Discovering XSD Keys from XML Data

MARCELO ARENAS, PUC Chile and University of Oxford
JONNY DAENEN and FRANK NEVEN, Hasselt University and Transnational University of Limburg
MARTIN UGARTE, PUC Chile
JAN VAN DEN BUSSCHE, Hasselt University and Transnational University of Limburg
STIJN VANSUMMEREN, Université Libre de Bruxelles (ULB)

A great deal of research into the learning of schemas from XML data has been conducted in recent years to enable the automatic discovery of XML schemas from XML documents when no schema or only a low-quality one is available. Unfortunately, and in strong contrast to, for instance, the relational model, the automatic discovery of even the simplest of XML constraints, namely XML keys, has been left largely unexplored in this context. A major obstacle here is the unavailability of a theory on reasoning about XML keys in the presence of XML schemas, which is needed to validate the quality of candidate keys. The present article embarks on a fundamental study of such a theory and classifies the complexity of several crucial properties concerning XML keys in the presence of an XSD, like, for instance, testing for consistency, boundedness, satisfiability, universality, and equivalence. Of independent interest, novel results are obtained related to cardinality estimation of XPath result sets. A mining algorithm is then developed within the framework of levelwise search. The algorithm leverages known discovery algorithms for functional dependencies in the relational model, but incorporates the properties mentioned before to assess and refine the quality of derived keys. An experimental study on an extensive body of real-world XML data evaluating the effectiveness of the proposed algorithm is provided.

Categories and Subject Descriptors: H.2.8 [**Information Systems**]: Database Management—*Database Applications, Data mining*

General Terms: Algorithms, Languages, Experimentation, Theory

Additional Key Words and Phrases: XML key

ACM Reference Format:
Marcelo Arenas, Jonny Daenen, Frank Neven, Martin Ugarte, Jan van den Bussche, and Stijn Vansummeren. 2014. Discovering XSD keys from XML data. ACM Trans. Datab. Syst. 39, 4, Article 28 (December 2014), 49 pages.
DOI: http://dx.doi.org/10.1145/2638547

The authors used the infrastructure of the VSC- Flemish Supercomputer Center, funded by the Hercules Foundation and the Flemish Government. The authors acknowledge the financial support of the Fondecyt grant no. 1131049, FP7-ICT-233599, FWO G082109, and ERC grant agreement DIADEM, no. 246858.
Authors' addresses: M. Arenas, Pontificia Universidad Catolica de Chile (PUC Chile), Av Libertador Bernardo O Higgins 340, Santiago, Region Metropolitana, Chile and University of Oxford, Wellington Square, Oxford OX1 2JD, UK; J. Daenen, F. Neven (corresponding author), Hasselt University and Transnational University of Limburg, Agoralaan D, 3900 Diepenbeek, Belgium; email: frank.neven@uhasselt.be; M. Ugarte, Pontificia Universidad Catolica de Chile (PUC Chile), Av Libertador Bernardo O Higgins 340, Santiago, Region Metropolitana, Chile; J. van den Bussche, Hasselt University and Transnational University of Limburg, Agoralaan D, 3900 Diepenbeek, Belgium; S. Vansummeren, Universite Libre de Bruxelles, Franklin Rooseveltlaan 50, 1050 Brussels, Belgium.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2014 ACM 0362-5915/2014/12-ART28 $15.00
DOI: http://dx.doi.org/10.1145/2638547

1. INTRODUCTION

The automatic discovery of constraints from data is a fundamental problem in the scientific database literature, especially in the context of the relational model in the form of key, foreign key, and functional dependency discovery (e.g., [Mannila and Räihä 1991]). Although the absence of DTDs and XML Schema Definitions (XSDs) for XML data occurring in the wild has driven a multitude of research on learning of XML schemas [Bex et al. 2009, 2010a, 2010b, 2007, 2008; Garofalakis et al. 2003], the automatic inference of constraints has been left largely unexplored (we refer to Section 2 for a discussion on related work). In this article, we address the problem of *XML key mining* whose core formulation asks to find all XML keys valid in a given XML document. We use a formalization of XSD keys (defined in Section 3) consistent with the definition of XML keys by W3C [2004]. We develop a key mining algorithm within the framework of levelwise search that additionally leverages discovery algorithms for functional dependencies in the relational model. Our algorithm iteratively refines keys based on a number of quality requirements; a significant portion of the article is devoted to a study of the complexity of testing these requirements.

To illustrate the challenges of key mining in the presence of a schema, we first introduce the following example.

Example 1.1. Consider the key

$$\phi := (\underbrace{(\text{order}, q_{\texttt{order}})}_{\text{context } c}, \quad \underbrace{\texttt{.//book}}_{\text{target path } \tau}, \quad \underbrace{(\texttt{.//title}, \ \texttt{.//year})}_{\text{key paths } p_1, p_2, \ldots}).$$

Here, the pair $(\text{order}, q_{\texttt{order}})$ is a *context* consisting of the label "order" and the state or type[1] $q_{\texttt{order}}$ that identifies the context nodes for which ϕ is to be evaluated. Further, `.//book` is an XPath expression, called *target path*, selecting within every context node a set of target nodes. The key constraint now states that every target node must be uniquely identified by the record determined by the key paths `.//title` and `.//year`, which are XPath expressions as well. In other words, no two target nodes should have both the same title and the same year. A schematic representation of the semantics of a key is given in Figure 2. So, over the XML document t displayed in Figure 1, the key ϕ gives rise to the table $R_{\phi,t}$ as follows.

$(\text{order}, q_{\texttt{order}})$	`.//book`	`.//title`	`.//year`
$(o_1,$	$b_1,$	*'Movie analysis'*,	*2012*$)$
$(o_1,$	$b_2,$	*'Programming intro'*,	*2012*$)$
$(o_2,$	$b_3,$	*'Programming intro'*,	*2012*$)$

In Figure 1, the names of the order and book nodes from left to right are o_1, o_2, and b_1, b_2, b_3, respectively, and every order node has type $q_{\texttt{order}}$. Then, ϕ holds in t if the functional dependency

$$(\text{order}, q_{\texttt{order}}), \texttt{.//title}, \texttt{.//year} \quad \rightarrow \quad \texttt{.//book}$$

holds in $R_{\phi,t}$. That is, within the same context node, "title" and "year" uniquely determine the "book" element.

As a necessary condition for a key to be valid on a tree t, the XML key specification [W3C 2004, Section 3.11.4] requires every key path to always select precisely one node

[1]Types are defined in the accompanying schema, which is not given here but discussed in Section 3.

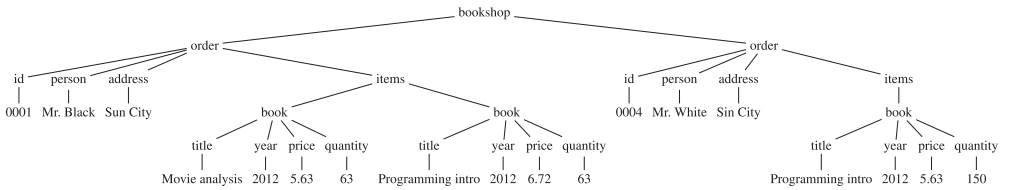


Fig. 1. An example XML tree (order and book nodes are named o_1, o_2 and b_1, b_2, b_3 from left to right, respectively).

carrying a data value.[2] The key is then said to *qualify* on t. As an example,

$$\phi' := ((\text{bookshop}, q_{\texttt{bookshop}}), \quad \texttt{.//order}, \quad (\texttt{.//address})),$$

qualifies for the particular tree given in Figure 1 (assuming every node labeled "bookshop" has type $q_{\texttt{bookshop}}$) since every target node o_1 and o_2 has precisely one address node. But, the accompanying XSD might allow XML documents without or with multiple addresses, for which ϕ' would not qualify. So, qualifying for the given document does not necessarily entail qualifying for every document in the schema. We say that a key is *consistent* with respect to an XSD if the key qualifies on every document satisfying the XSD. As a quality criterion for keys, we want our mining algorithm to only consider consistent keys. We therefore start by studying the complexity of deciding consistency and obtain the pleasantly surprising result that consistency can be tested in polynomial time for keys disallowing disjunction on the topmost level. We show that consistency for general keys is CONP-hard and even PSPACE-hard for keys with regular expressions (that are not allowed in W3C keys).

In addition to consistency, we want to enforce a number of additional quality requirements on keys. In particular, we want to disregard keys that can only select an a priori bounded number of target nodes independent of the size of the input document. Since the main purpose of a key is to ensure uniqueness within a collections of nodes, it does not make sense to consider bounded keys for which the size of this collection is fixed in advance and cannot grow with the size of the document. Similarly, we want to ignore so-called universal keys that hold in every document. We obtain that testing for bounded and universal keys is tractable.

A final theoretical theme of this article is that of reasoning about keys. On the negative side and in strong contrast to reasoning about relational keys [Abiteboul et al. 1995; Ramakrishnan and Gehrke 2003] or XML keys without an accompanying schema [Buneman et al. 2002, 2003], we show that testing satisfiability, equivalence, and implication between keys is EXPTIME-hard. As an aside, we show that a milder form of equivalence, namely that of target path equivalence (i.e., determining that two target paths always select the same set of target nodes over documents satisfying the schema), is tractable. The latter can be used as an instrument to reduce the number of candidate target paths.

After laying the theoretical groundwork given before, we turn to the theme of mining. Example 1.1 indicates how XML key mining can leverage algorithms for the discovery of functional dependencies (FDs) over a relational database. Indeed, once a context c and a target path τ are determined, any FD of the form $c, p_1, \ldots, p_n \rightarrow \tau$ that holds in the relational encoding $R_{(c,\tau,\overline{P}),t}$ entails the key $(c, \tau, (p_1, \ldots, p_n))$ in t where $\overline{P}$ is a sequence consisting of all possible consistent key paths. Of course, it remains to

[2]Actually, the specification is a bit more general in allowing the use of attributes. For ease of presentation, we disregard attributes and let leaf nodes carry data values. We note that all the results in the article can be easily extended to include attributes.

investigate how to efficiently explore the search space of candidate contexts c, target paths τ, and consistent key paths p. To this end, we embrace the framework of levelwise search (as, e.g., described by Mannila and Toivonen [1997]) to enumerate target and key paths. The components of this framework consist of a search space U, a Boolean search predicate q, and a specialization relation $\preceq$ that is a partial order on U and monotone with respect to q. In particular, the partial order arranges objects from most general to most specific and when q holds for an object then q should also hold for all generalizations of this object. The solution then consists of all objects $u \in U$ for which $q(u)$ holds, enumerated according to the specialization relation while avoiding testing objects for which q cannot hold anymore given already obtained information.

We define a target path miner within the framework just described as follows: the search predicate holds for a target path when the number of selected target nodes exceeds a predetermined threshold value, and the partial order $\preceq$ is determined by containment among target paths. To streamline computation, we utilize a syntactic one-step specialization relation $\prec_1$ that we prove optimal with respect to the considered partial order. Furthermore, the search predicate can be solely evaluated on a much smaller prefix tree representation of the input document and that therefore does not need access to the original document. In addition, we define a one-key path miner that searches for all consistent single-key paths p (with respect to the already determined context and target path). Specifically, the search predicate holds for a key path p when p selects at most one key node (with respect to the given context and target path). Even though consistency requires the selection of exactly one key node, this mismatch can be solved by confining the search space to all key paths that appear as paths from target nodes in the prefix tree. Even though the search predicate cannot always be computed on the much smaller prefix tree without access to the original document, we provide sufficient conditions for when this is the case. The partial order is defined as the set inclusion relation defined on key paths for which the one-step specialization relation is the inverse of $\prec_1$. Once all consistent one-key paths are determined, as explained earlier, a functional dependency miner can be used to determine the corresponding XML key (e.g., [Bitton et al. 1989; Mannila and Räihä 1989, 1994]).

Contributions. To summarize, our contributions are as follows.

(1) We characterize the complexity of the consistency problem for XML keys with respect to an XSD for different classes of target and key paths (Theorem 4.6). As a basic building block and of independent interest, we study the complexity of cardinality estimation of those XPath fragments in the presence of a schema (an overview is given in Table II). Moreover, we study the complexity of analogous cardinality estimation problems for the same XPath fragments but considering strings instead of trees and DFAs instead of XSDs (results are summarized in Table I). In addition, we characterize the complexity of boundedness, satisfiability, universality, and implication of XML keys (Theorem 4.32) as well as equivalence of target paths (Theorem 4.46).
(2) We develop a novel key mining algorithm leveraging on those for the discovery of relational functional dependencies and on the framework of levelwise search by employing an optimal one-step specialization relation for which the search relation can be computed, if not completely, then at least partly on a prefix tree representation of the document (Section 5).
(3) We experimentally assess the effectiveness of the proposed algorithm on an extensive body of real-world XML data.

Outline. In Section 2, we discuss related work and in Section 3 we introduce the necessary definitions. In Section 4, we investigate the complexity of decision problems

concerning keys in the presence of XSDs. In Section 5, we discuss the XML key mining algorithm, while in Section 6 we experimentally validate our algorithm. We conclude in Section 7. Due to space limitations, some proofs are moved to an online appendix accessible in the ACM Digital Library.

2. RELATED WORK

XML keys. One of the first definitions of keys for XML was introduced by Buneman et al. [2002, 2003]. These keys are of the form $(Q, (Q', P))$, where Q is the context path, Q' the target path, and P a set of key paths. Although the W3C definition of keys was largely inspired by this work, there are some important differences. First, Buneman et al.'s keys allow more expressive target and key paths by allowing several occurrences of the descendant operator. Context paths, however, are less expressive since W3C keys allow the context to be defined by an arbitrary Deterministic Finite state Automaton (DFA), while Buneman et al.'s keys limit themselves to path expressions. Furthermore, Buneman et al.'s key paths are allowed to select several nodes whereas W3C keys paths are restricted to select precisely one node. We stress that, in this article, we follow the W3C specification for the definition of keys. As is the case for the relational model, much is known about the complexity of key inference for Buneman et al.'s keys [Buneman et al. 2002, 2003; Hartmann and Link 2009]. Unfortunately, these results do not carry over to W3C keys as the latter are defined with respest to an XML schema but the former are not.

Decision problems in the presence of a schema. A number of consistency problems of XML keys with respest to a DTD have been considered by Fan and Libkin [2002]. They have shown, for instance, that key implication in the presence of a DTD is decidable in polynomial time. The keys that they consider, however, are much simpler than the W3C keys considered in the present article. Basically, a key in their setting is determined by an element name and a number of attributes. Their model is subsumed by ours since each such key can be defined by an XML key and every DTD can be represented by an XSD. We point out that Fan and Libkin [2002] provide many more results on the interplay between keys, foreign keys, inclusion dependencies, and DTDs. Arenas et al. [2002] discuss satisfiability[3] of XML keys with respect to a DTD. The result most relevant to the present article is NP-hardness of satisfiability with respect to a nonrecursive DTD and for keys with only one key path. We show that the problem becomes hard for EXPTIME in the presence of XSDs.

