Reasoning About Pattern-Based XML Queries

Amélie Gheerbrant[1], Leonid Libkin[1], and Cristina Sirangelo[2]

[1] School of Informatics, University of Edinburgh
[2] LSV, ENS-Cachan INRIA & CNRS

Abstract. We survey results about static analysis of pattern-based queries over XML documents. These queries are analogs of conjunctive queries, their unions and Boolean combinations, in which tree patterns play the role of atomic formulae. As in the relational case, they can be viewed as both queries and incomplete documents, and thus static analysis problems can also be viewed as finding certain answers of queries over such documents. We look at satisfiability of patterns under schemas, containment of queries for various features of XML used in queries, finding certain answers, and applications of pattern-based queries in reasoning about schema mappings for data exchange.

1 Introduction

Due to the complicated hierarchical structure of XML documents and the many ways in which it can interact with data, reasoning about XML data has become an active area of research, and many papers dealing with various aspects of static analysis of XML have appeared, see, e.g. [1, 6, 12, 16–18, 24, 26, 27, 29].

As most querying tasks for XML have to do with navigation through documents, reasoning/static analysis tasks deal with mechanisms for specifying interaction between navigation, data, as well as schemas of documents. Navigation mechanisms that are studied are largely of two kinds: they either describe paths through documents (most commonly using the navigational language XPath), or they describe *tree patterns*.

A tree pattern presents a *partial* description of a tree, along with some variables that can be assigned values as a pattern is matched to a complete document. For instance, a pattern $a(x)[b(x), c(y)]$ describes a tree with the root labeled a and two children labeled b and c; these carry data values, so that those in the a-node and the b-node are the same. This pattern matches a tree with root a and children b and c with all of them having data value 1, for instance; not only that, such a match produces the tuple $(1, 1)$ of data values witnessing the match. On the other hand, if in the tree the b and the c nodes carry value 2, there is no longer a match.

We deal with patterns that are naturally tree-shaped. This is contrast with some of the patterns appearing in the literature [9, 10] that can take the shape of arbitrary graphs (for instance, such a pattern can say that we have an a-node, that has b and c descendants, that in turn have the same d-descendant: this describes a directed acyclic graph rather than a tree). In many XML applications it is quite natural to use tree-shaped patterns though. For example, patterns used in specifying mappings between schemas (as needed in data integration and exchange applications) are such [3, 5, 7]. It is also

natural to use them for defining queries [4, 26] as well as for specifying incomplete XML data [8].

In database theory, there is a well-known duality between partial descriptions of databases (or databases with *incomplete* information), and *conjunctive queries*. Likewise for us, patterns can also be viewed as basic queries: in the above example, the pattern returns pairs (x, y) of data values. Viewing patterns as atomic formulas, we can close them under conjunction, disjunction, and quantification, obtaining analogs of relational conjunctive queries and their unions, for instance.

The main reasoning task we deal with is *containment* of queries. There are three main reasons for studying this question.

- Containment is the most basic query *optimization* task. Indeed, the goal of query optimization is to replace a given query with a more efficient but equivalent one; equivalence of course is testing two containment statements.
- Containment can be viewed as finding *certain answers* over incomplete databases, using the duality between queries and patterns. A pattern π describes an incomplete database; if, viewed as a query, it is contained in a query Q, then the certain answer to Q over π is true, and the converse also holds. This correspondence is well known for both relations and XML.
- Finally, containment is the critical task in *data integration*, specifically in query rewriting using views [22]. When a query needs to be rewritten over the source database, the correctness of a rewriting is verified by checking query containment.

The plan of the survey is as follows. We first explain the basic relevant notions in the relational case, particularly the pattern/query duality and the connection with incomplete information. We then define tree patterns, present their classification, and explain the notion of satisfaction in *data trees*, i.e., labeled trees in which nodes can carry data values. After that we deal with the basic pattern analysis problem: their satisfiability. Given that patterns are tree-shaped, satisfiability per se is trivial, but we handle it in the presence of a schema (typically given by an automaton).

We then introduce pattern-based queries, specifically analogs of conjunctive queries, their unions, and Boolean combination, and survey results on their containment. Using those results, we derive bounds on finding certain answers for queries over incomplete documents. Finally, we deal with reasoning tasks for pattern-based schema mappings, which also rely on a form of containment statement.

2 Relational patterns and pattern-based queries

Tableaux and naïve databases Relational patterns are known under the name of *tableaux* if one views them as queries, and as *naïve tables* if one views them as data. The instance below on the left is a usual relation, and the one on the right is a tableau/naïve table:

1	2	3	4
5	6	7	8

1	x	3	z
5	y	7	x

Some of the constant entries in relations can be replaced by *variables* in tableaux. Formally, we have two domains, $\mathcal{C}$ of constants and $\mathcal{V}$ of variables, and a relational vocabulary σ. A relational instance is an instance of σ over $\mathcal{C}$, and a naïve database is an instance over $\mathcal{C} \cup \mathcal{V}$. In case of a single relation, we talk about naïve tables rather than naïve databases.

A tableau has a list of variables, among those used in it, selected as 'distinguished' variables; that is, formally it is a pair $(D, \bar{x})$, where D is a naïve database and $\bar{x}$ is a tuple of variables among those mentioned in D.

As we already mentioned, there is a natural duality between incomplete databases and conjunctive queries. Each tableau $(D, \bar{x})$ can be viewed as a query $Q_D(\bar{x}) = \exists \bar{y} \bigwedge D$ where $\bar{y}$ is the list of variables in D except $\bar{x}$, and $\bigwedge D$ is the conjunction of all the facts in D. For instance, if D is the naïve table in the above picture, the query associated with (D, x) is $Q(x) = \exists y \exists z\ D(1, x, 3, z) \wedge D(5, y, 7, x)$. Likewise, every conjunctive query Q has a tableau *tab*(Q) which is obtained by viewing conjuncts in it as a database, and making the list of free variables its distinguished variables.

Homomorphisms A key notion for naïve databases and tableaux is that of a *homomorphism*. Given two naïve databases D_1 and D_2, a homomorphism h between them is a mapping h from $\mathcal{V}$ to $\mathcal{C} \cup \mathcal{V}$ defined on all the variables in D_1 so that, if R is a relation symbol in the vocabulary and $\bar{a}$ is a tuple in the relation R in D_1, then $h(\bar{a})$ is a tuple in the relation R in D_2. Of course $h(a_1, \ldots, a_n)$ stands for $(h(a_1), \ldots, h(a_n))$, and we assume $h(c) = c$ whenever $c \in \mathcal{C}$.

