree;
- λ associates with each node a label in the extended alphabet $\Sigma \cup \{\neg a \mid a \in \Sigma\} \cup \{\overline{a} \mid a \in \Sigma\}$. Internal nodes can only have labels in Σ, and sibling internal nodes have distinct labels.
- *cond* associates to each node a *condition*, which is a Boolean combination of statements of the form $= v, \neq v, \leq v, \geq v, < v, > v$, where $v \in \mathbb{Q}$.

The negative labels indicate the absence of a node with that label among the children of a node in the pattern. A node adorned with a bar indicates that the entire subtree rooted at that node is extracted. Examples of queries are shown in Figure 1 and in Section A of the Appendix.

We next formalize the notion of answer to a query using the auxiliary concept of valuation. Given a ps-query $q = \langle t, \lambda, \text{cond} \rangle$ and an input data tree t', a *valuation ν from q to t'* is a mapping from the nodes of t into nodes of t' such that:
(1) each edge $\langle n, m \rangle$ in t is mapped by ν into an edge of t' unless $\lambda(m) = \neg a$ for some a;
(2) for each node n in t such that $\lambda(n) \in \{a, \overline{a}\}$, $\nu(n)$ has label a in t';
(3) for each n, the value of $\nu(n)$ in t' satisfies *cond*(n).
(4) if $\langle n, m \rangle$ is an edge in t and $\lambda(m) = \neg a$ then $\nu(n)$ has no child labeled a in t'.

The *answer* $q(t')$ is the prefix tree of t' consisting of the nodes n' which are in the image of some valuation ν from q to t', or are descendants of such a node with label $\overline{a}$, $a \in \Sigma$. Possible answers to Queries 1 and 2 in the catalog example are depicted in Figure 1 (d,e).



(a) DTD of the source

(b) Query 1

(c) Query 2

(d) Answer to Query 1

(e) Answer to Query 2

(f) Incomplete information after Query 2

Figure 1:

The following remark insists of an essential aspect of the model.

Remark 2.1 (Object identifiers) Consider a tree t and two queries q_1, q_2. The answers $q_1(t)$ and $q_2(t)$ are both prefixes of t. In particular, their roots are the same and they share nodes from the input t. This is an important aspect of the model, which amounts to having persistent node identifiers in the input and answers. □

Conditional tree types We next discuss our representation of incomplete information. The main idea is that at any given time the Webhouse knows, as a result of previous queries, a prefix of the full data tree representing the complete data. In addition, using the queries and the initial type definitions of the sources, there is partial knowledge about the missing portion of the full tree. Our representation of incomplete information includes the prefix tree known so far and the description of the missing information.

To describe the missing information we need to extend tree types in three ways: by allowing disjunctions of multiplicity atoms, by specifying *conditions* on data values, and by adding a *specialization* mechanism allowing to define several types for the same element name. For instance in Figure 1, after posing Query 1, we know that the missing data contains 2 new types of products: *product1* and *product2*, the first one for products which category is not "electrical", the second for products which price is greater that 200.

We next define formally our representation of incomplete information, extending the notion of tree type. Note that the data sources continue to be described by tree types, as previously defined.

Before introducing specialization, we define *simple conditional tree types*. A *condition* is defined as for queries. A *simple conditional tree type* over alphabet Σ is a pair (τ, cond) where:

- τ is a mapping associating to each $a \in \Sigma$ a disjunction $\tau(a)$ of multiplicity atoms; and,
- cond associates a condition to each $a \in \Sigma$ (the condition applies to the data value of nodes with label a).

The *set of trees* represented by a simple conditional tree type (τ, cond) is defined in the obvious manner and denoted $rep(\tau, \text{cond})$. We extend our notation for tree types to conditional tree types by allowing on right hand sides of productions disjunctions of multiplicity atoms, as in $a \rightarrow ab^\star \vee c^? d^+$.

Next we consider specialization, found useful in the context of DTDs for expressing structural properties that are dependent on the context of a node. It has been previously considered in [BM99, CDSS98, PV00]. Specialization is achieved by allowing several possible *types* for the same element name. This suggests the following definition. A specialization mapping σ is a mapping from some Σ' to some Σ, two sets of element names. It transforms a data tree T with names in Σ' into a data tree $\sigma(T)$ with names in Σ' in the obvious manner, by replacing each label a by $\sigma(a)$. We are now ready to define the most complex types used in the paper.

A *conditional tree type* over Σ and specialized alphabet Σ' is a triple $(\tau, \text{cond}, \sigma)$ where:

- (τ, cond) is a simple conditional tree over Σ' and
- σ is a specialization mapping from Σ' to Σ.

Intuitively, the labels in Σ' specialize the labels in Σ. In the catalog example (Figure 1 (f)), *product1*, *product3* would be elements of Σ' such that $\sigma(\textit{product}1) = \sigma(\textit{product}3) = \textit{product}$.

The semantics of conditional tree types is defined as follows. A data tree T over Σ is in $rep(\tau, \text{cond}, \sigma)$ iff there exists a tree T' in $rep(\tau, \text{cond})$ such that $T = \sigma(T')$.

Intuitively, there is a similarity between conditional tree types and unranked tree automata [AKMD98]. Both are used to define valid sets of trees, and the role of the specialized alphabet in conditional tree types is similar to that of states in a non-deterministic top-down tree automaton. The analogy does not fully go through because of the lack of order and the presence of data values in our trees. However, some of the flavor of the automata techniques carries through, and the sets of trees definable by conditional tree types have some properties similar to regular tree languages. For example, the sets of trees defined by conditional tree types are closed under union, intersection, and complement. A key technical point for our algorithms is testing emptiness of the set of trees satisfying a conditional tree type. An easy reduction to and from testing emptiness of context-free grammars shows that :

Lemma 2.2 Checking emptiness of $rep(\tau, \text{cond}, \sigma)$ is PTIME-complete.

Incomplete trees As discussed earlier, the representation of incomplete information consists of two aspects: a prefix of a full data tree, and information on the missing portion of the tree. The missing portion is described by some conditional tree type $(\tau, \text{cond}, \sigma)$ over alphabet Σ, with a specialized alphabet Σ'. In order to fully capture the incomplete information, it is not sufficient to provide the labeling of the prefix tree with element names from Σ; instead, we must provide the possible interpretation of the nodes in terms of the specialized alphabet Σ'.

We are now ready to define incomplete trees.

Definition: *An* incomplete tree *consists of the following:*

- *a conditional tree type* $(\tau, cond, \sigma)$, *over* Σ, *with specialized alphabet* Σ';
- *a data tree* $T_d = \langle t, \lambda, \mu \rangle$ *over* Σ; *and,*
- *a* data labeling mapping λ' *associating to each node* n *of* T_d *a subset of* Σ' *included in* $\sigma^{-1}(\lambda(n))$, *such that* $\lambda'(n_1) \cap \lambda'(n_2) = \emptyset$ *for* $n_1 \neq n_2$.

We denote incomplete trees by $\mathbf{T}$, $\mathbf{T}_1$, $\mathbf{T}_2$, etc. In the catalog example, the incomplete tree resulting from Queries 1 and 2 is represented in Figure 1 (f). The grey nodes and the associated information describe the missing portion of the input tree.

The set of trees represented by an incomplete tree $\mathbf{T}$ as in the definition consists of the trees T over Σ for which there exists a data tree T' over Σ' such that: $T = \sigma(T')$; T' satisfies (τ, cond); T' has t as a prefix; and, every node n of t is labeled in T' by a symbol in $\lambda'(n)$, and symbols in $\lambda'(n)$ do not label any nodes other than n. The set of trees represented by $\mathbf{T}$ is denoted $rep(\mathbf{T})$. Example A.4 further illustrates incomplete trees.

Given an incomplete tree, it is often of interest to check whether certain facts or sets of facts are *certain*, or if they are *possible* given the partial information available. In our framework, the most natural facts of interest are usually prefixes of trees. Given an incomplete tree $\mathbf{T}$ and a data tree T over Σ, we say that T is a *certain* prefix if every tree in $rep(\mathbf{T})$ has T as a prefix (up to node identifiers), and it is a *possible* prefix is some tree in $rep(\mathbf{T})$ has T as a prefix. We can show the following, using standard techniques for DTDs.

Theorem 2.3 Given an incomplete tree $\mathbf{T}$ and a data tree T over Σ, it can be checked in PTIME whether T is a certain prefix or whether T is a possible prefix.

3 Acquiring and Using Incomplete Information

We present here the main results of the paper, showing that our framework can be efficiently used to deal with incomplete information in the scenario we described. We first deal with acquiring partial information, and discuss the possible exponential blowup of the representation. We then consider the problem of answering queries when our knowledge consists of an incomplete tree, i.e., the classical problem of querying incomplete databases applied to our model. Finally, we consider the issue of "completing" our knowledge in order to fully answer a given query.

3.1 Acquiring incomplete information

In our basic scenario, information about the Web is acquired gradually using answers to queries. We next show how this can be done in the framework we developed. For simplicity, we assume that the input is a single document described by a tree type. The case of multiple sources can be easily reduced to this case by virtually merging the sources into a single document.