XML constraint mining. The automatic discovery of Buneman et al.'s keys from XML data has previously been considered by a number of researchers. Grahne and Zhu [2002] considered mining of approximate keys and proposed an Apriori-style algorithm which uses the inference rules of Buneman et al. [2003] for optimization. Necaský and Mlýnková [2009] ignore the XML data but present an approach to infer keys and foreign keys from element/element joins in XQuery logs. Fajt et al. [2011] consider the inference of keys and foreign keys building further on algorithms for the relational model. The algorithms mentioned earlier cannot be used for W3C keys since they do not take the presence of XSDs into account and keys are not required to be consistent. Yu and Jagadish [2008] consider discovery of functional dependencies (FDs) for XML. Similar to Buneman et al.'s keys, the considered FDs have paths that can select multiple data elements, and contexts are defined using a selector expression as opposed to using a DFA. For these reasons, W3C keys cannot be encoded as a special case of FDs. Barbosa and Mendelzon [2003] proposed algorithms to find ID and IDREFs attributes in XML documents. They show that the natural decision problem associated to this

[3]We note that satisfiability is called consistency in Arenas et al. [2002].

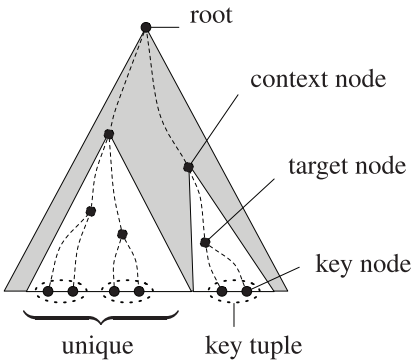


Fig. 2. Schematic representation of a key.

discovery problem is NP-complete, and present a heuristic algorithm. Abiteboul et al. [2012] consider probabilistic generators for XML collections in the presence of integrity constraints but do not consider the mining of such constraints.

The present article expands upon the conference version [Arenas et al. 2013] by providing all proofs; no proofs were present in Arenas et al. [2013]. The obtained results are nontrivial and use a variety of advanced techniques including tiling games and finite automata of bounded degree of ambiguity, both in the string and in the tree domain.

3. DEFINITIONS

In this section, we introduce the required definitions concerning trees, XSDs, and XML keys, and formally define the XML key mining problem. The correspondence between our definition of XML keys and the W3C definition is discussed in Section 3.3.

For a finite set R, we denote by $|R|$ the cardinality of R.

3.1. Trees and XML

As is standard, we represent XML documents by means of labeled trees. Formally, for a set S, an *S tree* is a pair (t, lab_t), where t is a finite tree and lab_t maps each node of t to an element in S. To reduce notation, we identify each tree simply by t and leave lab_t implied. We assume the reader to be familiar with standard common terminology on trees like *child*, *parent*, *root*, *leaf*, and so on. For a node v, we denote by $\text{anc-string}_t(v)$ the string formed by the labels on the unique path from t's root to (and including) v, called the *ancestor string* of v. By $\text{child-string}_t(v)$, we denote the string obtained by concatenating the labels of the children of v. If v is a leaf, then $\text{child-string}_t(v)$ is the empty string, denoted by ε. Here, we assume trees are sibling ordered. We fix a finite set of element names Σ and an infinite set **Data** of data elements. An *XML tree* is a $(\Sigma \cup \textbf{Data})$ tree where non-leaf nodes are labeled with Σ and leaf nodes are labeled with elements from $(\Sigma \cup \textbf{Data})$. As the XSD specification does not allow mixed content models for fields in keys [W3C 2004], we ignore "mixed" content models altogether to simplify presentation and assume that, when a node is labeled with a **Data** element, it is the only child of its parent. We then denote by $\text{value}_t(v)$ the **Data** label of v's unique child when it exists; otherwise we define $\text{value}_t(v) = \perp$, with $\perp$ a special symbol not in **Data**. When $\text{value}_t(v) \in \textbf{Data}$, we also say that v is a **Data** node.

Example 3.1. Figure 1 displays an XML tree t. In this tree, $\text{anc-string}_t(b_1) =$ `bookshop order items book`, and also $\text{child-string}_t(b_1) =$ `title year price quantity`.

Furthermore, every node labeled `id`, `person`, `address`, `title`, `year`, `price`, or `quantity` is a **Data** node, while, for instance, b_1 is not.

3.2. XSDs

XML keys are defined within the scope of an XSD. We make use of the DFA-based characterization of XSDs introduced by Martens et al. [2007]. An *XSD* is a pair $X = (A, \lambda)$ where $A = (\text{Types}, \Sigma \cup \{\texttt{data}\}, \delta, q_0)$ is a Deterministic Finite Automaton (or DFA for short) without final states (called the type automaton) and λ is a mapping from Types to deterministic[4] regular expressions over the alphabet $\Sigma \cup \{\texttt{data}\}$. Here, Types is the set of states, `data` is a special symbol (not in Σ) which will serve as a placeholder for **Data** elements, $\delta : \text{Types} \times \Sigma \cup \{\texttt{data}\} \to \text{Types}$ is the (partial) transition function, and $q_0 \in \text{Types}$ is the initial state. Additionally, the labels of transitions leaving q should be precisely the symbols in $\lambda(q)$. That is, for every $q \in \text{Types}$, $Out(q) = Symb(\lambda(q))$, where $Out(q) = \{\sigma \in \Sigma \cup \{\texttt{data}\} \mid \delta(q, \sigma) \text{ is defined}\}$ and $Symb(r)$ consists of all $(\Sigma \cup \{\texttt{data}\})$ symbols in regular expression r.

A *context* $c = (\sigma, q)$ is a pair in $\Sigma \times \text{Types}$. By $\text{CNodes}_t(c)$, we denote all nodes v of t for which $\text{lab}_t(v) = \sigma$ and A halts in state q when started in q_0 on the string $\text{anc-string}_t(v)$. Let $\mathcal{L}(r)$ denote the language defined by the regular expression r. We say that the tree t *adheres to* X, if for every context $c = (\sigma, q)$ and every v in $\text{CNodes}_t(c)$ one of the following holds:

—$\text{value}_t(v) \in$ **Data** and $\texttt{data} \in \mathcal{L}(\lambda(q))$; or
—$\text{value}_t(v) = \perp$ and $\text{child-string}_t(v) \in \mathcal{L}(\lambda(q))$.

Intuitively, A determines the vertical context of a node v by the state q it reaches in processing $\text{anc-string}_t(v)$. When v is a **Data** node, the content model specified by q, that is $\lambda(q)$, should contain the placeholder `data`. Otherwise, when v is not a **Data** node, $\text{child-string}_t(v)$ should satisfy the content model $\lambda(q)$. Recall that we do not allow for mixed content models. We stress that this DFA-based characterization of XSDs corresponds precisely to the more traditional abstraction in terms of single-type grammars [Martens et al. 2006; Murata et al. 2005]. We let $\mathcal{L}(X)$ denote the set of all trees adhering to XSD X. We assume that an XSD always defines trees with the same root label. In this way, the root is always assigned the same context, also referred to as the root context c_{root}.

Example 3.2. Let $X_{\text{bookshop}} = (A, \lambda)$ be the XSD where A is given in Figure 3 and λ is defined as follows.

$$
\begin{aligned}
q_0 &\mapsto \texttt{bookshop} \\
q_{\text{bookshop}} &\mapsto \texttt{order}^+ \\
q_{\text{order}} &\mapsto \texttt{id person address items}^+ \\
q_{\text{items}} &\mapsto \texttt{book}^+ \\
q_{\text{book}} &\mapsto \texttt{title year? price quantity}
\end{aligned}
$$

For all other types q, $\lambda(q) = \texttt{data}$. Here, r^+ and $r?$ are the usual abbreviations for rr^* and $r + \varepsilon$, where r is a regular expression.

Then tree t in Figure 1 adheres to X_{bookshop}. Moreover, $b_1 \in \text{CNodes}_t(\text{book}, q_{\text{book}})$ and $\text{child string}_t(b_1) \in \mathcal{L}(\lambda(q_{\text{book}}))$.

[4]Also referred to as 1-unambiguous regular expressions [Brüggemann-Klein and Wood 1998].

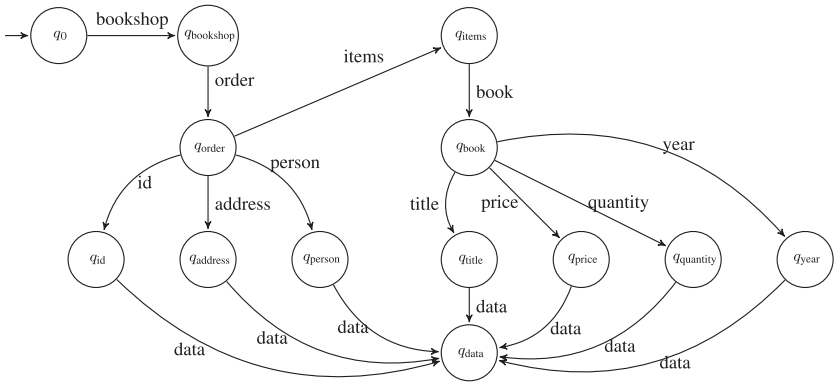


Fig. 3. The type automaton of X_{bookshop}.

3.3. XML Keys

A *selector expression* is a restricted XPath expression of the form $./l_1/l_2/\ldots/l_k$ (starting with the child axis) or $.//l_1/l_2/\ldots/l_k$ (starting with the descendant axis), where $k \geq 1$, and $l_1, \ldots, l_k$ are element names or the wildcard symbol "*". A string $w = w_1 \cdots w_k$, where each w_i is an element name, is said to *match* $./l_1/l_2/\ldots/l_k$ when $w_i = l_i$ or $l_i = *$ for each i. For selector expressions starting with the descendant axis, we say that w matches $.//l_1/l_2/\ldots/l_k$ if a suffix of w matches $./l_1/l_2/\ldots/l_k$. For a tree t, a node v of t, and a selector expression τ, the set $\tau(t, v)$ contains all nodes v' such that v' is a descendant of v and the path of labels from v (but excluding the label of v) to (and including) v' matches τ. A *disjunction of selector expressions* is of the form $\tau = \tau_1 \mid \cdots \mid \tau_m$, where each τ_i is a selector expression. In this case, $\tau(t, v)$ is defined as the union of all $\tau_i(t, v)$. When v is the root of the document, we simply write $\tau(t)$ for $\tau(t, v)$. We denote by $\mathcal{SE}$ and $\mathcal{DSE}$ the class of selector expressions and disjunctions of selector expressions, respectively. In proofs, we sometimes omit the leading dot in selector expressions and simply write $/l_1/l_2/\ldots/l_k$ (or even $l_1/l_2/\ldots/l_k$) or $//l_1/l_2/\ldots/l_k$ to denote $./l_1/l_2/\ldots/l_k$ and $.//l_1/l_2/\ldots/l_k$, respectively.

Definition 3.3. An *XML key*, defined with respect to an XSD X, is a tuple $\phi = (c, \tau, P)$, where: *(i)* c is a context in X, *(ii)* $\tau \in \mathcal{DSE}$ is called the *target path*, and *(iii)* P is an ordered sequence of expressions in $\mathcal{DSE}$ called *key paths*.

To emphasize that ϕ is defined with respect to X, we sometimes write a key simply as a pair (ϕ, X).

We stress that the definition of XML keys given before corresponds to the definition of keys in XML schema [W3C 2004]. In particular, the context is given implicitly by declaring a key inside an element and an element has a label and a certain type. Target paths are called *selector paths* [W3C 2004, Section 3.11.6.2] and key paths are called *fields*. They obey the same grammar as used here with the difference that we do not make use of attributes but require key paths to select data nodes.

The semantics of an XML key is as follows. The context c defines a set of context nodes that divides the document into separate (but not necessarily disjoint) parts. Specifically, each node in $\text{CNodes}_t(c) = \{v_1, \ldots, v_n\}$ can be considered as the root of a separate tree. For each of these trees, that is, for each $i \in \{1, \ldots, n\}$, every node in $\tau(t, v_i)$ should uniquely define a record. Such a record is determined by the key paths in $P = (p_1, \ldots, p_k)$. That is, each v in $\tau(t, v_i)$ defines the record $[\text{value}_t(u_1), \ldots, \text{value}_t(u_k)]$,

denoted by $\text{record}_P(t, v)$, where $p_j(t, v) = \{u_j\}$ for all $j \in \{1, \ldots, k\}$. We graphically illustrate the preceding in Figure 2.

Note that $p_j(t, v)$ might select more than one node, a node u for which $\text{value}_t(u)$ is undefined, or might select nothing; these three cases are disallowed by the XML schema specification.

Definition 3.4. A key $\phi = (c, \tau, P)$ *qualifies* in a document t if, for every $v \in \text{CNodes}_t(c)$, every $u \in \tau(t, v)$, and every $p \in P$, $p(t, u)$ is a singleton containing a **Data** node.

Finally, following the W3C specification, we define satisfaction of an XML key with respect to a document.

Definition 3.5. An XML tree t *satisfies* a key $\phi = (c, \tau, P)$ or a key is *valid* with respect to t, denoted by $t \models \phi$, iff: *(i)* ϕ qualifies in t and *(ii)* for every node v in $\text{CNodes}_t(c)$, $\text{record}_P(t, u) \neq \text{record}_P(t, u')$, for every two different nodes u and u' in $\tau(t, v)$.

Notice there can be two causes for a key to be invalid: (i) the key does not qualify in the document and actually is ill defined with respect to the document, or (ii) the data values in the document invalidate the key. The first cause can be seen as structural invalidation, while the second is semantical and more informative.

In this article, we are interested in inferring keys that *always* qualify to a document satisfying the schema. We call such keys consistent. In Section 4, we show that consistency can be decided efficiently for target and key paths in $\mathcal{SE}$, and is intractable otherwise.

Definition 3.6. A key is *consistent* with respect to a schema if the key qualifies in every document adhering to the schema.

Example 3.7. Consider the key ϕ from Example 1.1. Then ϕ is valid with respect to the tree in Figure 1 but ϕ is not consistent with respect to X_{bookshop}. Indeed, X_{bookshop} defines the "year" element of a "book" element to be optional.

3.4. XML Key Mining

Given an XML document t adhering to a given XSD, we want to derive all supported XML keys ϕ that are valid with respect to t.[5] We define the support of a key as the quantity measuring the number of nodes captured by the key. Define $\text{TNodes}_t(\phi)$ as the set of target nodes selected by $\phi = (c, \tau, P)$ on t. That is,

$$\text{TNodes}_t(\phi) = \bigcup_{v \in \text{CNodes}_t(c)} \tau(t, v).$$

Then, following Grahne and Zhu [2002], we define the *support* of ϕ on t to be the total number of selected target nodes: $\text{supp}(\phi, t) = |\text{TNodes}_t(\phi)|$. Since this support only depends on the context c and the target path τ of ϕ, we also write $\text{supp}(c, \tau, t)$ for $\text{supp}(\phi, t)$.