If h is a homomorphism from D_1 to D_2, we write $h : D_1 \to D_2$. Such a map is a homomorphism of two tableaux $(D_1, \bar{x}_1)$ and $(D_2, \bar{x}_2)$ if, in addition, $h(\bar{x}_1) = \bar{x}_2$. If we need to state that there is a homomorphism, but it is not important to name it, we will simply write $D_1 \to D_2$.

Homomorphisms can also be used to give semantics of incomplete databases. It is assumed that a naïve database D represents all complete databases D' (i.e., databases over $\mathcal{C}$) such that there is a homomorphism $h : D \to D'$. The set of all such D' is denoted by $[\![D]\!]$.

Note that the satisfiability problem for relational patterns expressed via naïve databases – whether the set $[\![D]\!]$ is not empty – is trivial, the answer is always yes. In the presence of constraints on the schema it can become a fairly complicated problem, sometimes even undecidable.

Containment Containment asks if for two queries, Q_1 and Q_2, the result of Q_1 is contained in the result of Q_2 on every input; equivalence asks if the results are always the same. We write $Q_1 \subseteq Q_2$ and $Q_1 = Q_2$ to denote containment and equivalence. Of course equivalence is just a special case of containment: $Q_1 = Q_2$ iff $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$.

The containment problem for conjunctive queries is solved via homomorphisms. Given two conjunctive queries Q_1 and Q_2, we have $Q_1 \subseteq Q_2$ iff there is a homomorphism $h : \mathit{tab}(Q_2) \to \mathit{tab}(Q_1)$; this makes the problem NP-complete [13].

In addition to conjunctive queries (sometimes abbreviated as CQs), we shall consider their unions and Boolean combinations. The former class, denoted by UCQs sometimes, is obtained by closing CQs under union (i.e., if Q_1, Q_2 are UCQs producing relations of the same arity, then $Q_1 \cup Q_2$ is a UCQ). For Boolean combinations of

conjunctive queries (abbreviated BCCQs), the additional closure rules are that $Q_1 \cap Q_2$, $Q_1 \cup Q_2$, and $Q_1 - Q_2$ are BCCQs.

For these classes containment is still decidable, and the complexity stays in NP for UCQs given explicitly as unions of CQs, and goes up to Π_2^p-complete for BCCQs [28].

Certain answers and naïve evaluation Now suppose we have a naïve database D and a query Q; assume that Q is Boolean. The standard notion of answering a query on an incomplete database is that of *certain answers*:

$$\mathit{certain}(Q, D) \;=\; \bigwedge \{Q(D') \mid D' \in [\![D]\!]\}$$

Let Q be a conjunctive query. Then, for an arbitrary database D', we have $D' \models Q$ iff there is a homomorphism $h : \mathit{tab}(Q) \to D'$. Thus, for an incomplete database D, we have the following easy equivalences:

$$\mathit{certain}(Q, D) = \texttt{true} \;\Leftrightarrow\; \forall D' \in [\![D]\!] :\; \mathit{tab}(Q) \to D' \;\Leftrightarrow\; \mathit{tab}(Q) \to D \;\Leftrightarrow\; D \models Q$$

Thus, to compute certain answers, all one needs to do is to run a query on the incomplete database itself. This is referred to as *naïve evaluation*. Note that the *data complexity* of finding certain answers is tractable, as it is the same as evaluation of conjunctive queries.

The fact that naïve evaluation works for Boolean conjunctive queries extends in two ways: to UCQs, and to queries with free variables [21]. In some way (for the semantics we considered) the result is optimal within the class of relational algebra queries [23]. In particular, naïve evaluation does *not* work for BCCQs (even though it was shown recently that data complexity of finding certain answers for BCCQs remains polynomial [19]).

3 Trees, patterns

3.1 Data trees

Data trees provide a standard abstraction of XML documents with data. First we define their structural part, namely unranked trees. A finite unranked tree domain is a non-empty, prefix-closed finite subset D of $\mathbb{N}^*$ (words over $\mathbb{N}$) such that $s \cdot i \in D$ implies $s \cdot j \in D$ for all $j < i$ and $s \in \mathbb{N}^*$. Elements of unranked tree domains are called nodes. We assume a countably infinite set $\mathcal{L}$ of possible labels that can be used to label tree nodes. An unranked tree is a structure $\langle D, \downarrow, \rightarrow, \lambda \rangle$, where

- D is a finite unranked tree domain,
- $\downarrow$ is the child relation: $s \downarrow s \cdot i$ for $s \cdot i \in D$,
- $\rightarrow$ is the next-sibling relation: $s \cdot i \rightarrow s \cdot (i+1)$ for $s \cdot (i+1) \in D$, and
- $\lambda : D \to \mathcal{L}$ is the labeling function assigning a label to each node.

We denote the reflexive-transitive closure of $\downarrow$ by $\downarrow^*$ (descendant-or-self), and the reflexive-transitive closure of $\rightarrow$ by $\rightarrow^*$ (following-sibling-or-self).

In data trees, nodes can carry not only labels but also *data values*. Given a domain $\mathcal{C}$ of data values (e.g., strings, numbers, etc.), a *data tree* is a structure $t = \langle D, \downarrow, \rightarrow, \lambda, \rho \rangle$, where $\langle D, \downarrow, \rightarrow, \lambda \rangle$ is an unranked tree, and $\rho : D \to \mathcal{C}$ assigns each node a data value. Note that in XML documents, nodes may have multiple attributes, but this is easily modeled with data trees.

3.2 Patterns

To explain our approach to defining tree-shaped patterns, consider first data trees restricted just to the child relation, i.e., structures $\langle D, \downarrow, \lambda, \rho\rangle$. They can be defined recursively: a node labeled with $a \in \mathcal{L}$ and carrying a data value $v \in \mathcal{C}$ is a data tree, and if $t_1, \ldots, t_n$ are trees, we can form a new tree by making them children of a node with label a and data value v.

Just like in the relational case, patterns can also use variables from $\mathcal{V}$. So our simplest case of patterns is defined as:

$$\pi := a(x)[\pi, \ldots, \pi] \tag{1}$$

with $a \in \mathcal{L}$ and $x \in \mathcal{C} \cup \mathcal{V}$. Here the sequence in $[\ldots]$ could be empty. In other words, if $\pi_1, \ldots, \pi_n$ is a sequence of patterns (perhaps empty), $a \in \mathcal{L}$ and $x \in \mathcal{C} \cup \mathcal{V}$, then $a(x)[\pi_1, \ldots, \pi_n]$ is a pattern. If $\bar{x}$ is the list of all the variables used in a pattern π, we write $\pi(\bar{x})$.