Consider an input tree T. Initially, all we know about T is its tree type, say τ. As consecutive ps-queries are asked, each answer refines our partial information about T, which we describe using an incomplete tree. At each stage of the process, we have an incomplete tree $\mathbf{T}$, a ps-query q and the answer $A = q(T)$. Using this, we refine our incomplete information by computing $\mathbf{T}'$ which describes *precisely* the trees in $rep(\mathbf{T})$ and compatible with the answer A to query q. The refinement algorithm is called $Refine(T, q, A)$ and is outlined in the appendix. We illustrate the idea using the catalog example.

Example 3.1 The incomplete tree after Query 1 contains a data tree which is output of query 1, and an incomplete tree which describe the missing products. A product is not returned by Query 1 if (i) it is not a electronic product or (ii) its price is greater than 200. This is done by creating two new labels *product1* and *product2* with obvious conditions attached to it and which are specialization of *product*.

The construction of the new incomplete tree 2 requires us to represent several kinds of products:

Products returned by both Query 1 and 2: these are the cheap cameras with pictures. Suppose the node ids indicate that the products with name *Canon* in the answers to Queries 1 and 2 are the same. The information returned for this node by the two queries can be merged. Note that persistent node id assumption is critical here.

Products in Query 2 and not Query 1: The typing information is used to register the fact that the (unknown) price of these products must be more than 200. This is the case of the Olympus camera, which is of type *product2a*.

Products in Query 1 and not Query 2: This is the case of the Nikon camera. A product returned by Query 1 is in this category either because it is not a camera or because it is a camera and has no picture. In the case of Nikon, we already know it is a camera, so we can infer that it has no picture, i.e., its type is *product3*.

Missing products: a product may be returned neither by Query 1 nor by Query 2 because it is not an electronic product, because it is expensive but not a camera, or because it is an expensive camera without pictures. This yields the three categories of missing nodes (colored grey).

Note that *product2b* and *product2c* are refinements of *product2*.

Similarly to the example, the refinement algorithm computing $\mathbf{T}'$ from $\mathbf{T}$, q, and A requires a case analysis to consider the nodes present in both $\mathbf{T}$ and A, those in $\mathbf{T}$ and not in A, and those in A but not in $\mathbf{T}$. It also requires refining the portion of $\mathbf{T}$ describing missing information with the added knowledge that the nodes still missing were not in the answer to q. This uses several nice properties of our representation system, such as the fact that conditional tree types are closed under difference with answers to ps-queries, and that the representation of the difference can be computed efficiently. We can show the following:

Theorem 3.2 Given an incomplete tree $\mathbf{T}$ and a ps-query q with answer A, Algorithm *Refine* computes in polynomial time an incomplete tree $\mathbf{T}'$ such that $rep(\mathbf{T}') = rep(\mathbf{T}) \cap q^{-1}(A)$.

Algorithm *Refine* can be used repeatedly to incrementally refine an initial tree type given successive query/answer pairs $(q_1, A_1) \ldots, (q_n, A_n)$. Although each incremental step can be done in PTIME, the size of the incomplete tree may become exponential in the overall sequence of query/answer pairs, as illustrated in Example 3.4 below.

All of our algorithms on incomplete trees have PTIME complexity. However, since the incomplete trees themselves can become exponential with respect to the sequence of query/answer pairs from which they are constructed, the algorithms we developed have, in the worst case, exponential complexity with respect to the overall sequence. One might legitimately wonder if this is due to the particular representation system we have chosen. The answer turns out to be negative: we can prove lower-bounds independent of the representation system. We illustrate this with the possible and certain prefix question, shown in Theorem 2.3 to be in PTIME with respect to the incomplete tree.

Theorem 3.3 Let τ be a tree type over a fixed alphabet Σ, and $\langle q_i, A_i \rangle$, a sequence of ps-query/answer pairs, $1 \leq i \leq n$. Let T be a data tree over Σ:

(i) it is NP-complete to determine whether T is the prefix of some $T' \in rep(\tau)$ such that $A_i = q_i(T')$, $1 \leq i \leq n$.

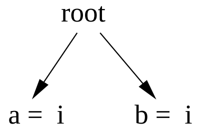
(ii) it is CO-NP-complete to determine whether T is a prefix of every tree T' as in (i), up to node identifiers.

Proof: Reduction from 3-SAT, see Appendix. □

3.2 Avoiding the exponential blowup

The exponential blowup of incomplete trees is illustrated by the following example.

Example 3.4 Consider a tree type $root \rightarrow a\ b$ and the queries q_i, $1 \leq i \leq n$, of the form:



whose answers are empty. The incomplete tree constructed for these queries by Algorithm *Refine* yields a disjunction of 2^n multiplicity statements.

We consider two ways of avoiding the exponential blowup of incomplete trees:

1. By allowing conjunctions of disjunctions of types in the definition of the incomplete tree,

2. By restricting the initial tree type and the ps-queries.

We also propose two heuristics for dealing with incomplete trees that become too large, regardless of the complexity-theoretic bound. Due to space limitations, we briefly sketch these approaches in Section E of the appendix.

3.3 Querying incomplete trees

We next show how incomplete trees can be used to answer queries. We look at two distinct issues. The first is the classical problem of answering a query using only the information provided by the incomplete tree. The answer is itself an incomplete tree, providing a description of the possible answers. This first issue is addressed in the present section. The second issue, addressed in next section, is using incomplete trees as a guide for a mediator which must decide what new queries should be asked on the source document in order to provide a complete answer to a user query.

Before considering such questions, we note an important technical property of the incomplete trees produced by Algorithm *Refine*. We call an incomplete tree *reachable* if it is constructed by Algorithm *Refine* from some initial tree type and some sequence of query/answer pairs. By inspection of the algorithm, it can be shown that reachable trees have a special form which, intuitively, ensures that a multiplicity statement of the incomplete tree involving several specialized versions of some symbol a can be unambiguously parsed: each specialization of a in the statement can be assigned to exactly one data node of T_d, except for possibly one which describes missing data. The impact of this property on the complexity of various manipulations is significant. For instance, without this property, computing the complement and intersection of conditional tree types is generally exponential, whereas the analogous manipulations of reachable incomplete trees can be done in polynomial time.

We now consider the first of the questions mentioned above. Suppose our knowledge of the world consists in an incomplete tree $\mathbf{T}$, obtained by Algorithm *Refine*. This means that the possible data trees are in $rep(\mathbf{T})$. Suppose we wish to answer a ps-query q. The possible answers for q are the data trees in $q(rep(\mathbf{T}))$. Incomplete trees form a *strong representation* system for ps-queries if the set q(rep($\mathbf{T}$)) can be described using an incomplete tree for arbitrary q and $\mathbf{T}$. Indeed, this is the case:

Theorem 3.5 Given an incomplete tree $\mathbf{T}$ and a ps-query q, one can effectively construct an incomplete tree denoted $q(\mathbf{T})$ such that

$$rep(q(\mathbf{T})) = \{q(T) \mid T \in rep(\mathbf{T})\} = q(rep(\mathbf{T})).$$

Furthermore, if $\mathbf{T}$ is reachable, then $q(\mathbf{T})$ can be constructed in PTIME.

As a very useful consequence, we can decide if a ps-query q can be fully answered using the information provided by an incomplete tree $\mathbf{T}$.

Corollary 3.6 Let $\mathbf{T}$ be an incomplete tree with data tree T_d, and q a ps-query. It is decidable in PTIME whether q can be fully answered using $\mathbf{T}$, i.e. whether for every $T \in rep(\mathbf{T})$, $q(T) = q(T_d)$.

Proof: This can be reduced to tests of emptiness of several conditional tree types, which can be done in PTIME (see Lemma 2.2). □

Remark 3.7 As an important side effect, Corollary 3.6 in combination with Theorem 3.2 provide a way to check if a ps-query q can be answered using the views provided by a sequence of ps-query/answer pairs. The problem of answering queries using views has been studied recently in other contexts (see related work). □

There are several important variants of the query answering problem with incomplete information, such as deciding if certain facts are *certain* or *possible* in the answers to a given query. The following is an immediate consequence of Theorems 2.3 and 3.5.

Theorem 3.8 Given a reachable incomplete tree $\mathbf{T}$, a ps-query q, and a data tree T over Σ, it can be checked in PTIME whether T is a certain prefix or whether T is a possible prefix of $q(\mathbf{T})$.

3.4 Guiding mediators

We consider here the following problem: suppose we have partial information about the input document(s) specified as an incomplete tree, and the user poses a query against the virtual input document. If we are lucky, we may be able to provide the complete answer to the query using the information available. Otherwise, additional queries may have to be generated against the input document to obtain the information needed to fully answer the query. The incomplete tree can be used as a guide to generate such queries. We assume that the generated queries further explore the input document starting from the nodes already available. We refer to such queries as *local*. In order to generate local queries intelligently, we must determine which sources possibly (or certainly) contain information relevant to the query.

The following states that these questions can be effectively answered:

Theorem 3.9 Given a reachable incomplete tree $\mathbf{T}$ and a ps-query q the following can be checked in PTIME:

(possible non-emptiness) $q(T) \neq \emptyset$ for *some* tree $T \in rep(\mathbf{T})$; and,

(certain non-emptiness) $q(T) \neq \emptyset$ for *every* tree $T \in rep(\mathbf{T})$.