Example 3.8. Consider the document t depicted in Figure 1 and the key ϕ from Example 1.1. The context nodes are those that match the context $(\text{order}, q_{\texttt{order}})$. There are two such nodes in t, both are labeled order and are direct children of the root node. The target nodes are those that can be selected from the two context nodes, using a

[5]Without loss of generality and to simplify presentation, we restrict attention to a single document as multiple XML documents can always be combined into one by introducing a common root.

path that matches `.//book`. There are three such target nodes, two for the first and one for the second context node. Therefore, the support of ϕ on t equals 3, that is, $\text{supp}(\phi, t) = 3$.

We are now ready to define the problem central to this article.

Definition 3.9. (*XML Key Mining Problem*). Given an XSD X, an XML document t adhering to X, and a minimum support threshold N, the *XML key mining problem* consists of finding all keys ϕ consistent with X such that $t \models \phi$ and $\text{supp}(\phi, t) > N$.

This is only the core definition of the XML key mining problem. We will discuss some quality requirements in the next section.

4. DECISION PROBLEMS

A basic problem in data mining is the abundance of found patterns. In this section, we address a number of fundamental decision problems relevant to identifying low-quality keys that can then be removed from the output of the key mining algorithm. In Section 4.1, we provide some additional definitions and properties. Consistency is a problem fundamental to this article. We therefore discuss testing for consistency in a separate section (Section 4.2) where we also give an overview of all related results (as summarized in Table I and Table II). These results are then formally proved in the following two sections. We start by considering the analogous case on strings (in Section 4.3), as these results are interesting in their own right and because they provide a starting ground for the more general case of trees which is treated in Section 4.4. The main result here is that consistency as defined in Definition 3.6 is tractable when target paths and key paths are restricted to selector expressions. Finally, in Section 4.5, we consider general decision problems. We obtain that universality and boundedness are tractable and show that testing for satisfiability and implication of keys is EXPTIME-hard even when disallowing disjunction, which complicates the inference of minimal keys.

4.1. Preliminaries

We assume the reader is familiar with basic concepts from formal language theory like nondeterministic finite automata (NFA), regular expressions, pumping lemma, product construction, complement construction for NFAs and DFAs, and refer the interested reader to a textbook (e.g., [Hopcroft et al. 2003]).

We introduce some definitions. Let $A = (\Sigma, Q, q_0, \delta, F)$ be an NFA and $s = \sigma_0\sigma_1 \cdots \sigma_{n-1}$ a string in Σ^*. A run of A on s is defined as a function $\rho : \{0, 1, \ldots, n\} \to Q$ such that:

—$\rho(0) = q_0$; and
—for every $i \in \{1, 2, \ldots, n-1\}$, $(\rho(i), \sigma_i, \rho(i+1)) \in \delta$.

An accepting run of A on s is a run ρ such that $\rho(n) \in F$. It is clear from the previous definition that $\mathcal{L}(A)$ is the set of strings s in Σ^* for which there exists an accepting run of A with input s. Of course, A is a DFA when $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. For a language $L \subseteq \Sigma^*$, we denote by $\overline{L} = \Sigma^* \setminus L$ the complement of L. Similarly, for an NFA A, we denote by $\overline{A}$ the automaton accepting $\overline{\mathcal{L}(A)}$. The size of A, denoted $|A|$, is defined as $|Q| + \Sigma_{q,a}|\delta(q, a)|$.

The following theorem states some well-known complexity results for finite automata that we will use in the sequel.

THEOREM 4.1. *Let A and A' be two NFAs.*

(1) *Deciding whether $\mathcal{L}(A) = \emptyset$ is in* PTIME *[Hopcroft et al. 2003].*
(2) *Deciding whether $\mathcal{L}(A) = \mathcal{L}(A')$ is* PSPACE*-complete [Stockmeyer and Meyer 1973].*

(3) *An NFA accepting the language $\mathcal{L}(A) \cap \mathcal{L}(A')$ can be constructed in time polynomial in the sizes of A and A′ [Hopcroft et al. 2003].*
(4) *When A is a DFA, a DFA accepting $\overline{\mathcal{L}(A)}$ can be constructed in time polynomial in the size of A [Hopcroft et al. 2003].*

For the remainder, it will be convenient to work with *trimmed* XSDs. Intuitively, we call an XSD *trimmed* if it does not contain useless states or transitions. The formal definition is as follows.

Definition 4.2. Let X be an XSD and let q_0 be its initial state. We call a context $c = (\sigma, q)$ *well formed* (with respect to X) if there exists a type q' in X such that $\delta(q', \sigma) = q$, where δ is the transition function of the type automaton of X. For type q in X, let X^q be the XSD obtained from X by replacing its start state by q. Then X is *trimmed* if: (1) $\mathcal{L}(X^q) \neq \emptyset$ for all $q \neq q_0$, and (2) there is a tree $t \in \mathcal{L}(X)$ with $\text{CNodes}_t(\sigma, q) \neq \emptyset$, for every well-formed context (σ, q) of X.

That is, there can be no state (except possibly the initial state q_0) that defines the empty tree language and every well-formed context in A should be realized in at least one tree in $\mathcal{L}(X)$.

The next lemma says that we can restrict attention to trimmed XSDs. A proof can be found in Appendix A.

LEMMA 4.3. *Every XSD can be converted in* PTIME *to a trimmed XSD defining the same language.*

The following lemma states that in trimmed XSDs we can always find XML trees adhering to the XSD that realize a given desired context in a node with a given desired child string. The proof is given in Appendix A.

LEMMA 4.4. *Let $X = (A, \lambda)$ be a trimmed XSD and let $c = (\sigma, q)$ a well-formed context of X. Let $w \in \Sigma^*$ be a word in $\mathcal{L}(\lambda(q))$. Then there exists $t \in \mathcal{L}(X)$ and $v \in \text{CNodes}_t(c)$ such that* child $\text{string}_t(v) = w$.

4.2. Consistency

As detailed in Section 3.3, the W3C specification requires keys to be consistent. We therefore define CONSISTENCY as the problem to decide whether ϕ is consistent with respect to X, given a key ϕ and an XSD X. In this section, we show that CONSISTENCY is in fact solvable in PTIME when patterns in keys are restricted to $\mathcal{SE}$. The proof of this result is the most technical result of the article. Actually, the PTIME result is also surprising since a minor variation of consistency is known to be EXPTIME-hard, as we explain next.

Consistency requires that, on every document adhering to X, every key path must select precisely *one* data node for every target node. This is related to deciding whether an XPath selector expression selects at least and at most a given number of nodes, on every document satisfying a given XSD. Indeed, define $\forall^{\bullet k}_{\text{tree}}$ with $k \in \mathbb{N}$ and $\bullet \in \{<, =, >\}$ to be the problem of deciding, given an XSD X and a selector expression p, whether it holds that $|p(t)| \bullet k$, for every $t \in \mathcal{L}(X)$. We show in Lemma 4.5 that CONSISTENCY can be easily reduced to $\forall^{=1}_{\text{tree}}$. Although Björklund et al. [2013] showed that $\forall^{>k}_{\text{tree}}$ is EXPTIME-complete, we obtain shortly that $\forall^{=k}_{\text{tree}}$ can in fact be solved in polynomial time through an intricate translation to the equivalence test for unambiguous tree automata [Seidl 1990].

In order to obtain these complexity results, we first focus on analogous problems considering DFAs instead of XSDs, and strings instead of trees. We define the problem $\forall^{\bullet k}_{\text{string}}$ with $k \in \mathbb{N}$ and $\bullet \in \{<, =, >\}$ as the problem of deciding, given a DFA A and a

Table I. Complexity of $\forall^{\bullet k,\mathcal{P}}_{\text{string}}$

$\mathcal{P}$	$\forall^{>k,\mathcal{P}}_{\text{string}}$	$\forall^{<k,\mathcal{P}}_{\text{string}}$	$\forall^{=k,\mathcal{P}}_{\text{string}}$
$\mathcal{RE}$	PSPACE-complete	in PTIME	PSPACE-complete
$\mathcal{CSE}$	PSPACE-complete	in PTIME	PSPACE-complete
$\mathcal{DSE}$	PSPACE-complete	in PTIME	CONP-complete
$\mathcal{SE}$	PSPACE-complete	in PTIME	in PTIME
$\mathcal{SE}^*$	in PTIME	in PTIME	in PTIME
$\mathcal{SE}^{//}$	in PTIME	in PTIME	in PTIME

Table II. Complexity of $\forall^{\bullet k,\mathcal{P}}_{\text{tree}}$

$\mathcal{P}$	$\forall^{>k,\mathcal{P}}_{\text{tree}}$	$\forall^{<k,\mathcal{P}}_{\text{tree}}$	$\forall^{=k,\mathcal{P}}_{\text{tree}}$
$\mathcal{RE}$	EXPTIME-complete	in PTIME	in EXPTIME PSPACE-hard ($k \geq 1$)
$\mathcal{CSE}$	EXPTIME-complete	in PTIME	in EXPTIME PSPACE-hard ($k \geq 1$)
$\mathcal{DSE}$	EXPTIME-complete	in PTIME	in EXPTIME CONP-hard ($k \geq 1$)
$\mathcal{SE}$	EXPTIME-complete	in PTIME	in PTIME
$\mathcal{SE}^*$	in EXPTIME	in PTIME	in PTIME
$\mathcal{SE}^{//}$	in PTIME	in PTIME	in PTIME

selector expression p, whether it holds that $|p(s)| \bullet k$, for every $s \in \mathcal{L}(A)$. The results over strings are important for our investigation not only because they can be used to obtain lower bounds for the complexity of the problems $\forall^{\bullet k}_{\text{tree}}$, but also because the extension of some of the techniques developed to prove them played a key role in pinpointing the complexity of some of the problems $\forall^{\bullet k}_{\text{tree}}$, most notably in the problem $\forall^{=k}_{\text{tree}}$ when target and key paths are restricted to $\mathcal{SE}$.

Because of its relevance to cardinality estimation of XPath result sets, we extend the problems $\forall^{\bullet k}_{\text{tree}}$ and $\forall^{\bullet k}_{\text{string}}$ by restricting target and key paths to different fragments of $\mathcal{SE}$. To obtain a more complete picture, we also consider the class of all regular expressions, denoted by $\mathcal{RE}$. For a regular expression r and a tree t, $r(t)$ then selects all nodes whose ancestor string[6] matches r. Furthermore, denote by $\mathcal{SE}^{//}$ and $\mathcal{SE}^*$ the set of all selector expressions without descendant and wildcard, respectively (recall that $\mathcal{SE}$ denotes the set of all selector expressions). In addition, we consider concatenations of selector expressions, as they are a natural extension to selector expressions. Denote by $\mathcal{CSE}$ the class of all expressions of the form $p_1 p_2 \cdots p_n$, where each p_i $(1 \leq i \leq n)$ is a selector expression (this allows for arbitrarily many descendant axes). The semantics for this fragment is inherited from the semantics for regular expressions.

For a class of patterns $\mathcal{P} \in \{\mathcal{RE}, \mathcal{DSE}, \mathcal{CSE}, \mathcal{SE}, \mathcal{SE}^{//}, \mathcal{SE}^*\}$, we denote by $\forall^{\bullet k,\mathcal{P}}_{\text{tree}}$ the problem $\forall^{\bullet k}_{\text{tree}}$ where expressions are restricted to the class $\mathcal{P}$. Analogously, $\forall^{\bullet k,\mathcal{P}}_{\text{string}}$ is the problem $\forall^{\bullet k}_{\text{string}}$ where expressions are restricted to the class $\mathcal{P}$. In the next two sections we treat the complexity of these decision problems which are summarized in Table I and Table II. Note that these results also provide an upper bound for testing whether the number of nodes selected by a selector expression always lies within a fixed interval $[k, k']$.

Before proving the results stated in Tables I and II, we explain how to apply them to obtain complexity results for the consistency problem. We introduce the following

[6]Defined in Section 3.2 as the string formed by the labels on the path from the root to the considered node.

definition. Let $k \in \mathbb{N}$, $\bullet \in \{<, =, >\}$, and $\mathcal{R}, \mathcal{S}$ be two pattern languages. We denote by $\forall_{\text{key}}^{\bullet k,\mathcal{R},\mathcal{S}}$ the problem to decide whether, for a given XSD X and a key $\phi = (c, \tau, (p))$ with $\tau \in \mathcal{R}$ and $p \in \mathcal{S}$, it holds that $|p(t, u)| \bullet k$ for every $t \in \mathcal{L}(X)$, every node v in $\text{CNodes}_t(c)$, and for every target node u in $\tau(t, v)$.

The following lemma now allows to transfer upper and lower bounds from Table I and Table II.

LEMMA 4.5. *Let $k \in \mathbb{N}$, let $\bullet \in \{<, >, =\}$, and let $\mathcal{P} \in \{\mathcal{RE}, \mathcal{DSE}, \mathcal{SE}, \mathcal{SE}^{//}, \mathcal{SE}^*\}$. Then:*

(1) $\forall_{\text{key}}^{\bullet k,\mathcal{RE},\mathcal{P}}$ is polynomial-time reducible to $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$; and
(2) $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ is polynomial-time reducible to $\forall_{\text{key}}^{\bullet k,\mathcal{SE},\mathcal{P}}$.

PROOF. (1) Let $\phi = (c, \tau, (p))$ be a key and let $X = (A = (Q_X, \Sigma, \delta_X, q_0^X), \lambda)$ be an XSD. Now, define $Q_{\tau,c}$ as the maximal subset of states $q \in Q_X$ for which there is a tree $t \in \mathcal{L}(X)$, a node v in $\text{CNodes}_t(c)$, and a node u in $\tau(t, v)$ such that A when executed on the ancestor string of u ends in state q. This means that q is a state of the XSD that can be reached when evaluating τ from a node in $\text{CNodes}_t(c)$, for some $t \in \mathcal{L}(X)$. Denote by X^q the XSD X with start type q. Then $\bigcup_{q \in Q_{\tau,c}} \mathcal{L}(X^q)$ is the set of trees on which the number of matches of p needs to be tested. Therefore, (X, ϕ) is in $\forall_{\text{key}}^{\bullet k,\mathcal{RE},\mathcal{P}}$ iff (X^q, p) is in $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ for every $q \in Q_{\tau,c}$. It remains to explain how to compute $Q_{\tau,c}$. We can assume that X is trimmed (see Definition 4.2). We also assume that $c = (q_c, \sigma_c)$ is well formed. That is, there is a state q' such that $\delta(q', \sigma_c) = q_c$. Let $\tau \in \mathcal{RE}$ and let $A_\tau = (Q_\tau, \Sigma, \delta_\tau, q_0^\tau, F_\tau)$ be the DFA accepting $\mathcal{L}(\tau)$. Define $\Gamma_0 = \{(q_c, q_0^\tau)\}$ and $\Gamma_i = \{(\delta_X(q_1, \sigma), \delta_\tau(q_2, \sigma)) \mid (q_1, q_2) \in \Gamma_{i-1}, \sigma \in \Sigma\}$. Then, $Q_{\tau,c} = \{q \mid \exists (q, q') \in \Gamma_{|Q_X| \times |Q_\tau|}$ and $q' \in F_\tau\}$.