We denote patterns from this class by $\textbf{PAT}(\downarrow)$. As with conjunctive queries, the semantics can be defined via homomorphisms of their tree representations [8, 19], but here we give it in a different, direct way. The semantics of $\pi(\bar{x})$ is defined with respect to a data tree $t = \langle D, \downarrow, \rightarrow, \lambda, \rho\rangle$, a node $s \in D$, and a valuation $\nu : \bar{x} \to \mathcal{C}$ as follows: $(t, s, \nu) \models a(x)[\pi_1(\bar{x}_1), \ldots, \pi_n(\bar{x}_n)]$ iff

- $\lambda(s) = a$ (the label of s is a);
- $\rho(s) = \begin{cases} \nu(x) & \text{if } x \text{ is a variable} \\ x & \text{if } x \text{ is a data value;} \end{cases}$
- there exist not necessarily distinct children $s{\cdot}i_1, \ldots, s{\cdot}i_n$ of s so that $(t, s{\cdot}i_j, \nu) \models \pi_j(\bar{x}_j)$ for each $j \leq n$ (if $n = 0$, this last item is not needed).

We write $(t, \nu) \models \pi(\bar{x})$ if there is a node s so that $(t, s, \nu) \models \pi(\bar{x})$ (i.e., a pattern is matched somewhere in the tree). Also if $\bar{v} = \nu(\bar{x})$, we write $t \models \pi(\bar{v})$ instead of $(t, \nu) \models \pi(\bar{x})$. We also write $\pi(t)$ for the set $\{\bar{v} \mid t \models \pi(\bar{v})\}$.

A natural extension for these simple patterns is to include both vertical and horizontal navigation, resulting in the class $\textbf{PAT}(\downarrow, \rightarrow)$:

$$\begin{aligned} \pi &:= a(x)[\mu, \ldots, \mu] \\ \mu &:= \pi \rightarrow \ldots \rightarrow \pi \end{aligned} \tag{2}$$

with $a \in \mathcal{L}$ and $x \in \mathcal{C} \cup \mathcal{V}$ (and the sequences, as before, could be empty). The semantics is given by:

- $(t, s, \nu) \models a(x)[\mu_1(\bar{x}_1), \ldots, \mu_n(\bar{x}_n)]$ if $a(x)$ is satisfied in s by ν as before and there exist not necessarily distinct children $s{\cdot}i_1, \ldots, s{\cdot}i_n$ of s so that $(t, s{\cdot}i_j, \nu) \models \mu_j(\bar{x}_j)$ for each $j \leq n$.
- $(t, s, \nu) \models \pi_1(\bar{x}_1) \rightarrow \ldots \rightarrow \pi_m(\bar{x}_m)$ if there exist *consecutive* siblings $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_m$, with $s_1 = s$, so that $(t, s_i, \nu) \models \pi_i(\bar{x}_i)$ for each $i \leq m$.

Next we consider more expressive versions with transitive closure axes $\downarrow^*$ (descendant) and $\rightarrow^*$ (following sibling). As in [3, 19], we define general patterns by the rules:

$$\begin{aligned} \pi &:= a(x)[\mu, \ldots, \mu]//[\mu, \ldots, \mu] \\ \mu &:= \pi \rightsquigarrow \ldots \rightsquigarrow \pi \end{aligned} \tag{3}$$

Here a, x and π are as before, and μ stands for a sequence of trees, i.e., a forest such that the roots of its trees are *sequential* siblings in a tree, and each $\rightsquigarrow$ is either $\rightarrow$ or $\rightarrow^*$.

The class of such patterns is denoted by $\mathbf{PAT}(\Downarrow, \Rightarrow)$, with $\Downarrow$ we use both types of downward navigation ($\downarrow$ and $\downarrow^*$) and $\Rightarrow$ meaning that we use both types of horizontal navigation ($\rightarrow$ and $\rightarrow^*$). The semantics is extended as follows.

- $(t, s, \nu) \models a(x)[\mu_1, \ldots, \mu_n]//[\mu'_1, \ldots, \mu'_k]$ if the satisfaction of $a(x)$ in node s is as before, and there exist n not necessarily distinct children $s_1, \ldots, s_n$ of s such that $(t, s_i, \nu) \models \mu_i$ for each $i \leq n$, and there exist k not necessarily distinct descendants $s'_1, \ldots, s'_k$ of s such that $(t, s'_i, \nu) \models \mu'_i$ for each $i \leq k$.
- $(t, s, \nu) \models \pi_1(\bar{x}_1) \rightsquigarrow \ldots \rightsquigarrow \pi_m(\bar{x}_m)$ if there is a sequence $s_1, \ldots, s_m$ of nodes with $s_1 = s$ so that $(t, s_i, \nu) \models \pi_i(\bar{x}_i)$ for each $i \leq m$, and $s_i \rightarrow s_{i+1}$ whenever the ith $\rightsquigarrow$ is $\rightarrow$, and $s_i \rightarrow^* s_{i+1}$ whenever the ith $\rightsquigarrow$ is $\rightarrow^*$.

Notice that the semantics of patterns allows different μ_i to be mapped into the same nodes in a tree.

We also consider a class $\mathbf{PAT}(\Downarrow)$ of patterns which is a restriction of the most general patterns to downward navigation only. These are defined by the grammar

$$\pi := a(x)[\pi, \ldots, \pi]//[\pi, \ldots, \pi] \tag{4}$$

where each of the sequences of patterns can be empty.

We shall be using standard shorthand notations: $a(x)/\pi$ stands for $a(x)[\pi]$, while $a(x)//\pi$ denotes $a(x)//[\pi]$, and $a(x)/\pi//\pi'$ stands for $a(x)[\pi]//[\pi']$.

Finally, we also look at patterns with *wildcard*. In those patterns, we assume that labels come from $\mathcal{L} \cup \{\_\}$, where $\_$ is a new wildcard label that matches every label in a data tree. That is, if x is a variable, $\_(x)$ is true in (t, s, ν) if $\nu(x) = \rho(s)$, and if c is a constant, $\_(c)$ is true in a node s of a tree t if $\rho(s) = c$. In other words, wildcard allows us to disregard the label matching condition. We shall write $\mathbf{PAT}(\sigma, \_)$ for patterns that use axes from σ and wildcard.

4 Basic analysis of patterns

We now look at the satisfiability of patterns. For a set of σ of axes we look at the problem $\mathrm{SAT}(\sigma)$: its input is a pattern $\pi \in \mathbf{PAT}(\sigma)$, and the question is whether $\pi(t) \neq \emptyset$ for some data tree t, i.e., whether there is a data tree t such that $t \models \pi(\bar{v})$ for some valuation $\bar{v}$ of free variables of π. In other words, we want to know whether a pattern is realizable in some data tree.