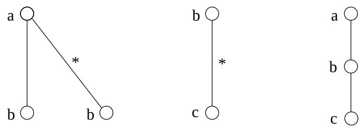
The proof is similar to that of Theorem 2.3.

We now consider in more detail the generation of local queries. We show that we can always efficiently compute a set of local queries that collect the additional information allowing to answer a given ps-query. More formally, let $\mathbf{T}$ be an incomplete tree with data tree T_d and q a ps-query. A *local* ps-query is an expression of the form $p@n$ where p is a ps-query and n is a node in T_d. The query returns the answer to p on the full subtree of T_d rooted at n. Consider a set L of local queries against $\mathbf{T}$. We say that L *completes* $\mathbf{T}$ *relative to* q if for each $T \in rep(\mathbf{T})$, $q(T)$ equals $q(T')$ where T' is obtained by extending each node n of T_d for which $p@n \in L$, with $p@n(T)$.

Obviously, the query q itself posed at the root is always a trivial completion of $\mathbf{T}$ relative to q. The point in using local queries is to avoid doing the work already done by previous queries. Therefore, we would like the completion L to have the property that no nodes already existing in $\mathbf{T}$ are retrieved again by queries in L, and that no new node is retrieved by distinct queries in L. Finally, L should not contain queries which always return empty answers on the possible input trees. If L has these properties, we call it *non-redundant*.

In general, it is not possible to find a non-redundant L for an arbitrary incomplete tree $\mathbf{T}$ and ps-query q, as illustrated next.

Example 3.10 Consider the following incomplete tree $\mathbf{T}$ and query q:



Incomplete tree Query

Let L be some completion of $\mathbf{T}$ relative to q. Clearly, L must contain query q itself applied to the root, which retrieves the already existing b-node in $\mathbf{T}$.

Luckily, although $\mathbf{T}$ in Example 3.10 is a valid incomplete tree, it is not a reachable one. For reachable incomplete trees, it turns out that non-redundant completions can be found, as stated next.

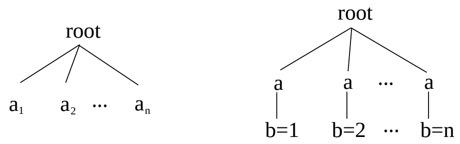
Theorem 3.11 Let $\mathbf{T}$ be a reachable incomplete tree, and q a ps-query. One can construct in PTIME a set of local queries L which forms a non-redundant completion of $\mathbf{T}$ relative to q.

Although non-redundancy of completions is an appealing property, enforcing it clearly comes at a cost. In practice, other parameters are likely to also be taken into account in order to generate efficient completions.

4 Extensions

Our framework for XML documents with incomplete information relies on many limitations and assumptions, such as the availability of persistent node ids, the lack of order, and a very simple query language. In this section we discuss several extensions to our framework, and their impact on handling incomplete information. A comprehensive study of possible extensions is beyond the scope of this paper. Instead, we illustrate the kinds of difficulties that various extensions may introduce. Many of the definitions in this section are informal. We begin by discussing several extensions to ps-queries, and their impact on handling incomplete information.

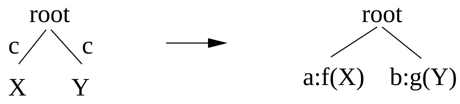
Branching Recall that ps-query tree patterns allow just one child with a given label for each node in the pattern. For instance, this disallows a query whose pattern looks simultaneously for a product which has a picture and another which is a camera. Branching allows multiple children with the same label. Incomplete trees remain a strong representation system for ps-queries extended with branching and can be maintained incrementally in PTIME. However, if $\mathbf{T}$ is a reachable incomplete tree and q a ps-query with branching, $q(\mathbf{T})$ may now be exponential with respect to $\mathbf{T}$. For example, if the data tree of $\mathbf{T}$ is (a), where $a_1 \ldots a_n$ are specializations of a, and q is the ps-query (b) with branching,



(a) (b)

then the incomplete tree for $q(\mathbf{T})$ has to describe $n!$ possibilities of assigning the n values of b to $a_1, \ldots a_n$.

Branching and constructed answers Queries with constructed answers consist of a *body* and a *head.* As for ps-queries, the body is a tree pattern. However, the nodes in the pattern are labeled by variables. For each input, the body defines a set of bindings of the variables to input nodes. The head of the query specifies how to construct an answer data tree from the bindings. This is done in the spirit of XML-QL, using Skolem functions. Incomplete trees are no longer a strong representation system for ps-queries with branching and constructed answers. For example, the query:



body head

produces answers with equal numbers of a's and b's (one a for each binding of X and one b for each binding of Y), which cannot be precisely described by incomplete trees. In fact, the existence of a strong representation system for such queries remains open.

Branching and optional subtrees Queries with optional subtrees allow labeling some subtrees by "?". The semantics is that a valuation is now a partial mapping, not required to be defined on the nodes of the optional subtrees. For example, this allows to request cameras and display their pictures if they exist. However, a camera is in the answer even if it has no picture. The combination of branching and optional subtrees yields an exponential blowup in complexity for several questions we considered. We illustrate this using a variant of the *certain prefix* question. Note that, by Theorems 3.2, 3.5, and 2.3, it can be checked in PTIME whether a tree T is a certain or possible prefix for the answers of a ps-query q' on trees compatible with a given an input tree type τ and a *single* ps-query/answer pair $\langle q, A\rangle$. For ps-queries extended with branching and optional subtrees, we can show the following:

Theorem 4.1 Given a tree T, a tree type τ, a query/answer pair $\langle q, A\rangle$, and a query q', where q and q' are ps-query with branching and optional subtrees, it is CO-NP-complete whether T is a certain prefix for $q'[rep(\tau) \cap q^{-1}(A)]$.

Proof: By reduction of validity of DNF formulas with 3 variables per disjunct, see Appendix. □

Note that this complexity lower bound is independent of a particular representation system.

Extended k-pebble transducers It turns out that our framework can be extended to *ordered* trees and very powerful restructuring queries, as long as data joins are not allowed. We illustrate this using the k-pebble tree transducers introduced in [MSV00] to model a wide range of XML transformation languages. The original k-pebble transducers work on trees without data values. However, they can be easily extended with selection conditions, which we refer to as *extended k-pebble transducers.* Their acceptor analog is called *extended k-pebble automaton*, and languages of ordered data trees they accept are called *extended regular tree languages.* Informal definitions are provided for convenience in the appendix.

Input tree types can be defined as extended regular tree languages. The natural representation system for such queries is the extended k-pebble automaton. For such an automaton τ, $rep(\tau)$ is the data tree language accepted by τ. Extended k-pebble automata provide a concise representation system that can be maintained efficiently and stays polynomial in the input type and the entire sequence of query/answer pairs. Indeed, the following can be shown by extending the techniques from [MSV00]:

Theorem 4.2 Let τ be an input type specified by an extended k-pebble automaton, and $\langle q_1, A_1\rangle, \ldots, \langle q_n, A_n\rangle$ a sequence of query/answer pairs where each q_i is an extended k-pebble transducer and $A_i \in q_i(T)$, $i \in [1, n]$. There exists an extended k-pebble automaton τ', computable in PTIME from τ and the query/answer sequence, such that

$$rep(\tau') = rep(\tau) \cap q_1^{-1}(A_1) \cap \cdots \cap q_n^{-1}(A_n).$$

Despite the fact that k-pebble automata provide an efficient representation system for a very broad class of restructuring queries, they have several drawbacks. First, the intuitively appealing representation of incomplete information provided by incomplete trees is lost. Second, k-pebble automata are not a *strong* representation system. Indeed, as discussed in [MSV00], $q(rep(\tau))$ is not necessarily a regular tree language for an input type τ and k-pebble transducer q (this problem already occurred in the simpler setting of branching ps-queries with very simple constructed answers, see above). Finally, the basic manipulations needed to handle incomplete information have very high complexity, as indicated by the following lower bound result:

Theorem 4.3 It is non-elementary to determine, for an extended k-pebble automaton τ, whether $rep(\tau) = \emptyset$.

The proof, due to Thomas Schwentick [Sch00], uses the fact that testing emptiness of star-free generalized regular expressions[1] is non-elementary [Sto74]. Recall that emptiness of conditional trees types can be tested in PTIME by Lemma 2.2, and this is a basic step in many of our manipulations.

The next three extensions involve join on data values. This turns out to be an extremely powerful feature that leads to a dramatic increase in the difficulty of handling incomplete information. Indeed, many of the key questions now become undecidable.

Branching, join on data values, and negation Negation consists of labeling some subtrees by "$\neg$". With this semantics, a valuation must match positive subtrees and there must be no extension of the valuation matching the negative subtrees. This extends the negation on leaves already allowed in ps-queries. Join on data values allows comparing the data values of different nodes in the pattern of the query (using $=$, $\neq$). This combination of features leads to undecidability of several questions. For example, given an input tree type and a sequence of query/answer pairs (with branching, data value joins, and negation) it is undecidable whether a new query always has empty answer.