(2) Assume given (X, p). Let $\#_1$ be a new symbol and $X_\#$ be the XSD defining the set $\{\#_1((t)) \mid t \in \mathcal{L}(X)\}$. Then $(X, p) \in \forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ iff $(X_\#, (c_{\text{root}}, ./*, p))$ is in $\forall_{\text{key}}^{\bullet k,\mathcal{SE},\mathcal{P}}$ where c_{root} is the root context. □

We finally present the main result of this section. Given a class of patterns $\mathcal{P}$, we denote by CONSISTENCY($\mathcal{P}$) the problem CONSISTENCY restricted to keys using expressions in $\mathcal{P}$.

THEOREM 4.6.

(1) CONSISTENCY($\mathcal{SE}$) is in PTIME;
(2) CONSISTENCY($\mathcal{DSE}$) is coNP-hard and in EXPTIME; and
(3) CONSISTENCY($\mathcal{RE}$) is PSPACE-hard and in EXPTIME.

PROOF. Let X be an XSD and $\phi = (c, \tau, P)$ be a key. Then (X, ϕ) is consistent iff $(X, (c, \tau, (p)))$ is consistent for every $p \in P$. The results then follow by Lemma 4.5 and the corresponding results in Table II. □

Section 4.3 is devoted to proving the complexity results stated in Table I, while the proofs for trees mentioned in Table II can be found in Section 4.4.

4.3. Proofs for Results on Strings Mentioned in Table I

In this section, we prove the complexity of decision problems $\forall_{\text{string}}^{\bullet k,\mathcal{P}}$ with $k \in \mathbb{N}$, $\bullet \in \{<, =, >\}$ and $\mathcal{P} \in \{\mathcal{RE}, \mathcal{DSE}, \mathcal{CSE}, \mathcal{SE}, \mathcal{SE}^{//}, \mathcal{SE}^*\}$, which is summarized in Table I.

4.3.1. Proofs for $\forall_{\text{string}}^{>k,\mathcal{P}}$. We start by considering the problem $\forall_{\text{string}}^{>k,\mathcal{P}}$. The following lemma is used to pinpoint the exact complexity of this problem for selector expressions, disjunctions of selector expressions, concatenations of selector expressions, and regular expressions. The proof of the next lemma can be found in Appendix B.

LEMMA 4.7. *Let $k \geq 0$ be a constant. There exists a polynomial-time algorithm that, given an NFA A and a regular expression r, constructs an automaton A' accepting precisely all strings s in $\mathcal{L}(A)$ for which $|r(s)| \geq k$.*

We can now prove the first four results in Table I concerning the problem $\forall^{>k,\mathcal{P}}_{\text{string}}$. We gratefully acknowledge that the specific encoding of tiles and the selector expression used in the lower bound proof that follows is borrowed from the EXPTIME-hardness result in Björklund et al. [2013].

THEOREM 4.8. *Let $k \geq 0$ be a fixed constant. Then $\forall^{>k,\mathcal{SE}}_{\text{string}}$, $\forall^{>k,\mathcal{DSE}}_{\text{string}}$, $\forall^{>k,\mathcal{CSE}}_{\text{string}}$, and $\forall^{>k,\mathcal{RE}}_{\text{string}}$ are all* PSPACE*-complete.*

PROOF. For the upper bound it suffices to prove that $\forall^{>k,\mathcal{RE}}_{\text{string}}$ is in PSPACE. Thereto, let A be a DFA, let r be a regular expression, and let A' be the automaton accepting precisely all strings in $\mathcal{L}(A)$ for which r selects at least $k+1$ distinct positions (as given in Lemma 4.7). Then, $(A, r) \in \forall^{>k,\mathcal{RE}}_{\text{string}}$ if and only if $\mathcal{L}(A') = \mathcal{L}(A)$ which, by Theorem 4.1, is in PSPACE [Stockmeyer and Meyer 1973].

For the lower bound, it suffices to prove PSPACE-hardness for $\forall^{>k,\mathcal{SE}}_{\text{string}}$. We show that the complement of $\forall^{>0,\mathcal{SE}}_{\text{string}}$, denoted by $\overline{\forall^{>0,\mathcal{SE}}_{\text{string}}}$, is PSPACE-hard through a reduction from CORRIDOR TILING, a well-known PSPACE-complete problem [van Emde Boas 1997]. As PSPACE is closed under complement, $\forall^{>0,\mathcal{SE}}_{\text{string}}$ is PSPACE-hard as well. Afterwards, we will show that $\forall^{>k,\mathcal{SE}}_{\text{string}}$ is hard for every k.

An instance of CORRIDOR TILING is a tuple $T = (D, H, V, \bar{a}, \bar{b}, n)$, where D is a set of tiles, $H, V \subseteq D \times D$ are the horizontal and vertical constraints, $\bar{a} = (a_1, \ldots, a_n)$, $\bar{b} = (b_1, \ldots, b_n)$ are n-tuples of tiles, and n is a natural number. The decision problem consists of deciding whether T has a valid tiling of a board with m rows and n columns for some $m \in \mathbb{N}$. A tiling is said to be valid if:

—the first row is $\bar{a}$;
—the last row is $\bar{b}$;
—For each pair of tiles (x, y), if (x, y) are horizontally adjacent then $(x, y) \in H$; and
—For each pair of tiles (x, y), if (x, y) are vertically adjacent then $(x, y) \in V$.

Let $T = (D, H, V, \bar{a}, \bar{b}, n)$ be a tiling instance. We construct a DFA A and a selector expression p such that there is a valid tiling for T if and only if there is a string s in $\mathcal{L}(A)$ such that $p(s) = \emptyset$.

The idea is to encode every tiling for T as a string. The encoding of a tiling will be the concatenation of the encodings of its tiles, which are defined next. Assume without loss of generality that $D = \{1, 2, \ldots, k\}$ and let $v_{i,j}$ be defined as

$$v_{i,j} = \begin{cases} t & \text{if } (i, j) \in V, \text{ and} \\ f & \text{if } (i, j) \notin V. \end{cases}$$

The encoding of $i \in D$ is defined as

$$e_i = \sigma_i 0^{i-1} 1 0^{k-i} v_{i,1} v_{i,2} \cdots v_{i,k},$$

where σ_i is a new alphabet symbol for every i. So, here σ_i denotes that the next tile is i, $0^{i-1}10^{k-i}$ denotes tile number i in unary, and $v_{i,1}v_{i,2}\cdots v_{i,k}$ encodes which tiles satisfy the vertical constraints when placed on top of the current tile in the row.

For example, if we have four tiles and the vertical constraints are $V = \{(1, 2), (1, 4), (2, 4)\}$ then the encodings are as follows.

$$e_1 = \sigma_1 1000 ftft \qquad e_2 = \sigma_2 0100 fftt$$
$$e_3 = \sigma_3 0010 ffff \qquad e_4 = \sigma_4 0001 ffff$$

Notice that $|e_i| = 2k + 1$ for every i. A tiling $(i_1, i_2, \ldots, i_{n \cdot m})$ (where $i_1, i_2, \ldots, i_n$ is the first row of the tiling, $i_{n+1}, i_{n+2}, \ldots, i_{2n}$ the second row, and so on) has as encoding the string $e_{i_1} e_{i_2} \ldots e_{i_{nm}}$.

The DFA A is constructed such that it accepts precisely all encodings of tilings satisfying the following requirements: (1) the first row is $\bar{a}$, (2) the final row is $\bar{b}$, (3) no horizontal constraint is violated, (4) the input is a sequence of encodings e_i, and (5) every row has the correct length n. Clearly, A can be constructed as the intersection of five automata, each one taking care of one of the requirements just given. Note that each such requirement can be checked by a simple linear scan of the input string using a number of states linear in the size of T.

It remains to verify the vertical constraints. Next, we create the selector expression, which is defined as

$$p = //f \underbrace{/*/*/\cdots/*/}_{n\cdot(2k+1)-k-1 \text{ times}} 1.$$

It is important to understand that, in the encoding of a tiling, $n \cdot (2k + 1) - k$ positions after an f there is a 1 if and only if the tiling violates a vertical constraint in the current column. For example, consider the tiling instance $(D, H, V, \bar{a}, \bar{b}, n)$ with $D = \{1, 2, 3, 4\}$, $H = \{(1, 2), (3, 4)\}$, $V = \{(1, 1), (2, 2), (1, 4), (2, 4)\}$, $\bar{a} = (1, 2)$, $\bar{b} = (3, 4)$, and $n = 2$. Then, $k = 4$ and $n \cdot (2k + 1) = 18$ and

$$p = //f \underbrace{/*/*/*/*/*/*/*/*/*/*/*/*/*/}_{n \cdot (2k+1) - k - 1 = 13 \text{ times}} 1.$$

Now, consider the string

$$e_1 e_2 e_3 e_4 = \sigma_1 1000 tff \overbrace{t\sigma_2 0100 ftft\sigma_3 0010}^{13 \text{ positions}} ffff\sigma_4 0001 ffff$$

encoding a tiling consisting of two rows. The first row consists of tiles 1 and 2, while the second row contains the tiles 3 and 4. The tiling begins with $\bar{a}$, ends with $\bar{b}$, and respects every horizontal constraint. Thus, by definition of the DFA A in the reduction, we have that $e_1 e_2 e_3 e_4 \in \mathcal{L}(A)$. The tiling is not valid as the first column contains the tiles 1 and 3 but the pair $(1, 3)$ is not in V. The expression p now matches at the second symbol f (which encodes that $(1, 3) \notin V$) and the symbol 1 encoding that tile 3 is the first tile in row 2.

It remains to argue the correctness of the reduction. Suppose there is a valid tiling for T. Then the encoding for this valid tiling is a string in $\mathcal{L}(A)$ for which p does not select any position, that is, $(A, p) \in \overline{\forall^{>0,\mathcal{SE}}_{\text{string}}}$. Conversely, if $(A, p) \in \overline{\forall^{>0,\mathcal{SE}}_{\text{string}}}$, then there is a string accepted by A for which p selects no position. By definition of A and p, this string encodes a valid tiling for T.

To finish the proof, we show that $\forall^{>k,\mathcal{SE}}_{\text{string}}$ is PSPACE-hard for every k. To this end, we show that $\forall^{>0,\mathcal{SE}}_{\text{string}}$ can be reduced in polynomial time to $\forall^{>k,\mathcal{SE}}_{\text{string}}$. Given the preceding reduction, it suffices to only consider selector expressions of the form $//a/*/*/\cdots/*/b$ with $a \neq b$. The whole construction is based on padding. Let A be a DFA and p a selector expression of the form $//a/\underbrace{*/*/\cdots/*}_{n \text{ times}}/b$. For a string s, define $ext(s) = (a \underbrace{XX\cdots XX}_{n \text{ times}} b)^k s$, where

X is a fresh symbol. Let $k > 0$. Now, define A' as the DFA accepting the language $\{ext(s) \mid s \in \mathcal{L}(A)\}$. Note that A' can be constructed in time polynomial in $|A| + n$. It now readily follows that $(A, p) \in \forall^{>0,\mathcal{SE}}_{\text{string}}$ iff $(A', p) \in \forall^{>k,\mathcal{SE}}_{\text{string}}$. □

The next theorem states that both $\forall^{>k,\mathcal{SE}^*}_{\text{string}}$ and $\forall^{>k,\mathcal{SE}^{//}}_{\text{string}}$ are tractable. A proof can be found in Appendix B.

THEOREM 4.9. *For every $k \geq 0$:*

(1) $\forall^{>k,\mathcal{SE}^{//}}_{\text{string}}$ is in PTIME*; and*
(2) $\forall^{>k,\mathcal{SE}^*}_{\text{string}}$ is in PTIME*.*

4.3.2. Proofs for $\forall^{<k,\mathcal{P}}_{\text{string}}$. We continue our study by considering the problem $\forall^{<k,\mathcal{P}}_{\text{string}}$. All results in Table I concerning $\forall^{<k,\mathcal{P}}_{\text{string}}$ follow from the tractability of $\forall^{<k,\mathcal{RE}}_{\text{string}}$.

THEOREM 4.10. *For every $k \geq 0$, $\forall^{<k,\mathcal{RE}}_{\text{string}}$ is in* PTIME*.*

PROOF. If $k = 0$, then the property trivially holds. For $k \geq 1$, $\forall^{<k,\mathcal{RE}}_{\text{string}}$ reduces to emptiness of NFAs. Indeed, for a DFA A and a regular expression r, let A' be the automaton accepting every string in $\mathcal{L}(A)$ for which r selects at least k distinct positions, as shown in Lemma 4.7. Then $(A, r) \in \forall^{<k,\mathcal{RE}}_{\text{string}}$ iff $\mathcal{L}(A') = \emptyset$. □

4.3.3. Proofs for $\forall^{=k,\mathcal{P}}_{\text{string}}$. Finally, we study the complexity of $\forall^{=k,\mathcal{P}}_{\text{string}}$. We start by proving that this problem is, in general, PSPACE-complete for concatenations of selector expressions and regular expressions. Tractability of $\forall^{=k,\mathcal{SE}}_{\text{string}}$ is handled at the end of this section.

THEOREM 4.11. *$\forall^{=0,\mathcal{RE}}_{\text{string}}$ and $\forall^{=0,\mathcal{CSE}}_{\text{string}}$ are in* PTIME*. Moreover, $\forall^{=k,\mathcal{RE}}_{\text{string}}$ and $\forall^{=k,\mathcal{CSE}}_{\text{string}}$ are* PSPACE*-complete, for every $k \geq 1$.*

PROOF. Given that $\forall^{=0,\mathcal{RE}}_{\text{string}} = \forall^{<1,\mathcal{RE}}_{\text{string}}$, we conclude from Theorem 4.10 that $\forall^{=0,\mathcal{RE}}_{\text{tree}}$ is in PTIME, from which we can also conclude that $\forall^{=0,\mathcal{CSE}}_{\text{string}}$ is in PTIME. Thus, assume that $k > 0$. Deciding $(A, r) \in \forall^{=k,\mathcal{RE}}_{\text{string}}$ reduces to testing whether $(A, r) \in \forall^{<k+1,\mathcal{RE}}_{\text{string}}$ and $(A, r) \in \forall^{>k-1,\mathcal{RE}}_{\text{string}}$. Given that the former test can be done in polynomial time (by Theorem 4.10) and the latter in polynomial space (by Theorem 4.8), we conclude that $\forall^{=k,\mathcal{RE}}_{\text{string}}$ is in PSPACE. Hence it follows that $\forall^{=k,\mathcal{CSE}}_{\text{string}}$ is in PSPACE as well.

Next, we prove that $\forall^{=k,\mathcal{CSE}}_{\text{string}}$ is PSPACE-hard, from which we conclude that $\forall^{=k,\mathcal{RE}}_{\text{string}}$ is also PSPACE-hard. We reduce from $\forall^{>0,\mathcal{SE}}_{\text{string}}$, which is a PSPACE-hard problem by Theorem 4.8. Let A be a DFA over an alphabet Σ and p a selector expression over the same alphabet. Given that the selector expressions used in the proof of Theorem 4.8 are of the form $//a/*/\cdots/*/b$ with $b \neq *$, we can assume that $p = //a_1/\cdots/a_n$ with $a_n \neq *$. We need to construct a DFA B and an expression $d \in \mathcal{CSE}$ such that $(A, p) \in \forall^{>0,\mathcal{SE}}_{\text{string}}$ if and only if $(B, d) \in \forall^{=k,\mathcal{CSE}}_{\text{string}}$. Assume that x is a new symbol ($x \notin \Sigma$), and then define B as a DFA over $\Sigma \cup \{x\}$ accepting the language $\{w \in (\Sigma \cup \{x\})^* \mid w = w_0 x^k$ for some $w_0 \in \mathcal{L}(A)\}$. Further, define the expression $d \in \mathcal{CSE}$ as

$$d = p//x.$$

Next we argue that $(A, p) \in \forall^{>0,\mathcal{SE}}_{\text{string}}$ if and only if $(B, d) \in \forall^{=k,\mathcal{CSE}}_{\text{string}}$.