Since our patterns are essentially tree-shaped, the problem as formulated above is trivial for them: the answer is trivially yes as one just turns a pattern into a tree. What is more interesting for us is satisfiability with a *schema*.

As common in the study of XML [25], we abstract a schema as an *unranked tree automaton*. Such an automaton over trees labeled with letters from a finite alphabet Σ is a tuple $\mathcal{A} = (S, \Sigma, \delta, F)$, where:

- S is a finite set of states,

– $F \subseteq S$ is the set of final states, and
– $\delta : S \times \Sigma \to 2^{(S^*)}$ is a transition function; we require that $\delta(s, a)$'s be regular languages over S for all $s \in S$ and $a \in \Sigma$. For reasoning about complexity, we represent values of the transition function by NFAs.

A run of $\mathcal{A}$ over a tree $t = \langle D, \downarrow, \to, \lambda \rangle$ (note that automata do not talk about data values) is a function $\rho_{\mathcal{A}} : D \to S$ such that for each node v with n children $v \cdot 0, \ldots, v \cdot (n-1)$, the word $\rho_{\mathcal{A}}(v \cdot 0) \cdots \rho_{\mathcal{A}}(v \cdot (n-1))$ is in the language $\delta(\rho_{\mathcal{A}}(v), \lambda(v))$. Of course, for a leaf v labeled a this means that v could be assigned state s iff the empty word ϵ is in $\delta(s, a)$. A run is accepting if $\rho_{\mathcal{A}}(\epsilon) \in F$, i.e., if the root is assigned an accepting state. A tree t is accepted by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t. The set of all trees accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$. A data tree is in $L(\mathcal{A})$ iff its "data-free" part is in $L(\mathcal{A})$ (i.e., the tree obtained by simply dropping the data-value assigning function ρ).

The problem we look at now $\mathrm{SAT}_{\mathrm{aut}}(\sigma)$: its input consists of a pattern $\pi \in \mathbf{PAT}(\sigma)$ and an automaton $\mathcal{A}$, and the question is whether there is a tree $t \in L(\mathcal{A})$ such that $\pi(t) \neq \emptyset$.

There are different versions of this problem depending on σ and the features allowed in the automaton $\mathcal{A}$; essentially all of them are known to be NP-complete. This result has appeared several times in the literature in different incarnations [5, 7–10]. For the definition of patterns as given here, the directly applicable one is the following result from [8].

Theorem 1. *The problem* $\mathrm{SAT}_{\mathrm{aut}}(\sigma, \_)$ *is in* NP. *Moreover, the problem* $\mathrm{SAT}_{\mathrm{aut}}(\downarrow)$ *is already* NP*-complete, as is the problem* $\mathrm{SAT}_{\mathrm{aut}}(\downarrow, \to, \_)$ *restricted to patters without variables.*

In fact the results hold if automata are given as DTDs, i.e., extended context-free grammars (and rather simple ones, see [8]). The case of $\mathrm{SAT}_{\mathrm{aut}}(\downarrow)$ restricted to trees without variables is tractable though, as such a pattern can be efficiently translated into an automaton, and the problem is reduced to checking nonemptiness of the product of two automata.

The upper NP bound is proved by a "cutting" technique: it shows that if there is a data tree $t \in L(\mathcal{A})$ in which the pattern π is satisfied, then there is one which is not too large in terms of π and $\mathcal{A}$ (a low degree polynomial).

5 Pattern-based queries

The most fundamental static analysis question is query equivalence/containment. As in the relational case, we consider conjunctive queries, their unions, and Boolean combinations. However, now the role of atomic formulae is played by patterns. That is, pattern-based conjunctive XML queries are obtained by closing patterns by conjunction and existential quantification. Since we have different classes of patterns $\mathbf{PAT}(\sigma)$, we have different classes of conjunctive queries denoted by $\mathrm{CQ}(\sigma)$. More precisely, $\mathrm{CQ}(\sigma)$ queries are of the form:

$$Q(\bar{x}) \;=\; \exists \bar{y} \bigwedge_{i=1}^{n} \pi_i(\bar{z}_i) \tag{5}$$

where each π_i is a $\mathbf{PAT}(\sigma)$ pattern, and each $\bar{z}_i$ is contained in $\bar{x}, \bar{y}$. The semantics is standard: $(t, \nu) \models Q(\bar{x})$ if there is an extension ν' of valuation ν to variables $\bar{y}$ such that $(t, \nu') \models \pi_i(\bar{z}_i)$ for every $i \leq n$. That is to say, one evaluates all the $\pi_i(t)$ and then combines the results as prescribed by the conjunction in (5), using the standard relational semantics. We also write $t \models Q(\bar{v})$ if $(t, \nu) \models Q(\bar{x})$ with $\nu(\bar{x}) = \bar{v}$ and, as usual, $Q(t)$ for $\{\bar{v} \mid t \models Q(\bar{v})\}$.

Observe that if the set σ of axes contains $\downarrow^*$ then we can restrict $\mathrm{CQ}(\sigma, \_)$ to the conjunctive queries which use a single pattern, i.e. queries of the form $\exists \bar{y} \pi(\bar{x}, \bar{y})$. In fact any query in $\mathrm{CQ}(\sigma, \_)$ is equivalent to a single-pattern $\mathrm{CQ}(\sigma, \_)$ query: it suffices to connect all patterns $\pi_1, \ldots \pi_n$ of the query as descendants of a common wildcard-labeled root.

As in the relational case, we extend CQs by taking their union (to the class UCQ) and Boolean combinations (to the class BCCQ). Formally, a query from $\mathrm{UCQ}(\sigma)$ is of the form $Q(\bar{x}) = Q_1(\bar{x}) \cup \ldots \cup Q_m(\bar{x})$, where each $Q_i(\bar{x})$ is a $\mathrm{CQ}(\sigma)$ query. It returns the union of answers to the Q_i's.

Queries in the class $\mathrm{BCCQ}(\sigma)$ are obtained as follows: take some queries $Q_1(\bar{x}), \ldots, Q_m(\bar{x})$ from $\mathrm{CQ}(\sigma)$ and consider a Boolean combination of them, i.e., close CQs under operations $Q \cap Q'$, $Q \cup Q'$, and $Q - Q'$. The semantics is extended naturally, with operations interpreted as intersection, union, and set difference, respectively.

The answer to a query $Q(\bar{x})$, from any of the above classes, on a data tree t is defined as $Q(t) = \{\nu(\bar{x}) \mid (t, \nu) \models Q(\bar{x})\}$. Note that our definitions of query classes ensure that $Q(t)$ is always finite.