Theorem 4.4 It is undecidable, given a (non-recursive) input tree type τ and a sequence $\langle q_1, A_1\rangle, \cdots, \langle q_n, A_n\rangle$ of query/answer pairs and a query q, where q_i and q are ps-queries extended with branching, data value (in)equality, and negation, whether $q(T) = \emptyset$ for all

$$T \in rep(\tau) \cap q_1^{-1}(A_1) \cap \cdots \cap q_n^{-1}(A_n).$$

Proof: Reduction from the undecidability of implication for inclusion and functional dependencies, see Appendix. □

As an easy variation, it can be shown that it is undecidable whether a query is always non-empty for trees satisfying the input tree type and compatible with the query/answer pairs. It is also undecidable whether a given tree is a possible (certain) prefix for such trees.

Note that these results are independent of any representation system. In fact, they imply that there cannot exist an effective representation system for such queries, for which possible emptiness (or the *possible prefix* question) is decidable.

[1] These are expressions using alphabet symbols, union, concatenation, and complement.

Branching, join on data values, optional subtrees, and construction By a reduction similar to the proof of Theorem 4.4, we can show the following:

Theorem 4.5 It is undecidable, given a data tree T, a non-recursive input tree type τ, a sequence $\langle q_1, A_1\rangle, \ldots, \langle q_n, A_n\rangle$ of query/answer pairs and a query q, where q_i and q are ps-queries extended with branching, data value (in)equality, optional subtrees, and constructed answers, whether T is a possible prefix for answers to query q on trees in $rep(\tau) \cap q_1^{-1}(A_1) \cap \ldots \cap q_n^{-1}(A_n)$.

As an aside, Theorems 4.4 and 4.5 highlight an interesting trade-off between negation, and optional subtrees together with constructed answers.

Recursive path expressions and join on data values This allows specifying in a query pattern that a node is reachable from its parent in the pattern tree by a path whose labels spell a word in some regular language. Extending ps-queries with recursive path expressions and tests of (in)equality on data values leads once again to undecidability of various key questions. Indeed, we can show:

Theorem 4.6 It is undecidable, given an input tree type τ, a sequence $\langle q_1, A_1\rangle, \ldots, \langle q_n, A_n\rangle$ of query/answer pairs and a query q, where q_i and q are ps-queries extended with recursive path expressions and (in)equality tests on data values, whether $q(T) = \emptyset$ for some

$$T \in rep(\tau) \cap q_1^{-1}(A_1) \cap \ldots \cap q_n^{-1}(A_n).$$

Proof: By reduction of the emptiness of intersection of two context-free languages, see Appendix. $\Box$

In particular, the above shows that there can be no effective strong representation system for this class of queries.

The persistent node id assumption and the issue of order are discussed informally in Section F.

5 Conclusion

The main contribution of this paper is a simple framework for acquiring, maintaining, and querying XML documents with incomplete information. The framework provides a model for XML documents and DTDs, a simple XML query language, and a representation system for XML with incomplete information. We show that the incomplete information acquired by consecutive queries and answers can be efficiently represented and incrementally refined using our representation system. Queries are handled efficiently and flexibly. They are answered as best possible using the available information, either completely, or by providing an incomplete answer using our representation system. Alternatively, full answers can be provided by completing the partial information using additional queries to the sources, guaranteed to be non-redundant.

Our framework is limited in many ways. For example, we assume that sources provide persistent node ids. Order in documents and DTDs is ignored, and is not used by queries. The query language is very simple, and does not use recursive path expressions and data joins. In order to trace the boundary of tractability, we considered several extensions to our framework and showed that they have significant impact on handling incomplete information, ranging from cosmetic to high complexity or undecidability. This justifies the particular cocktail of features making up our framework, and suggests that it provides a practically appealing solution to handling incomplete information in XML.

References

[AD98] S. Abiteboul and O. Duschka. Answering queries using materialized views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1998.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading-Massachusetts, 1995.

[AKG91] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78:159–187, 1991.

[AKMD98] A.Bruggemann-Klein, M.Murata, and D.Wood. Regular tree languages over non-ranked alphabets. Unpublished manuscript, 1998.

[BM99] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proc. Int. Conf. on Database Theory (ICDT)*, 1999.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediator needs data conversion. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1998.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. IEEE Intl. Conf. on Data Engineering*, 1995.

[Cod75] T. Codd. Understanding relations (installment #7). In *FDT Bull. of ACM Sigmod 7*, pages 23–28, 1975.

[Cos83] S. S. Cosmadakis. The complexity of evaluating relational queries. *Inf. and Control*, 58:101–112, 1983.

[GMLY98] H. Garcia-Molina, W.J. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1998.

[HLY80] P. Honeyman, R.E. Ladner, and M. Yannakakis. Testing the universal instance assumption. *Inf. Proc. Letters*, 10(1):14–19, 1980.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[IL84] Tomasz Imieliński and Jr. Lipski, Witold. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[KNS99] Y. Kanza, W. Nutt, and Y. Sagiv. Queries with incomplete answers over semistructured data. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1999.

[LMSS95] A. Levy, A.O. Mendelzon, D. Srivastava, and Y. Sagiv. Answering queries using views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1995.

[LZW+97] W.J. Labio, Y. Zhuge, J. L. Wiener, H. Gupta, H. Garcia-Molina, and J. Widom. The WHIPS prototype for data warehouse creation and maintenance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1997.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for xml transformers. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 2000.

[MSY81] D. Maier, Y. Sagiv, and M. Yannakakis. On the complexity of testing implications of functional and join dependencies. *J. ACM*, 28(4):680–695, 1981.

[PV00] Y. Papakonstantinou and V. Vianu. Dtd inference for views of xml data. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 2000.

[Rei86] R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *J. ACM*, 33(2):349–370, 1986.

[RSU95] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1995.

[Sch00] T. Schwentik. Personal communication, 2000.

[Sto74] L. Stockmeier. *The complexity of decision problems in automata theory and logic*. PhD thesis, Massachusetts Institute of Technology, 1974. Report MAC TR-133, Project MAC.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.

[Var86] M. Y. Vardi. On the integrity of databases with incomplete information. In *Proc. ACM Symp. on Principles of Database Systems*, pages 252–266, 1986.

[Zan84] C. Zaniolo. Database relations with null values. *Journal of Computer and System Sciences*, 28(1):142–166, 1984.

Appendix

A More Examples

Example A.1 (**More catalog queries**) We illustrate the use of incomplete trees to answer queries. Continuing with the catalog example in Figure 1, suppose the following query arrives following Query 1 and Query 2:

- Query 3: find the name, price and pictures of all cameras costing less than \$100 and having at least one picture.

Clearly, we can answer this query fully using just the information available locally. More interestingly, suppose we ask the query:

- Query 4: list all cameras.

While we are not able to provide the complete answer, we can do the following:

1. provide the complete list of cameras that are less than \$200 or have a picture;
2. tell the user that there may be more cameras (that are expensive and have no pictures).

Thus, we can provide an incomplete answer to the query given the knowledge available, without accessing the data source for further information. If accessing the source is possible and desired, we can complement our result by addressing the following query:

- Query 5: find the name and price of cameras that cost at least \$200 and have no picture.

Thus, we need to fetch only the missing data.

Example A.2 (**Publications**) To see another example, consider a simplified Web site of a research group. The first page for each member of the group is her home-page. Suppose homepages provide the headers of documents and pointers to the actual document (see Figure 2). Suppose, as depicted in the figure, that we have only loaded the headers of publications since 1995 and only loaded the documents themselves for publications since 1998. We can abstract this as a single document that can be queried. URLs can serve here as ids for some nodes. Consider the two queries:

1. list the titles of papers written by Doe in 1997.
2. get the papers written by Doe in 1999.

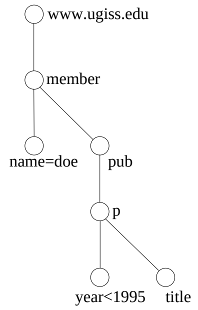
We can answer these queries using the available information. Now consider the query

3. list the titles and years of all papers written by Doe.

We can partially answer this query by:

1. a list of titles (for papers since 1995)
2. an indication to the user that there may be more papers written before 1995.

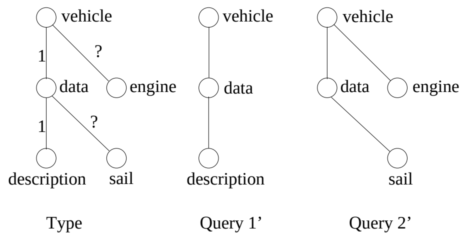
This may suffice. Otherwise, we can complete the missing knowledge with the following query:



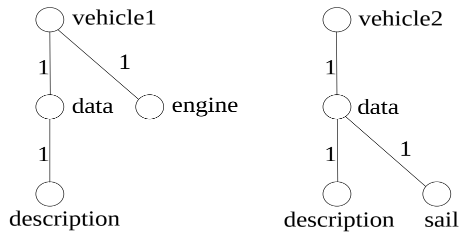
If we later need all of Doe's papers, we can further request only the missing documents, i.e., those written before 1997.

Example A.3 (**Vehicle example**) In this third example, we illustrate a subtlety that complicates somewhat the framework described so far, although it may be rare in practice. In some cases, the missing information at different levels in the tree may not be independent of each other. Thus, the missing information cannot always be described locally at each level.