($\Rightarrow$) If $(A, p) \in \forall^{>0,\mathcal{SE}}_{\text{string}}$, then for every word in $\mathcal{L}(A)$ the expression p selects at least one position. Thus, it is clear by the definition of B that d will select exactly the last k positions of each word in $\mathcal{L}(B)$. Therefore $(B, d) \in \forall^{=k,\mathcal{CSE}}_{\text{string}}$.

($\Leftarrow$) We prove this direction by considering the contrapositive. Assume that $(A, p) \notin \forall^{>0,\mathcal{SE}}_{\text{string}}$, and let $w \in \mathcal{L}(A)$ be a string such that $|p(w)| = 0$. Then d cannot select any position of the word wx^k, given that $|p(w)| = 0$ and $p = //a_1/\cdots/a_n$ with $a_n \neq *$ and $a_n \neq x$. Thus, given that $wx^k \in \mathcal{L}(B)$ by definition of B, we conclude that $(B, d) \notin \forall^{=k,\mathcal{CSE}}_{\text{string}}$. □

Limiting to disjunctions of selector expressions reduces the complexity of $\forall^{=k,\mathcal{RE}}_{\text{string}}$ to coNP. The proof for this theorem can be found in Appendix B.

THEOREM 4.12. *$\forall^{=0,\mathcal{DSE}}_{\text{string}}$ is in PTIME. Moreover, $\forall^{=k,\mathcal{DSE}}_{\text{string}}$ is coNP-complete, for every $k \geq 1$.*

We conclude this section by proving that $\forall^{=k,\mathcal{SE}}_{\text{string}}$ is in PTIME. Notice this also implies that $\forall^{=k,\mathcal{SE}^*}_{\text{string}}$ and $\forall^{=k,\mathcal{SE}^{//}}_{\text{string}}$ are in PTIME. This proof makes use of a result over finite automata of bounded degree of ambiguity, which is defined next. Let A be an NFA. Given a value $c > 0$, automaton A is said to be *c-ambiguous* if, for every $s \in \mathcal{L}(A)$, there exists at most c accepting runs of A on s. In particular, if $c = 1$, then A is said to be an *unambiguous finite automaton* (UFA). The following is a well-known result about the containment problem for finite automata of bounded degree of ambiguity.

THEOREM 4.13 ([STEARNS AND HUNT III 1985]). *Let $c > 0$ be a fixed constant. There exists a polynomial-time algorithm that, given two c-ambiguous NFAs A and B, verifies whether $\mathcal{L}(A) \subseteq \mathcal{L}(B)$.*

The following lemma forms a cornerstone in the proof of Theorem 4.15. Its proof can be found in Appendix B.

LEMMA 4.14. *Let $k > 0$ be a fixed constant. There exists a polynomial-time algorithm that, given a selector expression p, generates an NFA B^k_p such that:*

(1) for every string s, $s \in \mathcal{L}(B^k_p)$ if and only if $|p(s)| \geq k$, and
(2) for every string $s \in \mathcal{L}(B^k_p)$, the number of accepting runs of B^k_p on s is

$$\frac{|p(s)|!}{(|p(s)| - k)!}.$$

THEOREM 4.15. *For every $k \geq 0$, it holds that $\forall^{=k,\mathcal{SE}}_{\text{string}}$ is in PTIME.*

PROOF. If $k = 0$, then the property is a corollary of Theorem 4.12. Thus, assume that $k \geq 1$. Let A be a DFA and p a selector expression. By Lemma 4.14, we know it is possible to construct in polynomial time an NFA B^k_p such that, for every string s, it holds that $s \in \mathcal{L}(B^k_p)$ if and only if $|p(s)| \geq k$. Thus, to know whether (A, p) is in $\forall^{=k,\mathcal{SE}}_{\text{string}}$, it suffices to check whether $(A, p) \in \forall^{<k+1,\mathcal{SE}}_{\text{string}}$ and $\mathcal{L}(A) \subseteq \mathcal{L}(B^k_p)$. Notice that the latter condition is equivalent to checking whether $\mathcal{L}(A) \subseteq \mathcal{L}(A \times B^k_p)$, where $A \times B^k_p$ denotes the standard product construction of A and B^k_p computing $\mathcal{L}(A) \cap \mathcal{L}(B^k_p)$. With this in mind, we can decide in polynomial time whether $(A, p) \in \forall^{=k,\mathcal{SE}}_{\text{string}}$ by using the following algorithm.

(1) Check whether $(A, p) \in \forall_{\text{string}}^{<k+1,\mathcal{SE}}$. If this condition holds, then go to step (2); otherwise, return false. Notice this step can be executed in polynomial time by Theorem 4.10.
(2) Compute $A \times B_p^k$.
(3) Check whether $\mathcal{L}(A) \subseteq \mathcal{L}(A \times B_p^k)$. Given that $(A, p) \in \forall_{\text{string}}^{<k+1,\mathcal{SE}}$, we have that $|p(s)| \leq k$ for every string $s \in \mathcal{L}(A)$. Moreover, by Lemma 4.14 it follows that, for every string $s \in \mathcal{L}(B_p^k)$, $|p(s)| \geq k$ and the number of accepting runs of B_p^k on s is

$$\frac{|p(s)|!}{(|p(s)| - k)!}.$$

Therefore, for every string s that belongs to $\mathcal{L}(A \times B_p^k)$, it holds that $|p(s)| = k$ and the number of accepting runs of $A \times B_p^k$ on s is bounded by (given that A is a DFA)

$$\frac{k!}{(k-k)!} = k!.$$

We conclude that $A \times B_p^k$ is $k!$-ambiguous. Thus, given that A is a DFA, we have that A is also $k!$-ambiguous, and hence we can verify whether $\mathcal{L}(A) \subseteq \mathcal{L}(A \times B_p^k)$ by using the polynomial-time algorithm mentioned in Theorem 4.13 (for the containment problem for NFAs with bounded degree of ambiguity). □

4.4. Proofs for Results on Trees Mentioned in Table II

In this section we present formal proofs for the results shown in Table II. Most of these proofs require definitions of unranked and binary tree automata. We start by presenting these definitions, recalling some of their basic properties and proving some elementary lemmas.

4.4.1. Tree Automata. A nondeterministic *unranked tree automaton* (UTA) is a tuple $A = (Q, \Delta, \delta, F)$, where Q is a finite set of states, Δ a finite alphabet, $F \subseteq Q$ the set of final states, and $\delta : Q \times \Delta \to 2^{(Q^*)}$ a function mapping each $(q, \sigma) \in Q \times \Delta$ to a regular language over Q. A *run* of $A = (Q, \Delta, \delta, F)$ on a tree t is a function $\lambda : \text{nodes}(t) \to Q$ that assigns a state in Q to each node v of t such that, for every $v \in \text{nodes}(t)$ with children $v_1, v_2, \ldots, v_n$ (noted in the order in which they appear in t), it is the case that $\lambda(v_1)\lambda(v_2)\cdots\lambda(v_n) \in \delta(\lambda(v), \text{lab}_t(v))$. A run is *accepting* if the root of t is labeled by a state in F. If an accepting run of A exists for tree t then we say that t is accepted by A. The set of all trees accepted by A is denoted by $\mathcal{L}(A)$. We say that A is *deterministic* iff $\delta(q, a) \cap \delta(q', a) = \emptyset$ for all $q \neq q' \in Q$ and $a \in \Delta$.

We will also make use of binary tree automata that operate in a top-down fashion over trees where every inner node has precisely two children. Formally, a *binary (top-down) tree automaton* is a tuple $A = (Q, \Delta, q_0, \delta, F)$ where Q is the set of states, Δ the alphabet, $q_0 \in Q$ the start state, $\delta : Q \times \Delta \to 2^{Q \times Q}$ the transition function, and $F \subseteq Q$ the set of final states. A *run* of A on a tree t is a function $\lambda : \text{nodes}(t) \to Q$ that assigns a state in Q to each node v of t such that the root of t is labeled by q_0 and for every inner node v with children v_1 and v_2 it is the case that $(\lambda(v_1), \lambda(v_2)) \in \delta(\lambda(v), \text{lab}_t(v))$. A run is *accepting* if, for every leaf v, $\delta(\lambda(v), \text{lab}_t(v)) \in (F \times F)$. If an accepting run of A exists for tree t then we say that t is accepted by A. The set of all trees accepted by A is denoted by $\mathcal{L}(A)$. We say that A is *deterministic* iff $|\delta(q, a)| \leq 1$ for all $a \in \Delta$ and $q \in Q$. Finally, A is *k-ambiguous* for a natural number k if, for every $t \in \mathcal{L}(A)$, there are at most k distinct accepting runs of A on t.

The following theorem lists well-known results on the complexity of tree automata that will be used in the sequel.

THEOREM 4.16. *Let A and A' be two tree automata that are either both unranked or both binary.*

(1) *Deciding whether $\mathcal{L}(A) = \emptyset$ is in* PTIME.
(2) *A tree automaton B with $\mathcal{L}(B) = \mathcal{L}(A) \cap \mathcal{L}(A')$ can be constructed in time polynomial in the sizes of A and A'. Furthermore, when A and A' are deterministic binary automata, then so is B.*
(3) *Deciding whether $\mathcal{L}(A) = \mathcal{L}(A')$ is* EXPTIME*-complete [Seidl 1990].*
(4) *Assume k is fixed. When A and A' are binary tree automata and k-ambiguous, then deciding whether $\mathcal{L}(A) = \mathcal{L}(A')$ is in* PTIME *[Seidl 1990].*

Note that the tree automaton B computing $\mathcal{L}(A) \cap \mathcal{L}(A')$ is simply the usual product construction between A and A', which we also denote by $A \times A'$.

For an XML tree t, we denote by $\text{fcns}(t)$ the usual first-child next-sibling encoding of t, defined as follows. To facilitate the definition, we define fcns on ordered forests, which consists of concatenations of trees. By ε we denote the empty forest. By $h := t_1 \cdots t_n$, we denote the concatenation of the trees $t_1, \ldots, t_n$. Then, define

—$\text{fcns}(\varepsilon) = \#$; and
—$\text{fcns}(a(h)h') = a(\text{fcns}(h), \text{fcns}(h'))$, for forests h and h'.

Note that $\text{fcns}(t)$ is always a binary tree. We denote $\Sigma \cup \{\#\}$ by $\Sigma_\#$.

To simplify notation, we assume in the following that `data` always belongs to Σ. Therefore we write Σ rather than $\Sigma \cup \{\texttt{data}\}$.

The next lemma says we can always assume that an XSD is represented as a deterministic tree automaton. A proof can be found in Appendix C.

LEMMA 4.17. *Let X be an XSD. A deterministic binary tree automaton A_X can be constructed in time polynomial in the size of X such that, for every XML tree t,*

$$t \in \mathcal{L}(X) \text{ if and only if } \text{fcns}(t) \in \mathcal{L}(A_X).$$

The next lemma shows that the set of fcns encodings of all trees is in fact a regular tree language.

LEMMA 4.18. *There is a deterministic binary tree automaton $A_\#$ such that $t' \in \mathcal{L}(A_\#)$ if and only if $t' = \text{fcns}(t)$ for some XML tree t.*

PROOF. Let $A_X = (\{q_0, q_1, q_{stop}, q_f\}, \Sigma_\#, q_0, \{q_f, q_{stop}\})$. Then define δ as follows.

$$\begin{aligned} \delta(a, q_0) &= (q_1, q_{stop}) \text{ for every } a \in \Sigma \\ \delta(a, q_1) &= (q_1, q_1) \text{ for every } a \in \Sigma \\ \delta(\#, q_1) = \delta(\#, q_{stop}) &= (q_f, q_f) \end{aligned}$$

Basically, the automaton enforces the right child of the root to be a #-labeled node. It also enforces every Σ-labeled node distinct from the root to have two children, and every leaf to be #-labeled. □

It should be noticed that the following lemma is a well-known result about tree automata. Nevertheless, we show its proof as some of the results in the next section use the given construction.

LEMMA 4.19. *Let $k \geq 2$ be a fixed constant. There is a polynomial-time algorithm that, given the UTAs $A_1, A_2, \ldots, A_k$, returns an UTA B such that $\mathcal{L}(B) = \bigcap_{i=1}^{k} \mathcal{L}(A_i)$.*

PROOF. We begin by showing there is a polynomial-time algorithm that, given two regular expressions r_1, r_2 over alphabets Σ and Δ, respectively, returns an automaton

A_{r_1,r_2} over $\Sigma \times \Delta$ accepting the set of strings $(a_1, b_1)(a_2, b_2) \cdots (a_\ell, b_\ell)$ such that $a_1 a_2 \cdots a_\ell \in \mathcal{L}(r_1)$ and $b_1 b_2 \cdots b_\ell \in \mathcal{L}(r_2)$.

Let $A_{r_1} = (Q, \Sigma, \delta, q_0, F_{r_1})$ and $A_{r_2} = (P, \Delta, \rho, p_0, F_{r_2})$ be two automata accepting the languages $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$, respectively. It is well known that these automata can be constructed in polynomial time. Define A_{r_1,r_2} as $(Q \times P, \Sigma \times \Delta, \mu, (p_0, q_0), F_{r_1} \times F_{r_2})$, where μ is defined as

$$\mu((q, p), (a, b)) = \delta(q, a) \times \rho(p, b).$$

Then, it is easy to prove this automaton accepts the set of strings $(a_1, b_1) \cdots (a_\ell, b_\ell)$ such that $a_1 a_2 \cdots a_\ell \in \mathcal{L}(r_1)$ and $b_1 b_2 \cdots b_\ell \in \mathcal{L}(r_2)$.

We use the previous construction in the proof of the lemma. More precisely, given two UTAs A_1, A_2, we show how to construct an UTA B such that $\mathcal{L}(B) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. The extension of this construction for a fixed $k > 2$ can be easily obtained by associating automata in pairs. Assume $A_1 = (Q, \Delta, \delta_1, F_1)$ and $A_2 = (P, \Delta, \delta_2, F_2)$. The new automaton accepting $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ is denoted by $B = (Q \times P, \Delta, \mu, F_1 \times F_2)$, where μ is defined by including the transition

$$\mu((q, p), \sigma) \;=\; \mathcal{L}\left(A_{\delta_1(q,\sigma),\delta_2(p,\sigma)}\right)$$

for every $(p, q) \in P \times Q$ and $\sigma \in \Delta$.