The containment problem is formally stated as follows:

PROBLEM: CONTAINMENT-CQ(σ)
INPUT: queries $Q(\bar{x}), Q'(\bar{x}')$ in $\mathrm{CQ}(\sigma)$; QUESTION: is $Q \subseteq Q'$?

If instead of queries in $\mathrm{CQ}(\sigma)$ we use queries in $\mathrm{UCQ}(\sigma)$, we refer to the problem CONTAINMENT-UCQ(σ) and, if we use queries from $\mathrm{BCCQ}(\sigma)$, we refer to the problem CONTAINMENT-BCCQ(σ).

Note that one can look at satisfiability problems SAT-CQ, SAT-UCQ, and SAT-BCCQ, where the input is a query Q from a class and the question is whether $Q(t) \neq \emptyset$ for some tree. For the same reason as for patterns, the problems SAT-CQ and SAT-UCQ are trivial: such queries are always satisfiable. The problem SAT-BCCQ is the same as CONTAINMENT-BCCQ. Indeed, given a $\mathrm{BCCQ}(\sigma)$ query Q, it is not satisfiable iff Q is contained in $Q - Q$. Conversely, given two $\mathrm{BCCQ}(\sigma)$ queries Q_1, Q_2, we have $Q_1 \subseteq Q_2$ iff $Q_1 - Q_2$ is not satisfiable. The latter connection is actually important for providing upper bounds for containment as we shall deal with satisfiability instead.

6 Containment of pattern-based queries

We now look at the containment problems described in the previous section and start with a general upper bound. The following was shown in [14].

Theorem 2. *The problem* CONTAINMENT-BCCQ$(\Downarrow, \Rightarrow, _)$ *is in* Π_2^p.

The proof shows that the problem SAT-BCCQ$(\Downarrow, \Rightarrow, _)$ is in Π_2^p. Satisfiability of BCCQs can easily be reduced to simultaneous satisfiability of a CQ and *un*satisfiability of a UCQ. For this, we need to guess a witness tree t; then satisfiability of a CQ can be done in NP and unsatisfiability of a UCQ in coNP, giving us the bound. One can still use the cutting technique to reduce the size of this tree; however, this time the size is not polynomial but rather exponential. However, the only objects of exponential size are long non-branching paths in the tree, so they can be carefully re-labeled and encoded by polysize objects in a way that checking satisfiability or unsatisfiability of CQs and UCQs can still be done with the same complexity as before.

The next obvious question is about matching lower bounds. They can be shown with the simplest form of navigation, or, alternatively, with all the navigation but just for CQs [14].

Theorem 3. – *The problem* CONTAINMENT-BCCQ$(\downarrow)$ *is* Π_2^p*-complete.*
– *The problem* CONTAINMENT-CQ$(\Downarrow, \Rightarrow)$ *is* Π_2^p*-complete.*

Thus, we already see a big difference in the containment problem for XML pattern-based conjunctive queries, which is Π_2^p-hard, and relational CQs, for which the problem is in NP. The question is then when we can lower the complexity of containment to NP, to match the relational case.

One way to do so is to use the standard homomorphism technique. We know it will not work for all patterns due to complexity mismatch, but perhaps it will work for some. With each $\mathrm{CQ}(\sigma)$ query Q, we can associate a tableau *tab*(Q) which is essentially an incomplete tree obtained by parsing the patterns in Q. The full (and completely expected) definition is given in [14]; here we just give an illustrating example. Suppose we have a pattern $a(x)[b(y) \rightarrow c(x)]//d(y)$. The tableau is an incomplete tree with four nodes s_1, s_2, s_3, s_4 labeled a, b, c, d, respectively. The function ρ assigns x to s_1 and s_3, and y to s_2 and s_4. Finally the following hold: $s_1 \downarrow s_2$, $s_1 \downarrow s_3$, $s_2 \rightarrow s_3$, and $s_1 \downarrow^* s_4$.

Without transitive-closure axes, the standard connection between containment and tableaux homomorphism continues to work.

Theorem 4. *For queries from* $\mathrm{CQ}(\downarrow)$ *and* $\mathrm{CQ}(\downarrow, \rightarrow)$*, we have* $Q_1 \subseteq Q_2$ *iff there is a homomorphism from tab*(Q_2) *to tab*(Q_1)*. In particular, both* CONTAINMENT-CQ$(\downarrow)$ *and* CONTAINMENT-CQ$(\downarrow, \rightarrow)$ *are* NP*-complete.*

It is also possible to show that these results extend to UCQs, using techniques in the spirit of [28].

We know that CONTAINMENT-CQ$(\Downarrow, \Rightarrow)$ cannot be in NP (otherwise NP and coNP would be the same). But it turns out that CONTAINMENT-CQ$(\Downarrow, \rightarrow)$ does stay in NP. This, however, cannot be shown by exhibiting a homomorphism from *tab*(Q_2) to *tab*(Q_1). Consider, for instance, queries $Q_1 = \exists x\ a(x)//b(x)[c(x)]$ and $Q_2 = \exists x\ a(x)//c(x)$. While it is easy to see that $Q_1 \subseteq Q_2$, there is no homomorphism from *tab*(Q_2) to *tab*(Q_1). However, if we define *tab*$^*(Q)$ by replacing relation $\downarrow^*$ in *tab*(Q) by the transitive closure of the union of $\downarrow$ and $\downarrow^*$, we do get a homomorphism from *tab*$^*(Q_2)$ to *tab*$^*(Q_1)$. In fact, [14] showed:

Proposition 1. *For queries from* $\mathrm{CQ}(\Downarrow, \rightarrow)$*, we have* $Q_1 \subseteq Q_2$ *iff there is a homomorphism from* $tab^*(Q_2)$ *to* $tab^*(Q_1)$*. In particular,* CONTAINMENT-CQ$(\Downarrow, \rightarrow)$ *remains* NP*-complete.*

When wildcard is added, things change dramatically. Consider, for instance, two Boolean queries $Q_1 = r[a, b]$ and $Q_2 = r[\_ \rightarrow \_]$. We know $Q_1 \subseteq Q_2$ but there is no homomorphism between the tableau as in $tab(Q_1)$ the relation $\rightarrow$ is empty. It is possible to recover the result about CONTAINMENT-CQ$(\downarrow)$ and CONTAINMENT-CQ$(\downarrow, \rightarrow)$ when wildcard is used *except* at the root of the pattern (by again modifying the tableau and establishing a homomorphism) but beyond that little is known. In fact in the presence of wildcard existing results do not extend to UCQs even with restriction on the use of wildcard: the problem CONTAINMENT-UCQ$(\downarrow, \rightarrow, _)$ is Π_2^p-complete, as is CONTAINMENT-UCQ$(\Downarrow)$ with wildcard used anywhere except the root.