Consider a document of the following type (we are not suggesting this is good design!):



Suppose we ask Query $1'$ and obtain as answer a vehicle with a data and a description nodes. Next we ask Query $2'$ and the answer is empty. Now we know that the vehicle returned by Query $1'$ cannot have both an engine and a sail. Thus the type of the root is *vehicle1* or *vehicle2* where these are defined by:



In this case, we cannot describe what is missing at the *vehicle* node independently from what is missing at the *data* node. To get around the problem, we have to introduce a disjunction of types.

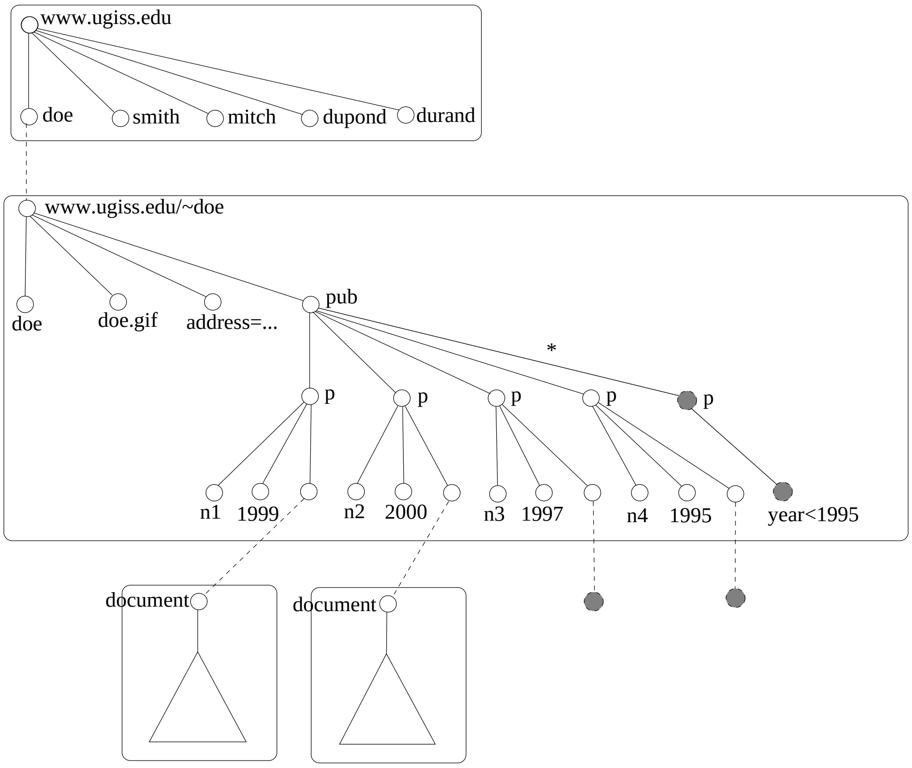


Figure 2: Publications example

Example A.4 We use the vehicle example to illustrate the use of disjunctive types in incomplete trees. Recall that Query $1'$ returns a prefix tree consisting of a vehicle node v, a data node d, and a description node d'. The alphabet Σ is {vehicle, data, engine, description, sail}. The incomplete tree summarizing the information that takes into account Query $1'$ is as follows. The specialized alphabet is $\Sigma' = $ {vehicle1, vehicle2, data1, data2, engine, description, sail}. The Σ'-specialized prefix tree is the tree returned by Query $1'$, together with the labeling λ' defined by:

$$\lambda'(v) = \{\text{vehicle1, vehicle2}\}$$
$$\lambda'(d) = \{\text{data1, data2 }\}$$
$$\lambda'(d') = \{\text{description}\}$$

The conditional tree type of the incomplete tree is $\langle\tau, \text{cond}, \sigma\rangle$ where *cond* is *true* for every node, σ maps Σ' to Σ in the obvious way, and τ is defined by

$$\begin{array}{lcl} vehicle1 & \to & \textit{data1 engine} \\ vehicle2 & \to & \textit{data1 description} \\ data2 & \to & \textit{description sail} \end{array}$$

B Algorithm Refine

We informally outline in this section the main steps of Algorithm *Refine*.

Its input is an incomplete tree $\mathbf{T}$ and a ps-query q with answer A, where:

- $\mathbf{T}$ consists of a *conditional tree type* $(\tau, \text{cond}, \sigma)$ over Σ, with specialized alphabet Σ' and a Σ-*specialized data tree* (t, λ, μ).
- $q = \langle t_q, \lambda_q, \text{cond}_q\rangle$

The output is a new incomplete tree $\mathbf{T}'$ such that

$$rep(\mathbf{T}') = rep(\mathbf{T}) \cap q^{-1}(A).$$

1. First compute the conditional tree type of the *negation* of q. This is the conditional tree type $\tau_{q^{-1}}$ corresponding to trees which return an empty answers to q. This can been done as follows. For each node a of the query tree t_q, create types t_a, $\bar{t}_a$, and $\hat{t}_a$. Let l_a be the label of a ($\lambda_q(a)$). Each of the new types are a specialization of l_a ($\sigma'(t_a) = \sigma'(\bar{t}_a) = \sigma'(\hat{t}_a)$). Intuitively t_a will accept any subtree starting in a assuming it is not involved in the emptiness the query; $\bar{t}_a$ accepts any subtree starting in a not verifying the condition $\text{cond}_q(a)$; and $\hat{t}_a$ accepts subtrees starting in a satisfying the condition $\text{cond}_q(a)$ but returning an empty answer to the query q : it propagates downward the cause for emptiness.

   Set $\text{cond}'(t_a) = true$,$\text{cond}'(\bar{t}_a) = \neg\text{cond}_q(a)$, $\text{cond}'(\hat{t}_a) = \text{cond}_q(a)$.

   Now create a new tree type $\tau_{q^{-1}}$ by processing the query from the root to the leaves inductively.

The new root has type $\hat{t}_r \vee \bar{t}_r$ where r is the root of t_q. For each node a of t_q, let $a_1, \cdots, a_n$ be its children in t_q (by definition of a ps-query no two children can have the same label, thus n is bounded by $|\Sigma|$). Add the following rules to $\tau_{q^{-1}}$.

- $t_a \to t^\star_{a_1} \cdots t^\star_{a_n}$, accept everything.
- $\bar{t}_a \to t^\star_{a_1} \cdots t^\star_{a_n}$, accept everything below a because there the condition of q is not satisfied by a.
- $\hat{t}_a \to \bigvee_i t^\star_{a_1} \cdots \hat{t}^\star_{a_i} \bar{t}^\star_{a_i} \cdots t^\star_{a_n}$, one of the children must not satisfy a condition on q.

This requires $O(|q|.|\Sigma|)$ iterations.

Note that nodes which are leaves have to be treated differently : indeed they can't propagate downward an eventual failure and thus the type $\hat{t}_l$ where l is a leave is not created. It is easy to modify the above step accordingly.

The next step consists of computing the intersection of the above tree type $\tau_{q^{-1}}$ with the input tree type τ. This is straightforward and can be done in polynomial time. Note that this yields a tree type containing conjunctions of disjunctions of multiplicity atoms.

This produces the conditional tree type of the missing information of $\mathbf{T}'$.

2. The new data tree is obtained as follows. Create a new type for each node of A. Compute the *join* between A and t (the existence of node ids is crucial here). For data in A and t specialize it computing the intersection of both types. For data in t but not in A, the type is specialized using the conditional tree type **T'** previously computed.

C Extended k-pebble transducers

We provide for convenience an informal definition of the extended k-pebble transducer and automaton. The definition differs from the k-pebble transducer and automaton of [MSV00] only in the tests on data values.

The extended k-pebble transducer uses up to k pebbles to mark certain nodes in the tree. Transitions are determined by the current node symbol, the current state, and by the existence/absence of the various pebbles on the node. The pebbles are ordered and numbered $1, 2, \ldots, k$. The machine can place pebbles on the root, move them around, and remove them. In order to limit the power of the transducer (so that typechecking can be performed) the use of pebbles is restricted by a stack discipline: pebbles are placed on the tree in order and removed in reverse order, and only the highest-numbered pebble present on the tree can be moved. More precisely, pebble i, ($i > 1$) can only be placed on the tree if pebbles $1, \ldots, i-1$ are present, and pebble $i-1$ cannot be moved as long as pebble i is still on the tree. The stack discipline is essential to the results.

The transducer works as follows. The computation starts by placing pebble 1 on the root. At each point, pebbles $1, 2, \ldots, i$ are on the tree, for some $i \in \{1, \ldots, k\}$; pebble i is called the *current pebble*, and the node on which it sits is the *current node*. The current pebble serves as the head of the machine. The machine decides which *transition* to make, based on the following information: the current state, the symbol under the current pebble, a condition on the data value of the node under the current pebble (conditions are the same as for ps-queries), and the presence/absence of the other $i-1$ pebbles on the current node. There are two kinds of transitions: *move* and *output* transitions. *Move* transitions are of four kinds:

- *down-left, down-right, up-left, up-right:* the current pebble is moved from the current node in the given direction (one edge only). If a move in the specified direction is not possible, the transition does not apply.
- *stay:* the current pebble does not move, only the state changes.[2]
- *place-new-pebble:* pebble $i+1$ is placed on the root (becoming the current pebble).
- *pick-current-pebble:* the current pebble $i > 1$ is removed from the tree and pebble $i-1$ becomes the current pebble.