It is clear that all of the preceding can be done in polynomial time. Now we prove that a tree t is accepted by B if and only if it is accepted by both A_1 and A_2. Suppose that t is accepted by B. Then there is an accepting run $\rho : \text{nodes}(t) \to Q \times P$ of B on t. By the definition of transition function μ, we know that, for every node $v \in \text{nodes}(t)$, it is the case that child string$_t(v)$ belongs to $\delta_1(\rho_1(v), \text{lab}_t(v))$, where ρ_1 denotes the projection of ρ over its first component. Thus, the projection of ρ to its first component is a run for the automaton accepting $\mathcal{L}(A_1)$. Moreover, it is an accepting run given that the first coordinate of $\rho(root)$ belongs to F_1, from which we conclude that $t \in \mathcal{L}(A_1)$. For A_2 the proof is analogous by taking the projection over the second component, and the converse is proved analogously. Let t be a tree and ρ_1, ρ_2 be accepting runs of A_1 and A_2 on t, respectively. Then it is clear that $\rho : \text{nodes}(t) \to Q \times P$ defined by $\rho(v) = (\rho_1(v), \rho_2(v))$ is an accepting run of B on t. □

4.4.2. Proofs for $\forall_{\text{tree}}^{>k,\mathcal{P}}$. Next, we proceed to prove the results in Table II. We first consider the problem $\forall_{\text{tree}}^{>k,\mathcal{P}}$. The following lemma is instrumental for the proof of Theorem 4.21 (its proof can be found in Appendix C).

LEMMA 4.20. *Let $k \geq 1$ be a fixed constant. There is a polynomial-time algorithm that, given a regular expression r over an alphabet Δ, constructs an UTA $A_{r,k}$ such that for every Δ-tree t, it holds that $t \in \mathcal{L}(A_{r,k})$ if and only if $|r(t)| \geq k$.*

We now have the necessary ingredients to prove the first three results in Table II concerning $\forall_{\text{tree}}^{>k,\mathcal{P}}$.

THEOREM 4.21. *For every $k \geq 0$, $\forall_{\text{tree}}^{>k,\mathcal{SE}}$, $\forall_{\text{tree}}^{>k,\mathcal{DSE}}$, $\forall_{\text{tree}}^{>k,\mathcal{CSE}}$, and $\forall_{\text{tree}}^{>k,\mathcal{RE}}$ are all* EXPTIME*-complete.*

PROOF. EXPTIME-hardness of $\forall_{\text{tree}}^{>k,\mathcal{SE}}$ follows from results proved in Björklund et al. [2013]. It remains to show membership of $\forall_{\text{tree}}^{>k,\mathcal{RE}}$ in EXPTIME. Let X be an XSD and r a regular expression. From Lemma 4.20, we can construct in polynomial time an UTA $A_{r,k+1}$ accepting every tree t such that $|r(t)| \geq k + 1$. Verifying whether $(X, r) \in \forall_{\text{tree}}^{>k,\mathcal{RE}}$ then reduces to testing whether $\mathcal{L}(X) \subseteq \mathcal{L}(A_{r,k+1})$, which, according to Theorem 4.16, is in EXPTIME. □

Of course, Theorem 4.21 implies that $\forall_{\text{tree}}^{>k,\mathcal{SE}^*}$ is in EXPTIME. Finally, we show that the problem $\forall_{\text{tree}}^{>k,\mathcal{SE}^{//}}$ is tractable.

To prove this result, we need to introduce some terminology and prove two technical lemmas. Given a tree t and $n \geq 1$, we denote by $t_{|n}$ the tree obtained by keeping the nodes from t up to depth n (that is, the tree resulting from removing from t every node at depth $n+1$ along with its corresponding subtree).

LEMMA 4.22. *There is a polynomial-time algorithm that, given an XSD X and a number $n \geq 1$ given in unary, generates an XSD $X_{|n}$ such that, for every tree t, it is the case that*

$$t \in \mathcal{L}(X_{|n}) \text{ iff there is a tree } t' \in \mathcal{L}(X) \text{ such that } t = t'_{|n}.$$

PROOF. Let $X = (A, \lambda)$ be an XSD. The type automaton A' and the function λ' for the new XSD $X_{|n} = (A', \lambda')$ are defined next. Assume that $A = (Q, \Sigma, q_0, \delta)$, and then let $A' = (Q \times [0,n], \Sigma, (q_0, 0), \delta')$, where δ' is defined as

$$\delta'((q,i),a) = (\delta(q,a), i+1) \text{ for every } q \in Q,\ i \in [0, n-1] \text{ and } a \in \Sigma.$$

Now let λ be defined as follows.

$$\begin{array}{rcll} \lambda(q,i) & = & \lambda(q) & \text{for every } q \in Q \text{ and } i \in [0,n-1] \\ \lambda(q,n) & = & \varepsilon & \text{for every } q \in Q \end{array}$$

Notice that the previous automaton can be generated in polynomial time since n is given in unary notation. It is straightforward to prove that $X_{|n}$ satisfies the property in the statement of the lemma. □

In what follows, we make use of binary tree automata that operate in a bottom-up fashion over trees. Formally, a *binary bottom-up tree automaton* is a tuple $A = (Q, \Sigma, q_0, \delta, F)$, where Q is the set of states, Σ is the alphabet, $q_0 \in Q$ is the start state, $\delta : Q \times Q \times \Sigma \to 2^Q$ is the transition function, and $F \subseteq Q$ is the set of final states. A *run* of A on a tree t is a function $\rho : \text{nodes}(t) \to Q$ that assigns a state in Q to every node v of t such that: (1) if v is a leaf, then $\rho(v) \in \delta(q_0, q_0, \text{lab}_t(v))$, and (2) if v is an inner node with children v_1 and v_2, then $\rho(v) \in \delta(\rho(v_1), \rho(v_2), \text{lab}_t(v))$. A run is *accepting* if, for the root v of t, it holds that $\rho(v) \in F$. Finally, we say A is *deterministic* iff $|\delta(q_1, q_2, a)| \leq 1$ for all $a \in \Sigma$ and $q_1, q_2 \in Q$.

The proof of the next lemma can be found in Appendix C.

LEMMA 4.23. *Let $k \geq 0$ be a fixed constant. There is a polynomial-time algorithm that, given a selector expression $p = /a_0/a_1/\cdots/a_{n-1}$, computes a bottom-up deterministic binary tree automaton $A_{|n}^{p,k}$ such that, for every pair of trees t, t' for which $t = fcns(t'_{|n})$, it holds that*

$$t \in \mathcal{L}\big(A_{|n}^{p,k}\big) \text{ if and only if } |p(t')| > k.$$

THEOREM 4.24. *For every $k \geq 0$, $\forall_{\text{tree}}^{>k,\mathcal{SE}^{//}}$ is in* PTIME.

PROOF. Let X be an XSD and let p be a selector expression that does not mention the descendant axis. Assume $p = /a_0/\cdots/a_{n-1}$. Clearly, $(X, p) \in \forall_{\text{tree}}^{>k,\mathcal{SE}^{//}}$ if and only if $(X_{|n}, p) \in \forall_{\text{tree}}^{>k,\mathcal{SE}^{//}}$. Let $A_\#$ be the deterministic binary tree automaton such that $t \in \mathcal{L}(A_\#)$ if and only if $t = \text{fcns}(t')$ for some XML tree t' constructed in Lemma 4.18. Let $A_{X_{|n}}$ be a deterministic binary tree automaton such that, for every tree $t \in \mathcal{L}(A_\#)$,

$$t \in \mathcal{L}(A_{X_{|n}}) \text{ if and only if there exists } t' \in \mathcal{L}(X) \text{ such that } t = \text{fcns}(t'_{|n}).$$

This automaton can be constructed by first computing the XSD provided by Lemma 4.22, and then generating the deterministic binary tree automaton provided by Lemma 4.17. Moreover, let $A_{|n}^{p,k}$ be the bottom-up deterministic tree automaton provided by Lemma 4.23. To decide whether $(X_{|n}, p) \in \forall_{\text{tree}}^{>k,\mathcal{SE}^{//}}$, it suffices to check whether $\mathcal{L}(A_{X_{|n}}) \subseteq \mathcal{L}(A_{|n}^{p,k})$. Given that $A_{X_{|n}}$, $A_{\#}$ and $A_{|n}^{p,k}$ can be constructed in polynomial time in the size of A and p (recall that p mentions n symbols, so n can be assumed given in unary when using Lemma 4.22) and that all these automata are deterministic, we conclude that it can be checked in polynomial time whether $\mathcal{L}(A_{X_{|n}}) \cap \mathcal{L}(A_{\#}) \subseteq \mathcal{L}(A_{|n}^{p,k})$. Thus, it can be checked in polynomial time whether $(X_{|n}, p) \in \forall_{\text{tree}}^{>k,\mathcal{SE}^{//}}$, which was to be shown. □

4.4.3. Proofs for $\forall_{\text{tree}}^{<k,\mathcal{P}}$. We now study the complexity of the problem $\forall_{\text{tree}}^{<k,\mathcal{P}}$. The results in Table II concerning this problem all follow from the next theorem.

THEOREM 4.25. *For every* $k \geq 0$, $\forall_{\text{tree}}^{<k,\mathcal{RE}}$ *is in* PTIME.

PROOF. For $k = 0$ the theorem trivially holds. Thus assume that $k \geq 1$. Let X be an XSD and r a regular expression. From Lemma 4.20, we can construct in polynomial time an UTA $A_{r,k}$ accepting every tree t for which $|r(t)| \geq k$. Then verifying whether $(X, r) \in \forall_{\text{tree}}^{<k,\mathcal{RE}}$ reduces to testing whether $\mathcal{L}(A_{r,k}) \cap \mathcal{L}(X) = \emptyset$, which, by Lemma 4.19 and Theorem 4.16, can be tested in PTIME. □

4.4.4. Proofs for $\forall_{\text{tree}}^{=k,\mathcal{P}}$. Finally, we study the complexity of the problems $\forall_{\text{tree}}^{=k,\mathcal{P}}$. The proofs in some of the following theorems reduce from the corresponding string problems as treated in Section 4.3. Therefore, we need the following lemma to transfer a DFA to a corresponding XSD. It should be noticed that every string s can be naturally viewed as a tree, where each node has a single child except for the last element of s that is a leaf. Thus, given a string s and an XSD X, we use notation $s \in \mathcal{L}(X)$ to indicate that s viewed as a tree conforms to X.

LEMMA 4.26. *Let A be a DFA over an alphabet Σ and σ a symbol not in Σ. Then, there exists a polynomial-time algorithm that returns an XSD such that $\mathcal{L}(X_A) = \{\sigma s \mid s \in \mathcal{L}(A)\}$.*

PROOF. Let $A = (Q, \Sigma, q_0, \delta, F)$, and assume that q_0' is a state not mentioned in Q. Then define X_A as (A', λ), where DFA A' and function λ are defined as follows.

—The set of states, the alphabet, and the initial states of A' are $(Q \cup \{q_0'\})$, $(\Sigma \cup \{\sigma\})$, and q_0', respectively.
—Transition function δ' of A' is defined as follows.

$$\begin{aligned} \delta'(q_0', \sigma) &= q_0 \\ \delta'(q, a) &= \delta(q, a) \quad \text{for every } q \in Q \text{ and } a \in \Sigma \end{aligned}$$

—Finally, for every $q \in Q$, if $\{a_1, \ldots, a_n\}$ is the set of symbols $a \in \Sigma$ for which $\delta(q, a)$ is defined, then

$$\lambda(q) = \begin{cases} (a_1 + \cdots + a_n) & q \in (Q \setminus F) \\ (\varepsilon + a_1 + \cdots + a_n) & q \in F \end{cases}.$$

It is straightforward to prove that $\mathcal{L}(X_A) = \{\sigma s \mid s \in \mathcal{L}(A)\}$. □

The following theorem treats the case where $\mathcal{P}$ equals $\mathcal{RE}$.

THEOREM 4.27. *$\forall_{\text{tree}}^{=0,\mathcal{RE}}$ is in* PTIME *and, for every $k \geq 1$, $\forall_{\text{tree}}^{=k,\mathcal{RE}}$ is in* EXPTIME *and is* PSPACE*-hard.*

PROOF. As $\forall_{\text{tree}}^{=0,\mathcal{RE}} = \forall_{\text{tree}}^{<1,\mathcal{RE}}$, it follows from Theorem 4.25 that $\forall_{\text{tree}}^{=0,\mathcal{RE}}$ is in PTIME. For $k > 0$, $\forall_{\text{tree}}^{=k,\mathcal{RE}} = \forall_{\text{tree}}^{<k+1,\mathcal{RE}} \cap \forall_{\text{tree}}^{>k-1,\mathcal{RE}}$. As $\forall_{\text{tree}}^{<k+1,\mathcal{RE}}$ is in PTIME by Theorem 4.25 and $\forall_{\text{tree}}^{>k-1,\mathcal{RE}}$ is in EXPTIME by Theorem 4.21, it follows that $\forall_{\text{tree}}^{=k,\mathcal{RE}}$ is in EXPTIME.

By Theorem 4.11, we know that $\forall_{\text{string}}^{=k,\mathcal{RE}}$ is PSPACE-complete. In fact, from the proof of this theorem, PSPACE-hardness already follows when restricted to expressions of the form $//a/*/\cdots/*/b//c$, where a, b, and c are different from $*$.

Let A be a DFA and p an expression of the form mentioned earlier as $//a/*/\cdots/*/b//c$. We assume σ is a symbol occurring in neither A nor in p. By Lemma 4.26, we can construct in polynomial time an XSD X_A for which $\mathcal{L}(X_A) = \{\sigma s \mid s \in \mathcal{L}(A)\}$. Given that p is an expression of the form $//a/*/\cdots/*/b//c$, where a, b, and c are different from σ, it follows that $p(s) = p(\sigma s)$ for every string $s \in \mathcal{L}(A)$. Therefore, $(A, p) \in \forall_{\text{string}}^{=k,\mathcal{RE}}$ if and only if $(X_A, p) \in \forall_{\text{tree}}^{=k,\mathcal{RE}}$, which shows that $\forall_{\text{tree}}^{=k,\mathcal{RE}}$ is PSPACE-hard. □

Next, we consider disjunctions and concatenation of selector expressions.

THEOREM 4.28.

(1) $\forall_{\text{tree}}^{=0,\mathcal{DSE}}$ is in PTIME *and, for every $k \geq 1$, $\forall_{\text{tree}}^{=k,\mathcal{DSE}}$ is in* EXPTIME *and is* CONP*-hard.*
(2) $\forall_{\text{tree}}^{=0,\mathcal{CSE}}$ is in PTIME *and, for every $k \geq 1$, $\forall_{\text{tree}}^{=k,\mathcal{CSE}}$ is in* EXPTIME *and is* PSPACE*-hard.*

PROOF. (1) Membership of $\forall_{\text{tree}}^{=0,\mathcal{DSE}}$ in PTIME is a corollary of Theorem 4.27. For $k > 0$, membership of $\forall_{\text{tree}}^{=k,\mathcal{DSE}}$ in EXPTIME follows from Theorem 4.27 as well, so we only need to show that $\forall_{\text{tree}}^{=k,\mathcal{DSE}}$ is CONP-hard. By Theorem 4.12 it follows that $\forall_{\text{string}}^{=k,\mathcal{DSE}}$ is CONP-complete. In fact, from the proof the CONP-hardness already follows when restricted to expressions of the form $(p_1 \mid p_2 \mid \cdots \mid p_k)$, where each p_i $(1 \leq i \leq k)$ is a selector expression not mentioning $//$.