As we did before, we can relativize the containment problem to a schema expressed as an unranked tree automaton. Such a problem (indicated again by subscript AUT), takes as an input two queries Q_1, Q_2 and an automaton $\mathcal{A}$, and checks whether $Q_1(t) \subseteq Q_2(t)$ for every $t \in L(\mathcal{A})$. The addition of schemas adds one exponent to the complexity.

Theorem 5. *The problem* CONTAINMENT-BCCQ$_{\text{aut}}(\Downarrow, \Rightarrow, _)$ *is in* 2EXPTIME*. Furthermore,* CONTAINMENT-CQ$_{\text{aut}}(\Downarrow, \Rightarrow)$ *is already* 2EXPTIME*-hard.*

Finally, one can extend queries with inequality comparisons of data values. This makes the problem undecidable for BCCQs and all the axes, or for CQs with $\downarrow$ and $\downarrow^*$ under schemas.

Pattern containment vs XPath containment There has been significant interest in containment of XPath queries, see, e.g., [29] for a survey. In general, pattern queries considered here are incompatible with XPath: our queries return tuples of data values, while XPath queries return n-tuples of nodes, for $n \leq 2$. However, the cases of Boolean XPath queries (i.e., $n = 0$) and Boolean pattern-based queries are indeed comparable, and we offer a comparison here.

The closest language to the classes we consider here is the fragment of XPath called XP in [27]. Boolean queries from XP (with data variables and existential semantics) are tightly related to Boolean queries from $\mathrm{UCQ}(\Downarrow, \_)$. In particular, any Boolean $\mathrm{UCQ}(\Downarrow, \_)$ query (possibly with data inequalities) can be viewed as a Boolean XP query. Conversely, any Boolean XP query written in disjunctive normal form can be viewed as a Boolean $\mathrm{UCQ}(\Downarrow, \_)$ with inequalities, but with an additional restriction that patterns be evaluated at the root.

It was shown in [27] that Boolean containment of XP without wildcard is in Π_2^p, and therefore so is CONTAINMENT-UCQ$(\Downarrow)$ (even with inequalities) when restricted to boolean UCQs without wildcard. Moreover Boolean containment of XP is Π_2^p-hard in some restricted fragments of XP without wildcard, and undecidable in the presence of wildcard (due to inequalities). However lower bounds do not immediately carry over to containment of Boolean $\mathrm{UCQ}(\Downarrow, \_)$ queries because, in the presence of disjunction, XP formulae need not be in disjunctive normal form, and the disjunctive normal form

may be exponential in the size of the the original XP query; moreover XP patterns are evaluated at the root.

The containment of Boolean queries from $\mathrm{CQ}(\Downarrow, \_)$ without variables was also considered in [26] where it was shown to be in coNP. The problem was also proved coNP-hard for evaluation of patterns at the root. These results were later extended in [27] by introducing disjunction in patterns and schemas. They imply that containment of Boolean $\mathrm{UCQ}(\Downarrow, \_)$ queries without variables is still in coNP, while in the presence of schemas it is in ExpTime.

7 Certain answers over patterns

We have already said that the containment provides a way to address the problem of finding certain answers to queries over incomplete databases. In the relational case, we saw the equivalence $\mathit{certain}(Q, D) = \mathtt{true} \Leftrightarrow \mathit{tab}(Q) \to D$, which is the same as saying $Q_D \subseteq Q$. Here Q_D is the canonical query of the database D, i.e., the conjunction of all the facts in D preceded by existentially quantifying all the variables in D. For instance, if D contains tuples $(1, x)$ and (x, y), then Q_D is $\exists x \exists y\, D(1, x) \wedge D(x, y)$.

In the case of XML, the standard view of incomplete documents is that of tree patterns [2, 8]. For instance, a pattern in $\mathbf{PAT}(\downarrow)$ specifies the child relation, but no next-sibling relation, and nodes may contain incomplete information as data associated with them. In a tree in $\mathbf{PAT}(\Downarrow, \Rightarrow, \_)$, structural information may be missing too. Consider, for instance, a pattern $a(1)//b(x)[c(x) \to^* a(3)]$. It represents all trees in which there is an a-node holding value 1, with a b-descendant holding some value, that has two children: a c-node with the same value, and an a-node with value 3, about which we also know that it appears after the c-node in the sibling order.

Thus, as in the relational case, a pattern $\pi(\bar{x})$ represents all data trees t such that $t \models \pi(\bar{v})$ for some valuation $\bar{v}$ of free variables $\bar{x}$. In the above example, a tree $a(1)/b(2)/b(1)[c(1) \to c(2) \to a(3)]$ is one such tree. By analogy with the relational case, we write $[\![\pi]\!]$ for all the trees represented by a pattern. Also, as in the relational case, this can be defined via homomorphisms (which now are a bit more complex as they have to act on both tree nodes and data values; see [8] for details).

If we have a query $Q(\bar{x})$, certain answers over a pattern π are defined, as before, by $\mathit{certain}(Q, \pi) = \bigcap\{Q(t) \mid t \in [\![\pi]\!]\}$. If Q is Boolean, intersection is replaced by conjunction, of course.

We are interested in the complexity of finding certain answers: that is, checking, for a query Q, a pattern π, and a tuple of values $\bar{a}$, whether $\bar{a} \in \mathit{certain}(Q, \pi)$.

As in the relational case, certain answers can be reduced to the containment problem. If Q is Boolean then $\mathit{certain}(Q, \pi) = \mathtt{true}$ iff $Q_\pi \subseteq Q$, where Q_π is simply $\exists \bar{x} \pi(\bar{x})$. A similar equivalence holds for arbitrary queries as well.

Thus, it appears that we can lift results for containment to state results about certain answers. However, this is only partly true. When we deal with query answering, we are interested in a finer classification of complexity, namely:

- *Data complexity*, when the query Q is fixed and only π and $\bar{a}$ are inputs; and
- *Combined complexity*, when Q, π, and $\bar{a}$ are inputs.

In relational databases, it is common to have an exponential gap between data and combined complexity: for instance, data complexity of all first-order queries is very low (AC^0, i.e., a subset of DLOGSPACE), while combined complexity is NP-complete for CQs and PSPACE-complete for first-order.

We start with upper bounds. In [8], it was shown, using the cutting technique, that data complexity of UCQs is in CONP; the proof yielded non-elementary combined complexity though. These results were refined in [19] which showed:

Theorem 6. *For finding certain answers to* BCCQs*, data complexity is in* CONP*, and combined complexity is in* Π_2^p.