After each move transition the machine enters a new state, as specified by the transition.

An *output* transition emits some output node with a symbol in $a' \in \Sigma'$ and does not move the input head. There are two kinds of output transitions. In a *binary* output $a' \in \Sigma'_2$ and the machine spawns two computation branches computing the left and right child respectively. Both branches inherit the positions of all pebbles on the input, and do not communicate; each moves the k pebbles independently of the other. In a *nulary* output $a' \in \Sigma'_0$, the node being output is a leaf and that branch of computation halts.

Looking at the global picture, the machine starts with a single computation branch and no output nodes. After a while it has constructed some top fragment of the output tree, and several computation branches continue to compute the remaining output subtrees. The entire computation terminates when all

[2] Observe that this can be simulated by moves of the above form, and is used here only for convenience.

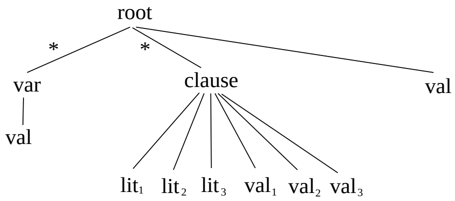
computation branches terminate. Thus, for each input tree the transducer produces a set of possible output trees, one for each possible computation.

The extended k-pebble automaton works similarly to the extended k-pebble transducer, but produces no output: its function is simply to accept or reject its input. As shown in [MSV00], k-pebble automata without conditions on data values accept precisely the regular tree languages. We call a set of data trees accepted by some extended k-pebble automaton an *extended regular tree language*.

The discussion in [MSV00] can be easily adapted to show that all transformations expressible in XML-QL, XSL, LOREL, UnQL, and StruQL, with selection conditions but no joins on data values, can also be expressed by extended k-pebble transducers.

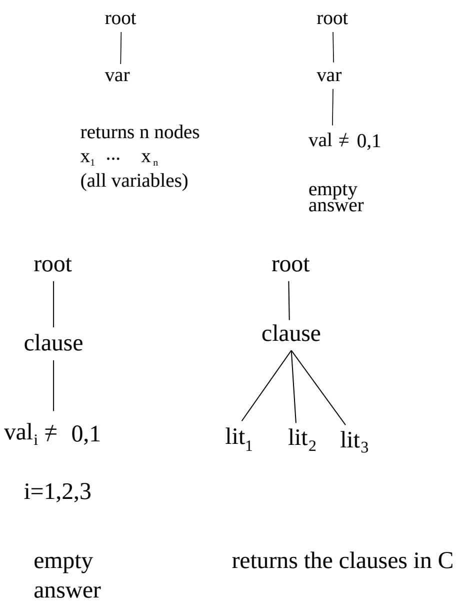
D Some proofs

Proof of Theorem 3.3 The proof of the lower bound for (i) is by reduction of 3-SAT. Let $\mathcal{C}$ be a set of clauses, each with 3 literals. The input tree type is:

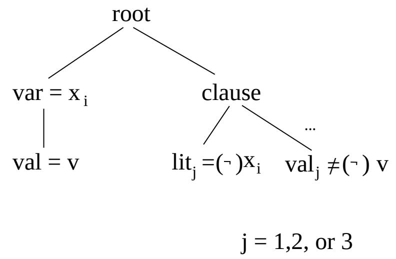


Each clause subtree is supposed to encode a complete clause in $\mathcal{C}$ (i.e. a disjunction of 3 literals). The values of the lit nodes will be literals (x_i or $\neg\ x_i$). The val$_i$ node is supposed to hold a value for literal lit$_i$, $i = 1, 2, 3$.

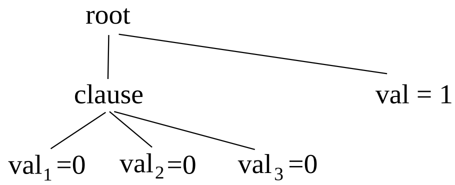
Now ask these queries:



where the answers are as described. So we know there is one value 0 or 1 per variable, that the encoding of the clauses is right, and the value of each literal is 0 or 1. Next, for each variable x_i, value v (0 or 1), literal $(\neg)\ x_i$ and compatible value $(\neg)\ v$ for the literal, ask this query:

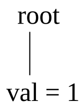


and suppose all answers are empty. This means that all literals in clauses have the right values, given the valuations for the variables. Next, ask the query:



and suppose all answers are empty. This means that *val* cannot equal 1 unless all clauses have at least one literal that is 1. It follows that $\mathcal{C}$ is satisfiable iff it is possible for *val* to equal 1.

Thus checking that the prefix



is possible according to the above sequence of query/answers is equivalent to checking that $\mathcal{C}$ is satisfiable. The result follows.

(ii) is straightforward using negation of (i).

Proof of Theorem 3.5 We outline a PTIME algorithm for computing an incomplete tree $q(\mathbf{T})$ such that

$$rep(q(\mathbf{T})) = \{q(T) \mid T \in rep(\mathbf{T})\} = q(rep(\mathbf{T})).$$

The input to the algorithm is an incomplete tree $\mathbf{T}$ and a ps-query q, where:

- $\mathbf{T}$ is a *conditional tree type* $(\tau, \text{cond}, \sigma)$ over Σ, with specialized alphabet Σ' and a Σ-*specialized data tree* (t, λ, μ).
- q is a tree $\langle t_q, \lambda_q, \text{cond}_q \rangle$

In order to simplify the presentation, we assume that τ has no recursion occurring at a depth $\leq d$, where d is the depth of t_q. This condition can always be enforced by *unfolding* the tree type τ up to the desired depth using specialization. Note that the output tree type is not recursive because q is not recursive.

The algorithm works as follows (sketch):

1. Parse the data tree (t, λ, μ) to collect the data that are relevant for q. This gives a new data tree (t', λ', μ').
2. Parse the tree type τ and remove all parts that are not relevant to q. In the process, replace in multiplicity atoms $\star$ by $+$ and ? by 1 depending on what is left in the data tree and what is needed to answer query q. This yields a new tree type τ'.
3. Parse τ' matching the tree type condition cond with the query condition cond_q.
   - If the conditions are inconsistent remove the corresponding subtree. If this subtree was noted as mandatory in the previous step, output the incomplete tree specifying the empty set.
   - If cond $\Rightarrow$ cond_q, do nothing.
   - Let $\alpha = \text{cond} \cap \text{cond}_q$. Specialize the corresponding node by creating a new type whose condition is α.

   This produces a new conditional tree type $(\tau'', \text{cond'}, \sigma')$ over Σ, with specialized alphabet Σ''.
4. Update λ' according to Σ'' and return $\langle\ (t', \lambda', \mu')\ ,\ (\tau'', \text{cond'}, \sigma')\ \rangle$.

Proof of Theorem 3.11 (sketch)

The input is an incomplete tree $\mathbf{T}$ and a ps-query q.

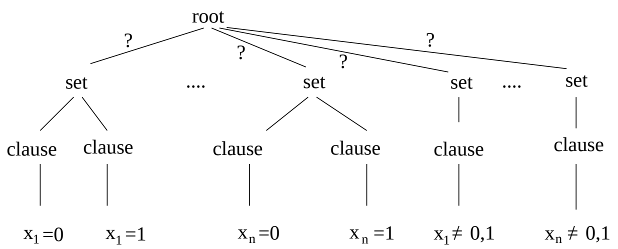
Compute in PTIME $q(\mathbf{T})$ using Theorem 3.5. For each branch b of the incomplete tree type of $q(\mathbf{T})$ starting from the root, check for its emptiness in PTIME using Lemma 2.2. If it is non empty, derive from it a query q_b (this is possible because the tree type of $q(\mathbf{T})$ is not recursive, see Theorem 3.5).

It is easy to check that the set of all q_b form a completion of $\mathbf{T}$ relative to q. It can been shown to be non-redundant as long as $\mathbf{T}$ is reachable.

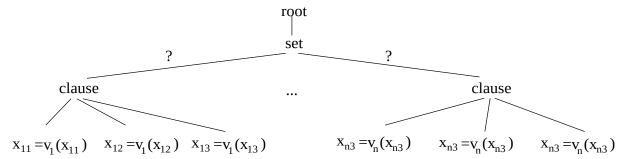
Proof of Theorem 4.1 The proof of CO-NP-hardness is by reduction of validity of DNF formulas with 3 variables per disjunct. Let $\mathcal{D}$ be a set of disjuncts $\{d_1, \cdots, d_k\}$, each with 3 literals, and using variables $x_1, \cdots, x_n$. Consider the input tree type defined by:

$$\begin{array}{lcl} \textit{root} & \rightarrow & \textit{set} \\ \textit{set} & \rightarrow & \textit{clause}^\star \\ \textit{clause} & \rightarrow & x_1 \cdots x_n \end{array}$$

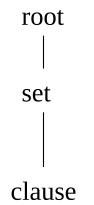
and the ps-query q with branching and optional subtrees:



whose answer A is the tree consisting of the root alone. This indicates that each variable has just one value (0 or 1). For each conjunct $d_i \in \mathcal{D}$, let x_{i1}, x_{i2}, x_{i3} be the three variables occurring in d_i and let v_i be the valuation on these variables defined by $v_i(x_{ij}) = 1$ if x_{ij} occurs in d_i, and $v_i(x_{ij}) = 0$ if $\neg\ x_{ij}$ occurs in d_i, $1 \leq j \leq 3$. Now consider the query q':



Clearly, $\mathcal{D}$ is valid iff the tree:



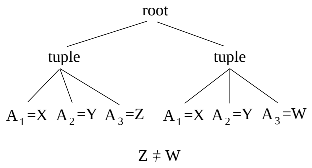
is a certain prefix for the answers to q', i.e. for the set $\{q'(T) \mid T \in rep(\tau)\ \cap\ q^{-1}(A)\}$.