Let A be a DFA and p be a disjunction of selector expressions of the form $(p_1 \mid p_2 \mid \cdots \mid p_k)$, where each p_i $(1 \leq i \leq k)$ does not mention $//$, and assume that σ is a symbol mentioned neither in A nor in p. Define q as the disjunction of selector expressions $(\sigma/p_1 \mid \sigma/p_2 \mid \cdots \mid \sigma/p_k)$ (notice that this expression is well defined as each p_i does not mention $//$). By Lemma 4.26, we can construct in polynomial time an XSD X_A such that $\mathcal{L}(X_A) = \{\sigma s \mid s \in \mathcal{L}(A)\}$. Then, by definition of p and q, we have that $p(s) = q(\sigma s)$ for every string $s \in \mathcal{L}(A)$. Therefore, $(A, p) \in \forall_{\text{string}}^{=k,\mathcal{DSE}}$ if and only if $(X_A, q) \in \forall_{\text{tree}}^{=k,\mathcal{DSE}}$. Hence $\forall_{\text{tree}}^{=k,\mathcal{DSE}}$ is CONP-hard.

(2) It follows as before from Theorem 4.27 and Theorem 4.11. □

We conclude this section by showing that $\forall_{\text{tree}}^{=k,\mathcal{SE}}$ is in PTIME, which implies membership of $\forall_{\text{tree}}^{=k,\mathcal{SE}^*}$ and $\forall_{\text{tree}}^{=k,\mathcal{SE}^{//}}$ in PTIME as well. In this proof, we need two technical lemmas whose proof can be found in Appendix C.

LEMMA 4.29. *There exists a polynomial-time algorithm that, given a selector expression p, computes a binary tree automaton A_p such that for every XML tree t it holds that $|p(t)|$ is equal to the number of accepting runs of A_p on $fcns(t)$.*

The following lemma is the counterpart of Lemma 4.14 for trees.

LEMMA 4.30. *Let $k \geq 1$ be a fixed constant. There is a polynomial-time algorithm that, given a selector expression p, computes a binary tree automaton B_p^k such that for every XML tree t:*

(1) fcns(t) ∈ $\mathcal{L}(B_p^k)$ if and only if $|p(t)| \geq k$, and
(2) if $t \in \mathcal{L}(B_p^k)$, then the number of accepting runs of B_p^k on t is

$$\frac{|p(t)|!}{(|p(t)|-k)!}.$$

We are now ready to prove our main tractability result.

THEOREM 4.31. *For every $k \geq 0$, $\forall_{\text{tree}}^{=k,\mathcal{SE}}$ is in* PTIME.

PROOF. If $k = 0$, then the property is a corollary of Theorem 4.28. Thus assume that $k \geq 1$. Let X be an XSD and p a selector expression. By Lemma 4.30, we know it is possible to construct in polynomial time a binary tree automaton B_p^k such that, for every XML tree t, it holds that fcns$(t) \in \mathcal{L}(B_p^k)$ if and only if $|p(t)| \geq k$. Moreover, by Lemma 4.29, we know it is possible to construct a deterministic binary tree automaton A_X such that, for every XML tree t, it holds that fcns$(t) \in \mathcal{L}(A_X)$ if and only if $t \in \mathcal{L}(X)$. Let $A_\#$ be the deterministic binary tree automaton such that $t' \in \mathcal{L}(A_\#)$ if and only if $t' =$ fcns(t) for some XML tree t, as shown in Lemma 4.18. Now, to know whether (X, p) is in $\forall_{\text{tree}}^{=k,\mathcal{SE}}$, it suffices to check whether $(X, p) \in \forall_{\text{tree}}^{<k+1,\mathcal{SE}}$ and $\mathcal{L}(A_\# \times A_X) \subseteq \mathcal{L}(B_p^k)$. Notice that the latter condition is equivalent to checking whether $\mathcal{L}(A_\# \times A_X) \subseteq \mathcal{L}(A_X \times B_p^k)$. With this in mind, we can decide in polynomial time whether $(X, p) \in \forall_{\text{tree}}^{=k,\mathcal{SE}}$ by using the following algorithm.

(1) Check whether $(X, p) \in \forall_{\text{tree}}^{<k+1,\mathcal{SE}}$. If this condition holds, then go to step (2); otherwise, return false. Notice that this step can be executed in polynomial time by Theorem 4.25, since every selector expression is a regular expression.
(2) Compute $A_\# \times A_X$ and $A_X \times B_p^k$.
(3) Check whether $\mathcal{L}(A_\# \times A_X) \subseteq \mathcal{L}(A_X \times B_p^k)$. Given that $(X, p) \in \forall_{\text{tree}}^{<k+1,\mathcal{SE}}$, we have that for every XML tree t in $\mathcal{L}(X)$, it holds that $|p(t)| \leq k$. Moreover, by Lemma 4.30 we have that for every XML tree t accepted by B_p^k, it holds that $|p(t)| \geq k$ and the number of accepting runs of B_p^k on t is

$$\frac{|p(t)|!}{(|p(t)|-k)!}.$$

Therefore, for every XML tree t that belongs to $\mathcal{L}(A_X \times B_p^k)$, it holds that $|p(t)| = k$ and the number of accepting runs of $A_X \times B_p^k$ on t is bounded by (given that A_X is a deterministic binary tree automaton):

$$\frac{k!}{(k-k)!} = k!.$$

We conclude that $A_X \times B_p^k$ is $k!$-ambiguous. Thus, given that $A_\# \times A_X$ is a deterministic binary tree automaton, we have that $A_\# \times A_X$ is also $k!$-ambiguous and hence we can verify whether $\mathcal{L}(A_\# \times A_X) \subseteq \mathcal{L}(A_X \times B_p^k)$ by using the polynomial-time algorithm mentioned in Theorem 4.16. □

4.5. Determining the Quality of Keys

In this section, we investigate a number of additional criteria to determine the quality of keys. Since the number of keys mined from a given document can be quite large, we are interested in identifying irrelevant keys that can be disregarded from the output of any key mining algorithm. Examples are keys that hold in any document (called *universal* keys), keys that can only address a bounded number of target nodes independent of

the size of the input document (called *bounded* keys), keys that are implied by keys that have already been found, and keys that are not satisfied by any document (called *unsatisfiable* keys). Whether bounded keys should be considered irrelevant is perhaps debatable but, intuitively, since the main purpose of a key is to ensure uniqueness of nodes within an *a priory unbounded* collection of nodes, we consider bounded keys to not be semantically very interesting.

Throughout our development we restrict our attention to keys already known to be consistent. Thereto, let X be an XSD, ϕ a key and Ψ be a set of keys such that every key in $\Psi \cup \{\phi\}$ is consistent with respect to X. Then:

—UNIVERSALITY is the problem to decide whether $t \models \phi$ for every tree in $t \in \mathcal{L}(X)$.
—BOUNDEDNESS is the problem to decide whether there is an $N \in \mathbb{N}$ such that, for every tree $t \in \mathcal{L}(X)$, $|\,\text{TNodes}_t(\phi)| \leq N$.
—KEY IMPLICATION, denoted by $\Psi \Rightarrow \phi$, is the problem to decide whether, for all trees $t \in \mathcal{L}(X)$ such that $\bigwedge_{\psi \in \Psi} t \models \psi$, it holds that $t \models \phi$.
—SATISFIABILITY is the problem to decide whether there is a tree $t \in \mathcal{L}(X)$ with $t \models \phi$.

We will show that identifying universal and bounded keys is algorithmically feasible, while determining implication and even satisfiability of keys is intractable. Therefore, determining a smallest set of keys (also known as a *cover*) is practically infeasible. It should be noted, however, that while the EXPTIME-completeness of SATISFIABILITY is discouraging, it does not pose a problem for key mining algorithms in practice. Indeed, by Definition 3.9 a key mining algorithm will, on input (X, t) with $t \in \mathcal{L}(X)$, only return keys ϕ with $t \models \phi$ (which can efficiently be checked since t is given). As such, the keys ϕ it returns are necessarily satisfiable.

The fact that a smallest set of keys cannot be found efficiently is more problematic from a practical viewpoint. Fortunately, there is a sufficient condition for key implication, called *target path equivalence*, that can be solved efficiently and that can hence be used to prune the set of mined keys. Formally, given an XSD X, a context c, and two selector expressions τ and τ', TARGET PATH CONTAINMENT is the problem to decide whether, for every tree $t \in \mathcal{L}(X)$ and every node $v \in \text{CNodes}_t(c)$, $\tau(t, v) \subseteq \tau'(t, v)$. We denote the latter condition by $\tau \subseteq_{X,c} \tau'$. By TARGET PATH EQUIVALENCE we denote the decision problem of checking containment in both directions, that is, whether $\tau \subseteq_{X,c} \tau'$ and $\tau' \subseteq_{X,c} \tau$. TARGET PATH EQUIVALENCE is a particularly relevant problem for key mining since it allows to identify, within the mined set of keys, the semantically equivalent but distinct keys (c, τ, P) and (c, τ', P) with τ target path equivalent to τ'. In this sense, target path equivalence is a sufficient condition for key implication.

In the following sections we will establish the following complexity results concerning the problems introduced previously. Similar to the previous section, we parametrize the prior problems by a class $\mathcal{P}$ of expressions so as to restrict attention to input keys that only use expressions in $\mathcal{P}$.

THEOREM 4.32. *The following hold.*

(*1*) UNIVERSALITY($\mathcal{DSE}$) *is in* PTIME.
(*2*) BOUNDEDNESS($\mathcal{DSE}$) *is in* PTIME.
(*3*) KEY IMPLICATION($\mathcal{SE}$) *is* EXPTIME*-hard.*
(*4*) SATISFIABILITY($\mathcal{SE}$) *is* EXPTIME*-complete.*
(*5*) TARGET PATH CONTAINMENT($\mathcal{SE}$) *and* TARGET PATH EQUIVALENCE($\mathcal{SE}$) *are in* PTIME.

4.5.1. Universality. We first prove that UNIVERSALITY($\mathcal{DSE}$) is in PTIME. We start with the following observation.

LEMMA 4.33. *Let t_1 and t_2 be two XML trees that only differ on* **Data***-labeled nodes, that is, t_1 and t_2 have the same sets of nodes, the same sets of edges (which in particular respects the ordering of children), and for all nodes n:*

—$\text{lab}_{t_1}(n) \in \Sigma$ *if and only if* $\text{lab}_{t_2}(n) \in \Sigma$*; and*
—*if* $\text{lab}_{t_1}(n) \in \Sigma$ *then* $\text{lab}_{t_1}(n) = \text{lab}_{t_2}(n)$.

Then, for every XSD X, every selector expression p, and every node n, we have:

(1) $t_1 \in \mathcal{L}(X)$ *if and only if* $t_2 \in \mathcal{L}(X)$;
(2) $p(t_1, n) = p(t_2, n)$.

The proof is straightforward. Note in particular that by this property every XML tree t can be transformed into an XML tree t' in which distinct **Data** nodes carry distinct labels such that t and t' are *invariant* with respect to XSD membership and selector expression evaluation. Likewise, t can be transformed into a tree t'' in which all **Data** nodes carry the same label. We will use both transformations in what follows.

PROPOSITION 4.34. UNIVERSALITY($\mathcal{DSE}$) *is in* PTIME.

PROOF. First observe a necessary and sufficient condition for the universality of key $\phi = (c, \tau, P)$ in the XSD X is that, for every $t \in \mathcal{L}(X)$, for every $v \in \text{CNodes}_t(c)$, the set $\tau(t, v_i)$ is empty or contains exactly one node.

Indeed, suppose (for the purpose of contradiction) that ϕ is universal but that there exists $t \in \mathcal{L}(X)$ and $v \in \text{CNodes}_t(c)$ such that $\tau(t, v)$ contains at least two distinct nodes, say n_1 and n_2. By Lemma 4.33, we may assume without loss of generality that all **Data** nodes in t carry the same label, say $a \in$ **Data**. Then clearly $\text{record}_P(t, n_1) = \text{record}_P(t, n_2)$ but $n_1 \neq n_2$. Hence $t \not\models \phi$, which yields the desired contradiction.

Conversely, it is straightforward to check that ϕ is universal if for every $t \in \mathcal{L}(X)$, for every $v \in \text{CNodes}_t(c)$, the set $\tau(t, v_i)$ is empty or contains exactly one node.

Now, it is well known that in polynomial time one can construct an XSD X' from X such that $\mathcal{L}(X')$ is exactly the set of trees t' for which there exists $t \in \mathcal{L}(X)$ and $v \in \text{CNodes}_t(c)$ such that t' is the subtree rooted at v in t.

As such, by our necessary and sufficient condition as given before, ϕ is universal if and only if $(X', \tau) \in \forall_{\text{tree}}^{<2,\mathcal{RE}}$, which can be decided in polynomial time by Theorem 4.25. □

4.5.2. Satisfiability and Key Implication. Next, we show that KEY IMPLICATION($\mathcal{SE}$) and SATISFIABILITY($\mathcal{SE}$) are EXPTIME-hard and that SATISFIABILITY($\mathcal{SE}$) is in EXPTIME. We begin with the hardness results.

PROPOSITION 4.35. KEY IMPLICATION($\mathcal{SE}$) *is* EXPTIME*-hard.*

PROOF. We reduce $\forall_{\text{tree}}^{>1,\mathcal{SE}}$ in polynomial time to KEY IMPLICATION($\mathcal{SE}$). Since the former is EXPTIME-hard by Theorem 4.21, so is the latter.

Let (X, p) be an input to $\forall_{\text{tree}}^{>1,\mathcal{SE}}$ with X an XSD and $p \in \mathcal{SE}$ a selector expression. From the results of Björklund et al. [2013], it follows that we may assume without loss of generality that p is of the form $p = .//a/*/*/\ldots/*/b$ with $a, b \in \Sigma$ (in other words, $\forall_{\text{tree}}^{>1}\mathcal{SE}$ remains hard when restricted to such inputs).

Let #,#$_1$, and #$_2$ be new symbols not in Σ. Let, for every $t \in \mathcal{L}(X)$, t' be the tree

$$t' = \#(t, \#_1(\#_2), \#_1(\#_2)).$$

Construct from X the XSD X' that accepts the set of all such trees t' in which #$_2$ is a **Data** node. Let c be the root context of X'.

Note in particular that, when we evaluate p on t', p can only match in the subtree t. Hence $p(t') = p(t)$. Therefore $|p(t)| > 1$ for every $t \in \mathcal{L}(X)$ if and only if $|p(t')| > 1$ for every $t' \in \mathcal{L}(X')$.