What about matching lower bounds? It turns out that they can be achieved quite easily. The following combines results in [8, 19]. Below we say that data complexity of a class of queries is CONP-hard if there exists a query from that class for which data complexity is CONP-hard.

Theorem 7. *Data complexity of finding certain answers is* CONP*-complete for:*

- $\mathrm{CQ}(\downarrow, \rightarrow)$ *queries over* $\mathbf{PAT}(\downarrow)$*;*
- $\mathrm{UCQ}(\downarrow, \_)$ *queries over* $\mathbf{PAT}(\Downarrow, \rightarrow)$*;*

Furthermore, combined complexity of finding certain answers to $\mathrm{UCQ}(\downarrow, \rightarrow)$ *queries over* $\mathbf{PAT}(\downarrow, \rightarrow)$ *is* Π_2^p*-hard.*

We now look at ways of lowering the complexity, especially data complexity of finding certain answers. Recall that for Boolean relational CQs over incomplete documents, we have the equivalence $\mathit{certain}(Q, D) = \mathtt{true} \Leftrightarrow D \models Q$. More generally, $\mathit{certain}(Q, D)$ can be obtained by evaluating Q on D and then dropping any tuples containing variables. This is referred to as *naïve evaluation*, and when it computes certain answers, we say that it works for a particular class of queries over a class of patterns.

To see when naïve evaluation works for XML queries, we define *rigid* patterns. These are given by

$$\pi := a(x)[\pi \rightarrow \ldots \rightarrow \pi] \tag{6}$$

They can also be seen as patterns in (2), where the μ sequence appears just once. For instance, $a(x)[b(y)[c(x) \rightarrow d(y)] \rightarrow b(2) \rightarrow c(3)[d(y) \rightarrow a(1)]]$ is a rigid pattern: it completely specifies the tree structure via the $\downarrow$ and $\rightarrow$ relations, leaving only data potentially incomplete. We write $\mathbf{PAT}_{\mathrm{rigid}}$ for the class of rigid patterns. The following combines results from [8, 15].

Theorem 8. *Naïve evaluation works for* $\mathrm{UCQ}(\Downarrow, \Rightarrow, \_)$ *queries over* $\mathbf{PAT}_{rigid}$*, and for* $\mathrm{UCQ}(\Downarrow, \_)$ *queries over* $\mathbf{PAT}(\downarrow)$*. Thus, in both cases data complexity of finding certain answers is tractable.*

For BCCQs, even rigid ones, naïve evaluation no longer works. Nonetheless, a more complex tractable algorithm can be devised [19]. In fact, such an algorithm first had to be applied in the relational case (where it had not been known until [19]) and then adapted to the XML case.

Theorem 9. *Data complexity of certain answers for* $\mathrm{BCCQ}(\downarrow, \rightarrow)$ *queries over* $\mathbf{PAT}_{rigid}$ *is in* PTIME. *Their combined complexity is* Π_2^p*-complete, but it drops to* NP*-complete for* $\mathrm{UCQ}(\downarrow, \rightarrow)$ *queries over* $\mathbf{PAT}_{rigid}$.

Another question is what happens in the presence of schemas. That is, what happens if the trees must conform to a schema given by an automaton $\mathcal{A}$, and we defined certain answers as $\mathit{certain}_{\mathcal{A}}(Q, \pi) = \bigcap\{Q(t) \mid t \in [\![\pi]\!] \cap L(\mathcal{A})\}$. We then refer to finding certain answers under schemas. For talking about data complexity, we assume that only π is the input. It turns out that there is little hope of finding well behaved classes:

Proposition 2. *Data complexity of finding certain answers under schemas is* coNP*-complete for* $\mathrm{CQ}(\downarrow)$ *queries over* $\mathbf{PAT}(\downarrow)$.

8 Tree patterns in data exchange

As mentioned in the introduction, one area where pattern-based queries are of particular importance is integration and exchange of data. We now consider the typical setting of data exchange, cf. [5]. In data exchange, we need to move data between databases of different schemas. Since we are talking about XML, we deal with XML schemas, given by two automata $\mathcal{A}_{\mathbf{s}}$ and $\mathcal{A}_{\mathbf{t}}$, describing source and target schemas respectively. The correspondence between them is provided by a set $\Sigma_{\mathbf{st}}$ of pairs of queries $(Q_{\mathbf{s}}(\bar{x}, \bar{y}), Q_{\mathbf{t}}(\bar{x}, \bar{z}))$ from $\mathrm{CQ}(\sigma)$.

A *schema mapping* is then a triple $\mathcal{M} = \langle \mathcal{A}_{\mathbf{s}}, \mathcal{A}_{\mathbf{t}}, \Sigma_{\mathbf{st}} \rangle$. We let $\mathrm{SM}(\sigma)$ stand for the class of schema mappings where all the CQs in $\Sigma_{\mathbf{st}}$ are from $\mathrm{CQ}(\sigma)$.

Given two data trees t, t', we say that t' is a *solution* for t under $\mathcal{M}$ if:

1. $t \in L(\mathcal{A}_{\mathbf{s}})$ and $t' \in L(\mathcal{A}_{\mathbf{t}})$
2. $\exists \bar{y} Q_{\mathbf{s}}(t) \subseteq \exists \bar{z} Q_{\mathbf{t}}(t')$.

The semantics of a mapping $\mathcal{M}$, denoted by $[\![\mathcal{M}]\!]$, is the set of pairs of trees (t, t') so that t' is a solution for t.

The second condition is a containment statement, albeit a bit unusual one. It does not say that the CQ $\exists \bar{y} Q_{\mathbf{s}}(\bar{x}, \bar{y})$ is contained in the CQ $\exists \bar{z} Q_{\mathbf{t}}(\bar{x}, \bar{y})$ but rather that the result of the first CQ on t is contained in the result of the second CQ on t'.

Another, more conventional way, to read that statement is as follows: for all values $\bar{x}, \bar{y}$ making $Q_{\mathbf{s}}$ true in t, there exist values $\bar{z}$ so that $Q_{\mathbf{t}}(\bar{x}, \bar{z})$ is true in t'.

The basic reasoning tasks about schema mappings relate to their consistency, or satisfiability:

- The problem $\mathrm{SAT}_{\mathrm{SM}}(\sigma)$ takes a $\mathrm{SM}(\sigma)$ mapping $\mathcal{M}$ as an input and asks whether $[\![\mathcal{M}]\!] \neq \emptyset$. That is, it checks whether the mapping makes sense.
- The problem $\forall\mathrm{SAT}_{\mathrm{SM}}(\sigma)$ takes a $\mathrm{SM}(\sigma)$ mapping $\mathcal{M}$ as an input and asks whether *every* tree $t \in L(\mathcal{A}_{\mathbf{s}})$ has a solution, i.e., whether the mapping always makes sense.