Proof of Theorem 4.4 We use the undecidability of implication for functional and inclusion dependencies. It is known that for some relation R with attributes $A_1, \ldots, A_n$, it is undecidable whether $\Sigma \models \sigma$ where Σ is a finite set of fd's and incd's over R and σ is an fd over R (see [AHV95]). We construct a tree type τ (of fixed depth) that represents relation R:

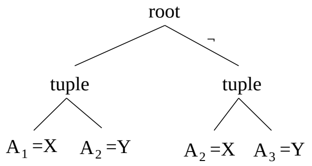
$$\begin{array}{lcl} \textit{root} & \rightarrow & \textit{tuple}^\star \\ \textit{tuple} & \rightarrow & A_1 \ldots A_n \end{array}$$

For each $\varphi \in \Sigma\ \cup\ \{\sigma\}$, we construct a ps-query q_φ with branching, data value (in)equality, and negation, such that for each $T\ \in\ rep(\tau)$, $q_\varphi(T)\ =\ \emptyset$ iff the relation represented by T satisfies φ. Then $\Sigma\ \models \sigma$ iff $q_\sigma(T)\ =\ \emptyset$ for all $T\ \in\ rep(\tau)\ \cap\ \bigcap_{\varphi \in \Sigma} q_\varphi^{-1}(\emptyset)$. This proves Theorem 4.4.

We show by example how to construct the queries q_φ. Suppose φ is a fd, say $A_1 A_2\ \rightarrow\ A_3$. Then q_φ is the query:



Now suppose φ is an inclusion dependency, say $R[A_1 A_2] \subseteq R[A_2 A_3]$. Then q_φ is the query:



Proof of Theorem 4.6 We use a variant of the problem of emptiness of the intersection of two CFGs. Recall that in the classical version, an instance of the problem consists of two CFG's G_1, G_2 over terminal alphabet $\{a, b\}$, and the question is whether $L(G_1) \cap L(G_2) \neq \emptyset$. This is known to be undecidable [HU79]. In the variant we use, the input consists again of two CFG's G_1, G_2, for which it is additionally known that

(†) there exist $w_1 \in L(G_1)$ and $w_2 \in L(G_2)$ such that $|w_1| = |w_2|$.

The question is, again, whether $L(G_1) \cap L(G_2) \neq \emptyset$. We call this the *weak* CFG *intersection emptiness problem.* Note that (†) is decidable, by reduction to testing emptiness of the intersection of the *regular* languages $f(L(G_1))$ and $f(L(G_2))$, where f is the homomorphism $f(a) = f(b) = a$. It easily follows that the weak CFG intersection emptiness problem remains undecidable.

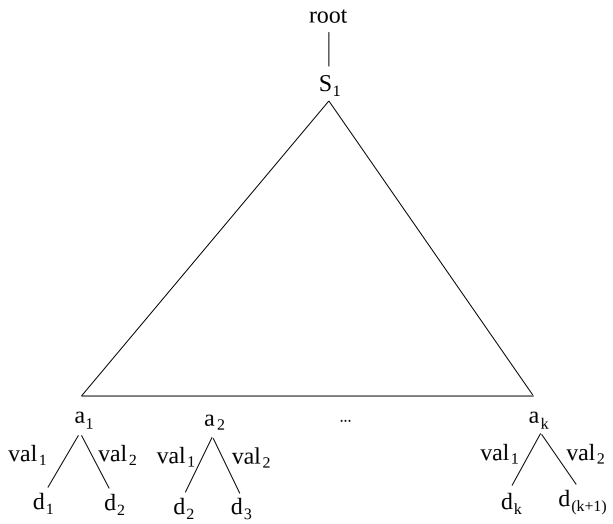
Let G_1, G_2 be CFG's with start symbols S_1 and S_2. Without loss of generality we can assume that their non-terminals are disjoint and that each is in Chomski Normal Form (i.e. productions are of the form $A \to BC$ or $A \to a$ where A, B, C are non-terminals and a is a terminal symbol). In addition, we assume that there are no productions $A \to BC$ and $A \to DB$ (B occurs first in one production of A, and second in another). This can be avoided by using different versions of B for the case when B occurs first and when B occurs second. Iterating this for all non-terminals may result in a quadratic blowup in the number of non-terminals and productions. The purpose of this requirement is to have the names of the children of a non-terminal uniquely determine their order. In particular, this implies that for any non-terminal A there exists a regular expression $r(A)$ such that in every derivation tree of the grammar starting from A, a path matches $r(A)$ iff it leads to the rightmost terminal derived from A. A similar regular expression $l(A)$ exists for the leftmost case. We use the notation $r_i(A), l_i(A)$ for these regular expressions for G_i, $i \in [1, 2]$.

We construct an input tree type τ and queries $q_1, \ldots, q_n$ and q with recursive path expressions and data value (in)equality tests, such that every tree T in $rep(\tau)$ for which $q_i(T) = \emptyset$, $1 \leq i \leq n$, encodes two words $w_1 \in L(G_1)$ and $w_2 \in L(G_2)$ of equal length. In particular, it follows that G_1 and G_2 satisfy (†) and are an instance of the weak intersection emptiness problem. Finally, $q(T) = \emptyset$ iff $w_1 = w_2$. It follows that q is possibly empty on trees in $rep(\tau) \cap q_1^{-1}(\emptyset) \cap \ldots \cap q_n^{-1}(\emptyset)$ iff $L(G_1) \cap L(G_2) \neq \emptyset$, which is undecidable. This proves the theorem.

We outline the construction of the tree type and queries q_i. The tree type τ consists of:

- $root \to S_1\ S_2$
- the productions of G_1 and G_2
- $a \to val_1\ val_2$
- $b \to val_1\ val_2$.

Note that trees in $rep(\tau)$ consist of a derivation tree of G_1 and a derivation tree of G_2. The terminals a, b additionally have children val_1, val_2 attached to them. Their role is to provide a successor relation on data values, inducing an ordering of the terminal symbols to which they are attached. Thus, a word $a_1 \ldots a_k$ derived from S_1 is represented as:



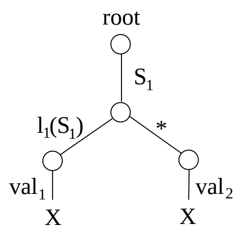
The queries $q_1, \ldots, q_n$ ensure that:

1. the data values for the leaves of the subtree rooted at S_1 form a successor relation, and similarly for S_2;
2. the two sequences of data values at the leaves of the subtrees rooted at S_1 and S_2 are the same.

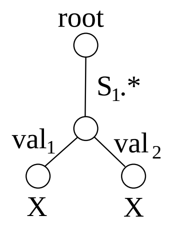
In the graphical representation of queries, we place the labels on edges of the pattern rather than nodes. The labels are regular path expressions, $\star$ being a shortcut for $\Sigma^\star$ (Σ is the set of symbols of the grammars). Variables stand for data values.

The queries whose empty answers ensure (1) above for the subtree rooted at S_1 are as follows:

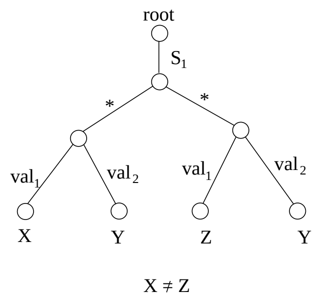
- the leftmost data value is minimal, i.e. never occurs as the value of val_2:



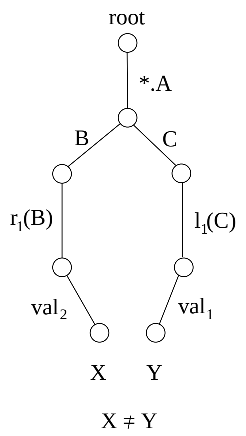
- sibling nodes val_1 and val_2 have different data values (an element's successor is not itself):



- distinct elements have distinct successors:

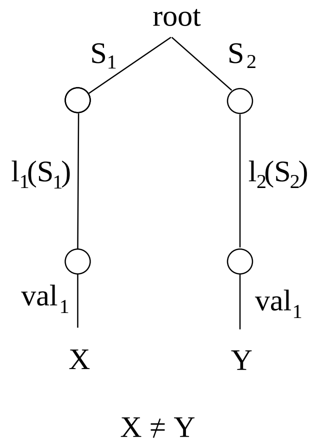


- for adjacent nodes, the val_2 value of the first equals the val_1 value of the second. For each production $A \to BC$ of G_1, we use the query:

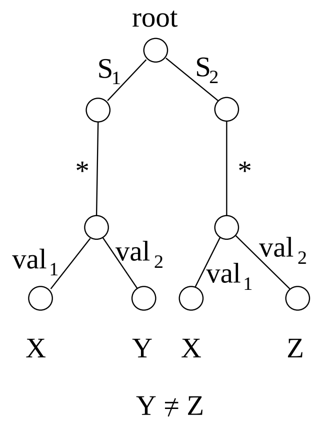


Similar queries ensure (1) for S_2. Item (2) is ensured by emptiness of the following queries:

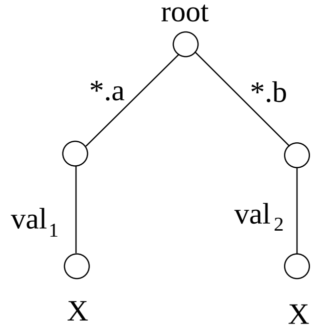
- the minimum data values for S_1 and S_2 are the same:



- the maximum data values in S_1 and S_2 are the same: similar.
- if two nodes in S_1 and S_2 have the same val_1 value then they have the same val_2 value:



Thus, emptiness of the above queries ensures that the words w_1 and w_2 corresponding to S_1 and S_2 are indexed by the same data values in the same order (in particular $|w_1| = |w_2|$). Finally, consider the query q:



Its answer is empty iff w_1 and w_2 have the same terminal symbol corresponding to each data value index, i.e. $w_1 = w_2$.

E More on avoiding the exponential blowup

We first discuss extensions of incomplete trees with conjunction, then restrictions of the initial tree type and ps-queries. We finally discuss the two heuristics for dealing with incomplete trees that become too large, regardless of the complexity-theoretic bound.

Conjunctive incomplete trees It is possible to prevent the exponential blowup of incomplete trees by allowing conjunctions of disjunctions of multiplicity atoms in type specifications, rather than just disjunction. The meaning of a conjunction of disjunctions of multiplicity atoms is that the tree described must be simultaneously valid with respect to *all* types specified by each conjunct. In terms of automata, this is analogous to allowing alternation rather than just nondeterminism in the control. We refer to incomplete trees augmented with conjunction as *conjunctive incomplete trees.* It can be shown that conjunctive incomplete trees can be constructed incrementally by an extension of Algorithm *Refine*, and stay polynomial in the sequence of query/answer pairs and input tree type. The usefulness of conjunction in incomplete trees is illustrated by Example 3.4. The incomplete information provided by the query/answer pairs can be represented concisely using a conjunctions of n disjunctions:

$$\begin{array}{lcl} \mathit{root} & \rightarrow & (a_1 b \ \vee \ ab_1) \wedge \cdots \wedge (a_n b \ \vee \ ab_n) \\ \mathit{cond}(a_i) & = & \neq \ i, \ 1 \leq i \leq n \\ \mathit{cond}(b_i) & = & \neq \ i, \ 1 \leq i \leq n \end{array}$$

where a_i and b_i specialize a and b, respectively. As discussed earlier, without conjunction, Algorithm *Refine* yields a disjunction of 2^n multiplicity statements, corresponding to the DNF form of the n conjuncts.

The price to pay for the conciseness of conjunctive trees is increased complexity of various manipulations. For example, the key problem of checking nonemptiness of a conjunctive incomplete tree becomes NP-complete (see Theorem 3.3), whereas it is polynomial for usual incomplete trees (see Lemma 2.2).

Restricting the tree types and queries A second approach to avoiding the blowup in the size of incomplete trees is to restrict the input tree types and the ps-queries. We exhibit one restriction that we believe is reasonable in many practical situations: the input tree type is non-recursive, and ps-queries are restricted to perform non-trivial data value tests only along one path in the query tree. More precisely, a ps-query $\langle t, \lambda, \text{cond}\rangle$ is *linear testing* if for all pairs of nodes n_i of t for which $\text{cond}(n_i) \neq \mathit{true}$, $1 \leq i \leq 2$, either n_1 is a parent of n_2 or conversely. We can show the following:

Theorem E.1 Let τ be a non-recursive tree type over Σ and $(q_1, A_1), \ldots, (q_n, A_n)$ a sequence of linear-testing ps-queries and their answers on some $T \in \mathit{rep}(\tau)$. The incomplete tree constructed for τ and $(q_1, A_1), \ldots, (q_n, A_n)$ using a simple modification of Algorithm *Refine* has size polynomial in τ and $(q_1, A_1), \ldots, (q_n, A_n)$.

Proof: (sketch)

A straightforward modification of Algorithm Refine produces on input $(q_1, A_1), \ldots, (q_n, A_n)$ a conjunctive incomplete tree **T'** of size polynomial in the sequence $(q_1, A_1), \ldots, (q_n, A_n)$.

Because the queries are linear, the conjunct at each node can be factorized. The transformation of the conjunctive incomplete tree into a regular incomplete tree can now be done in a straightforward way. The process is exponential in the maximal depth of the queries q_i. As the tree type τ is non-recursive, let d be its depth. Each q_i has a depth bounded by d. Thus the algorithm remains polynomial in q_i (exponential in d).

□

Heuristics We sketch two approaches for dealing with cases when the incomplete tree grows too large to be practical. The first consists of asking a small set of additional queries chosen so as to provide precisely the critical information needed to eliminate some of the unknown information and shrink the incomplete tree. The choice of additional queries may be guided by various heuristics, which can be applied whenever the incomplete tree becomes too large. There is a standard choice of additional queries which always keeps the incomplete tree polynomial in size and no larger than the complete input data tree together with its tree type.

Theorem E.2 For each input type τ and sequence of query/answer pairs $\langle q_1, A_1\rangle \ldots \langle q_n, A_n\rangle$ there exist queries $q'_1 \ldots q'_m$, with answers $A'_1, \ldots, A'_m$, such that:

(i) $m \leq O(|q_1| + \cdots + |q_n|)$,

(ii) for each i there exists j such that $|q'_i| \leq |q_j|$,

(iii) the incomplete tree **T** constructed by Algorithm *Refine* for τ and

$$\langle q_1, A_1\rangle \ldots \langle q_n, A_n\rangle, \langle q'_1, A'_1\rangle \ldots \langle q'_m, A'_m\rangle$$

is polynomial in τ and the query/answer pairs, and

(iv) for each $T \in \mathit{rep}(\mathbf{T})$, the size of **T** is $O(|T| + |\tau|)$.

The second approach for dealing with large incomplete trees is to gracefully loose some of the information in order to shrink the incomplete tree. The idea is illustrated by Example 3.4. Intuitively, the incomplete tree becomes large because it enumerates explicitly the restrictions on the pairs of values for a and b, which are tested repeatedly by the queries. This makes it expensive to maintain the information about the connection between the a and b values. One way around this is to "forget" the costly information on the connection between the values, and only retain the ranges of allowed values for a and for b. In general, the expensive combinations of values can be identified by a scoring system maintained dynamically as queries are asked. This approach can be extended to combinations of type specializations, which may involve constraints on both the data values and structure of the allowed trees. We omit further details in this abstract.

F Node ids and Order

To conclude, we informally discuss the persistent node id assumption, and the issue of order.

Node ids A significant assumption in our framework is the availability of persistent node ids. In other words, distinct queries against an XML document return nodes with the same id iff the nodes are identical. The availability of node ids allows us to enrich the information about a given node (e.g. a product) through consecutive queries, as illustrated in the catalog example. Without ids this is no longer possible in general. Furthermore, the representation system for incomplete information would have to be extended in order to keep track of the various possible ways of matching nodes returned by different queries.

Our assumption that node ids are available is generally dependent on sources providing persistent node ids. Even if this is not generally the case, ids are sometimes available as URLs, element names, known keys, etc. Without ids, our approach can still be used but our expectations would have to be lowered if we wish to keep processing cost down.

Order The issue of order has many facets. Indeed, it can be considered at various levels:
(1) the input tree may be ordered, as well as the answers to queries. In this case, one would like to preserve in the answer the order of elements from the input.
(2) the source DTD may describe the order of children at each node type, possibly using a regular expression (as done in full-fledged DTDs).
(3) queries may use ordering in their selection patterns. For example, a query might request all a elements that occur before some b element. To specify such conditions one could use regular expressions, or perhaps weaker partial order conditions.

Our discussion of extended k-pebble transducers in Section 4 shows that some of our framework can be extended in the presence of order, albeit at the cost of high complexity. Intuitively, it is clear that order complicates the handling of incomplete information. As one example, suppose the input is flat and contains a and b elements. Suppose a first query q_1 requests the list of a elements (produced in the order in which they appear in the input) and a second query q_2 asks for the list of b elements. Consider now a third query q_3 that asks for the list of all elements. Can we answer this using the known answers to q_1 and q_2? This depends on the type of the input: if the input is of the form $a{\star}b{*}$ then q_3 can be answered (concatenate the answer to q_1 with the answer to q_2); if the input is of the form $(a+b){*}$ then q_3 cannot be answered using the previous queries, since no information is available on how to interleave the a and b elements. The problem described above is somewhat similar to the issue of persistent ids. Indeed, one way around it is for wrappers of data sources to provide the rank of each element, which allows to merge answers to consecutive queries. In the absence of such information, a representation system has to maintain information about partial orders among the elements. Clearly, the order issue raises many interesting questions that need to be further explored.