The following was shown in [7, 11].

Theorem 10. – *The problem* $\mathrm{SAT}_{\mathrm{SM}}(\Downarrow, \Rightarrow, \_)$ *is in* EXPTIME. *In fact the problem* $\mathrm{SAT}_{\mathrm{SM}}(\Downarrow)$ *is already* EXPTIME*-complete.*

– *The problem* $\forall\mathrm{SAT}_{\mathrm{SM}}(\Downarrow, \Rightarrow, \_)$ *is in* Π_2^{EXP}. *In fact the problem* $\forall\mathrm{SAT}_{\mathrm{SM}}(\Downarrow, \_)$ *is already* Π_2^{EXP}*-complete.*

The class Π_2^{EXP} is the second level of the exponential hierarchy; it is to EXPTIME what Π_2^p is to PTIME. Being Π_2^{EXP}-complete means being in EXPSPACE and NEXPTIME-hard (incidentally, that was the first bound shown for $\forall\mathrm{SAT}_{\mathrm{SM}}(\Downarrow, \_)$ in [3], which was later improved in [11]).

Among restrictions imposed on schema mappings a common one is to restrict schemas to be *nested-relational DTDs*. These specify sequences of labels below a given one in a tree; they consist of rules like *book* $\rightarrow$ *title, author*$^+$*, chapter*$*$*, publisher?*, saying that a *book*-labeled node must have a *title* child, followed by one or more *author* children, followed by zero or more *chapter* children, and possible followed by a *publisher*-labeled node.

For instance, [3] showed that when schemas are given by nested-relational DTDs, the complexity of $\mathrm{SAT}_{\mathrm{SM}}(\Downarrow, \Rightarrow, \_)$ drops to PSPACE-complete. If, in addition, all the queries used in $\Sigma_{\mathbf{st}}$ are from $\mathrm{CQ}(\downarrow)$, then $\forall\mathrm{SAT}_{\mathrm{SM}}(\downarrow)$ can be solved in polynomial time.

Another variation of schema mappings that was considered allows augmenting CQs used in $\Sigma_{\mathbf{st}}$ with explicit equality and inequality comparisons. When just equality is allowed, we talk about the class $\mathrm{SM}(\sigma, =)$; if inequalities are allowed too, we talk about $\mathrm{SM}(\sigma, =, \neq)$. This addition increases the complexity of reasoning tasks dramatically [3].

Theorem 11. – *Both* $\mathrm{SAT}_{\mathrm{SM}}(\downarrow, \rightarrow, =)$ *and* $\mathrm{SAT}_{\mathrm{SM}}(\downarrow, \rightarrow, \neq)$ *are undecidable.*
– *Both* $\mathrm{SAT}_{\mathrm{SM}}(\Downarrow, =)$ *and* $\mathrm{SAT}_{\mathrm{SM}}(\Downarrow, \neq)$ *are undecidable as well.*
– *When schemas are nested relational DTDs,* $\mathrm{SAT}_{\mathrm{SM}}(\Downarrow, =)$ *is* NEXPTIME*-complete, but* $\mathrm{SAT}_{\mathrm{SM}}(\downarrow, \rightarrow, =)$ *remains undecidable.*

Acknowledgment This work was supported by EPSRC grants G049165 and J015377.

References

1. S. Abiteboul, B. Cautis, T. Milo. Reasoning about XML update constraints. In *PODS'07*, pages 195–204.
2. S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. *ACM TODS*, 31(1):208–254, 2006.
3. S. Amano, L. Libkin, F. Murlak. XML schema mappings. In *PODS'09*, pages 33–42.
4. S. Amer-Yahia, S. Cho, L. Lakshmanan, D. Srivastava. Tree pattern query minimization. *VLDB J.* 11(4): 315-331 (2002).
5. M. Arenas, P. Barceló, L. Libkin, F. Murlak. *Relational and XML Data Exchange*. Morgan & Claypool, 2010.
6. M. Arenas, W. Fan, L. Libkin. On the complexity of verifying consistency of XML specifications. *SIAM J. Comput.* 38(3): 841-880 (2008).
7. M. Arenas, L. Libkin. XML data exchange: consistency and query answering. *J. ACM* 55(2): (2008).
8. P. Barceló, L. Libkin, A. Poggi, C. Sirangelo. XML with incomplete information. *J. ACM*, 58:1 (2010).

9. H. Björklund, W. Martens, T. Schwentick. Optimizing conjunctive queries over trees using schema information. *MFCS'08*, pages 132–143.
10. H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. *JCSS* 77(3): 450-472 (2011).
11. M. Bojanczyk, L. Kolodziejczyk, F. Murlak. Solutions in XML data exchange. In *ICDT 2011*, pages 102-113.
12. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Regular XPath: constraints, query containment and view-based answering for XML documents. In *LID'08*.
13. A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC 1977*, pages 77–90.
14. C. David, A. Gheerbrant, L. Libkin, W. Martens. Containment of pattern-based queries over data trees. In *ICDT 2013*, pages 201-212.
15. C. David, L. Libkin, F. Murlak. Certain answers for XML queries. In *PODS 2010*, pages 191–202.
16. W. Fan, L. Libkin. On XML integrity constraints in the presence of DTDs. *J. ACM* 49(3): 368–406 (2002).
17. D. Figueira. Satisfiability of downward XPath with data equality tests. *PODS'09*, 197-206.
18. P. Genevés and N. Layaida. A system for the static analysis of XPath. *ACM TOIS* 24 (2006), 475–502.
19. A. Gheerbrant, L. Libkin, and T. Tan. On the complexity of query answering over incomplete XML documents. *ICDT 2012*, 169–181.
20. G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *J. ACM* 53 (2006), 238–272.
21. T. Imieliński and W. Lipski. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.
22. M. Lenzerini. Data integration: a theoretical perspective. In *PODS'02*, pages 233–246.
23. L. Libkin. Incomplete information and certain answers in general data models. In *PODS'11*, pages 59–70.
24. L. Libkin, C. Sirangelo. Reasoning about XML with temporal logics and automata. *J. Applied Logic*, 8:2, 210–232 (2010).
25. W. Martens, F. Neven, T. Schwentick. Simple off the shelf abstractions for XML schema. *SIGMOD Record* 36(3): 15-22 (2007).
26. G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1): 2–45, 2004.
27. F. Neven, T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *LMCS*, 2(3): (2006).
28. Y. Sagiv, M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM* 27(4): 633-655 (1980).
29. Th. Schwentick. XPath query containment. *SIGMOD Record* 33(1): 101-109 (2004).