Now consider the key (ϕ, X') with $\phi = (c, p, P)$ and $P = [\,]$ the empty sequence of key paths. Clearly (ϕ, X') is consistent. Moreover, (ϕ, X') is unsatisfiable if and only if $|p(t')| > 1$, for every tree $t' \in \mathcal{L}(X')$. Indeed, for every tree $t' \in \mathcal{L}(X')$ and all nodes u_1, u_2 in t' we have $\text{record}_P(t', u_1) = \text{record}_P(t', u_2) = [\,]$, the empty record. Hence (ϕ, X') is unsatisfiable if, and only if, for every tree $t' \in \mathcal{L}(X')$ and every $v \in \text{CNodes}_{t'}(c)$, there are at least two distinct nodes in $p(t', v)$.

Then let $\psi = (c, , ./\#_1, ./\#_2)$. Again, (ψ, X') is clearly consistent. Moreover, $\phi \Rightarrow \psi$ if and only if (ϕ, X) is unsatisfiable. Indeed, if ϕ is not satisfiable, then any implication with ϕ on the left-hand side holds. Conversely, if ϕ is satisfiable then there is a tree $t' \in \mathcal{L}(X)$ such that $t' \models \phi$. Then, let t'' be the tree obtained from t' by changing the **Data** value of both $\#_2$-labeled nodes to the same value. Then $t'' \models \phi$ by Lemma 4.33, but $t'' \not\models \psi$.

We conclude that $|p(t)| > 1$ for every $t \in \mathcal{L}(X)$ if and only if $|p(t')| > 1$ for every $t' \in \mathcal{L}(X')$ if and only if (ϕ, X') is unsatisfiable if and only if $\phi \Rightarrow \psi$. Hence KEY IMPLICATION($\mathcal{SE}$) is EXPTIME-hard. □

PROPOSITION 4.36. SATISFIABILITY($\mathcal{SE}$) *is* EXPTIME*-hard.*

PROOF. We reduce $\forall^{>1,\mathcal{SE}}_{\text{tree}}$ in polynomial time to SATISFIABILITY($\mathcal{SE}$). Since the former is EXPTIME-hard by Theorem 4.21, so is the latter.

Let (X, p) be an input to $\forall^{>1,\mathcal{SE}}_{\text{tree}}$ with X an XSD and $p \in \mathcal{SE}$ a selector expression. Let c be the root context of X. Then $|p(t)| > 1$ for every $t \in \mathcal{L}(X)$ if and only if the key (ϕ, X), with $\phi = (c, p, P)$ and $P = [\,]$ the empty sequence of key paths, is unsatisfiable. Indeed, for every tree $t \in \mathcal{L}(X)$ and all nodes u_1, u_2 in t we have $\text{record}_P(t, u_1) = \text{record}_P(t, u_2) = [\,]$, the empty record. Hence (ϕ, X) is unsatisfiable if and only if, for every tree $t \in \mathcal{L}(X)$ and every $v \in \text{CNodes}_t(c)$ there are at least two distinct nodes in $p(t, v)$. □

To tackle the upper bound of satisfiability, we first observe the following characterization of unsatisfiability. We say that a selector expression is *descendant based* if it starts with the descendant axis, that is, when it is of the form $.//l_1/l_2/\ldots/l_k$. An XML key (ϕ, X) with $\phi = (c, \tau, P)$ is *descendant based* if every $p \in P$ is descendant based. In other words, a key is not descendant based if a selector expression occurring in P starts with the child axis rather than the descendant axis.

LEMMA 4.37. *Let (ϕ, X) with $\phi = (c, \tau, P)$ be a consistent XML key such that P is nonempty and all paths in τ and P are in $\mathcal{SE}$. Then (ϕ, X) is unsatisfiable if and only if:*

(1) (ϕ, X) is descendant based; and
(2) for every tree $t \in \mathcal{L}(X)$ there is a node $v \in \text{CNodes}_t(c)$ and a pair of distinct nodes $u_1, u_2 \in \tau(t, v)$ with u_2 a descendant of u_1.

PROOF. ($\Leftarrow$) Let (ϕ, X) be a consistent XML key and suppose that (1) and (2) hold. We show that for every tree $t \in \mathcal{L}(X)$ we have $t \not\models \phi$. Hereto, let $t \in \mathcal{L}(X)$ be arbitrary. By (2) we know there exists $v \in \text{CNodes}_t(c)$ and a pair of distinct nodes $u_1, u_2 \in \tau(t, v)$ with u_2 a descendant of u_1. By (1) we know that every $p \in P$ is an expression of the form $.//l_1/l_2/\ldots/l_k$. Then, since u_2 is a descendant of u_1, we obtain that $p(t, u_2) \subseteq p(t, u_1)$, for every $p \in P$. Now, since (ϕ, X) is assumed consistent, $p(t, u_2)$ and $p(t, u_1)$ must be singleton sets. Hence $p(t, u_2) = p(t, u_1)$ for every $p \in P$. As such, $\text{record}_P(t, u_1) = \text{record}_P(t, u_2)$ but $u_1 \neq u_2$. Therefore $t \not\models \phi$.

($\Rightarrow$) Suppose for the purpose of contradiction that either (1) or (2) does not hold. We show that (ϕ, X) is necessarily satisfiable.

—If (1) does not hold then some $p \in P$ is of the form $p = ./l_1/l_2/\ldots/l_k$. Then let t be an arbitrary tree in $\mathcal{L}(X)$ such that distinct **Data** nodes in t carry distinct **Data** labels. Such a tree always exists since XSDs are invariant to the actual data values used in trees (Lemma 4.33). We show that $t \models \phi$. Hereto, let $v \in \text{CNodes}_t(c)$ be arbitrary and let u_1, u_2 be distinct nodes in $\tau(t, v)$. We need to show that $\text{record}_P(t, u_1) \neq \text{record}_P(t, u_2)$. Since (ϕ, X) is consistent, $p(t, u_1)$ and $p(t, u_2)$ are singletons, say $p(t, u_1) = \{n_1\}$ and $p(t, u_2) = \{n_2\}$ for some nodes n_1, n_2 in t. We will show that $n_1 \neq n_2$. Then, since distinct **Data** nodes carry distinct labels, we have $\text{value}_t(n_1) \neq \text{value}_t(n_2)$ and $\text{record}_P(t, u_1) \neq \text{record}_P(t, u_2)$, as desired. We distinguish two cases.
  (i) Case u_1 is not a descendant of u_2 and u_2 is not a descendant of u_1. Then, since t is a tree, the set of nodes $p(t, u_1) = \{n_1\}$ reachable by p from u_1 in t is necessarily disjoint with $p(t, u_2) = \{n_2\}$, that is, $n_1 \neq n_2$, as desired.
  (ii) Case u_1 is a descendant of u_2 or u_2 is a descendant of u_1. Then, since t is a tree, since u_1 and u_2 are at different levels in t, and since p is of the form $p = ./l_1/l_2/\ldots/l_k$, the set $p(t, u_1) = \{n_1\}$ of **Data** nodes reachable by p starting from u_1 in t is necessarily disjoint with $p(t, u_2) = \{n_2\}$, that is, $n_1 \neq n_2$, as desired.

—If (2) does not hold then there exists $t \in \mathcal{L}(X)$ such that, for all $v \in \text{CNodes}_t(c)$ and all pairs of distinct nodes $u_1, u_2 \in \tau(t, v)$, u_2 is not a descendant of u_1, nor is u_2 a descendant of u_1.
By Lemma 4.33 we may assume without loss of generality that each **Data** node in t carries a distinct **Data** element. We will show that $t \models \phi$. Hereto, let $v \in \text{CNodes}_t(c)$ be arbitrary and let u_1, u_2 be distinct nodes in $\tau(t, v)$. We need to show that $\text{record}_P(t, u_1) \neq \text{record}_P(t, u_2)$. Hereto, let $p \in P \neq \emptyset$ be arbitrary. Since (ϕ, X) is consistent, $p(t, u_1)$ and $p(t, u_2)$ are singletons, say $p(t, u_1) = \{n_1\}$ and $p(t, u_2) = \{n_2\}$ for some nodes n_1, n_2 in t. Observe that, since t is a tree, since u_1 isn't a descendant of u_2, and since u_2 isn't a descendant of u_1, it follows that $n_1 \neq n_2$. Hence since distinct **Data**-labeled nodes carry distinct labels, $\text{value}_t(n_1) \neq \text{value}_t(n_2)$. Hence $\text{record}_P(t, u_1) \neq \text{record}_P(t, u_2)$, as desired. □

PROPOSITION 4.38. SATISFIABILITY($\mathcal{SE}$) *is in* EXPTIME.

PROOF. Let $(\phi = (c, \tau, P), X)$ be a consistent key. The algorithm proceeds by a case analysis to check that (ϕ, X) is satisfiable.

Case P is empty. First, the algorithm checks whether P is empty. If so, then observe that, for every tree $t \in \mathcal{L}(X)$ and all nodes u_1, u_2 in t, we have $\text{record}_P(t, u_1) = \text{record}_P(t, u_2) = [\,]$, the empty record. Hence (ϕ, X) is unsatisfiable if and only if, for every tree t and every $v \in \text{CNodes}_t(c)$, there are at least two distinct nodes in $\tau(t, v)$.

Now, it is well known that in polynomial time one can construct an XSD X' from X such that $\mathcal{L}(X')$ is exactly the set of trees t' for which there exists $t \in \mathcal{L}(X)$ and $v \in \text{CNodes}_t(c)$ such that t' is the subtree rooted at v in t. Then, (ϕ, X) is satisfiable if and only if $(X', \tau) \in \forall^{>1,\mathcal{DSE}}_{\text{tree}}$, which can be decided in EXPTIME by Theorem 4.21.

Case P is nonempty and P not descendant based. If P is not empty, then the algorithm checks whether some element of P is not descendant based. If so, ϕ is satisfiable by Lemma 4.37.

Case P is nonempty and P is descendant based. Finally, if P is nonempty and descendant based then the algorithm uses the formalism of alternating tree-walking automata for which containment is known to be in EXPTIME [Bojanczyk 2008]. Basically, the automaton can be constructed as follows: let A_1 be an automaton that

nondeterministically walks to a node and checks that it is in context node c, then it nondeterministically walks down to a node u_1 selected by τ, and finally the automaton accepts if it finds a descendant u_2 of u_1 that is also selected by τ starting from the context node. Then, since X is descendant based, by Lemma 4.37 it suffices to check whether $\mathcal{L}(X) \subseteq \mathcal{L}(A_1)$, which can be tested in EXPTIME. □

4.5.3. Boundedness. Next, we show that BOUNDEDNESS($\mathcal{SE}$) is in PTIME. Using this result, we then show that BOUNDEDNESS($\mathcal{DSE}$) is in PTIME as well. We require the following notation, insight, and definitions.

Let t be an XML tree and u be a node in t. We write $t|_u$ for the subtree rooted at u in t. Furthermore, we write $t[v \leftarrow t']$ for the XML document obtained from t by replacing the subtree in t rooted at v by t'. Here, we assume without loss of generality that the set of nodes of t and t' are disjoint. If this is not the case, we first make an isomorphic copy of t', disjoint with t. Clearly, the result of $t[v \leftarrow t']$ is unique up to isomorphism.

The following result, due to Martens et al. [2006], states that XSDs are invariant under the modification of subtrees that have the same context.

PROPOSITION 4.39 (XSD SUBTREE EXCHANGE PROPERTY [MARTENS ET AL. 2006]). *Let X be an XSD and let $t_1, t_2 \in \mathcal{L}(X)$ be two XML trees adhering to X. Let c be a context, let $u \in \text{CNodes}_{t_1}(c)$, and let $v \in \text{CNodes}_{t_2}(c)$. Then $t_1[u \leftarrow (t_2|_v)]$ adheres to X.*

The preceding lemma will be a technical tool to show that unboundedness is decidable in PTIME. As a first step, we first show that *horizonal unboundedness*, defined as follows, is decidable in PTIME.

Definition 4.40. A key (ϕ, X) is *horizontally unbounded* if there exists $\ell \in \mathbb{N}$ such that, for every $N \in \mathbb{N}$, we can find $t \in \mathcal{L}(X)$ of depth at most ℓ with $|\text{TNodes}_t(\phi)| > N$. If a key is unbounded but not horizontally unbounded, then it is called *vertically unbounded*.

PROPOSITION 4.41. *It is decidable in* PTIME *whether a given key (ϕ, X) with ϕ containing only selector expressions is horizontally unbounded.*

PROOF. The crux of the decision algorithm is an analysis of the set of contexts reachable by X while simulating ϕ. We will need the following concepts to define the algorithm and prove it correct.

Let $X = (A, \lambda)$ with $A = (\text{Types}, \Sigma \cup \{\texttt{data}\}, \delta, q_0)$ and let $\phi = ((\sigma_c, q_c), \tau, P)$ with $\tau \in \mathcal{SE}$ all $p \in P$ in $\mathcal{SE}$. We focus here on the case where $\tau = .//l_1/l_2/\ldots/l_k$; the reasoning when τ starts with a child axis rather than a descendant axis is similar.

Since XSDs can be trimmed in polynomial time, we assume without loss of generality in what follows that our input XSD X is trimmed.

Now consider the directed graph $G_X = (C, \rightarrow)$ defined as follows.

—The set of nodes C of G_X consists of all well-formed contexts of X (well-formed contexts are defined in Definition 4.2).
—There is an edge $c \rightarrow c'$ from $c = (\sigma, q)$ to $c' = (\sigma', q')$ in G_X if and only if $\sigma' \in \mathit{Out}(q)$ and $q' = \delta(q, \sigma')$. (Recall that $\mathit{Out}(q)$ denotes the set of all Σ symbols for which $\delta(q, \sigma')$ is defined or, equivalently, the set of all Σ symbols occurring in the regular expression $\lambda(q)$.)

Note that, since there are at most $|\Sigma| \times |\text{Types}|$ well-formed contexts, G_X can clearly be constructed in polynomial time.

We write $\rightarrow^+$ for the transitive closure of the edge relation $\rightarrow$. Furthermore, we write $(\sigma, q) \stackrel{\infty}{\rightarrow} (\sigma', q')$ when $(\sigma, q) \rightarrow (\sigma', q')$ and σ' appears under the scope of a Kleene-star

operator in the regular expression $\lambda(q)$. To illustrate, a, b, and c all appear under the scope of a Kleene-star in $a(b + ca)^*d$, but d does not. We write $(\sigma, q) \xrightarrow{1} (\sigma', q')$ to indicate that $(\sigma, q) \to (\sigma', q')$ but σ' does not appear under the scope of a Kleene-star operator in $\lambda(q)$. We say that a path $c_1, c_2, \ldots, c_n$ from context c_1 to context c_n in G_x *traverses* a $\xrightarrow{\infty}$ edge if there exists $1 \leq i \leq n$ with $c_i \xrightarrow{\infty} c_{i+1}$.

Let the set of *initial contexts* I be defined by $I := \{(\sigma, \delta(q_0, \sigma)) \mid \sigma \in Out(q_0)\}$.

We say that a context $c = (\sigma, q)$ *matches* $l \in \Sigma \cup \{*\}$ if $\sigma = l$ or $l = *$.

To obtain the proposition, we use the following characterization of horizontal unboundedness.

CLAIM 1. *Key (ϕ, X) as given earlier is horizontally unbounded if and only if all of the following hold:*

(1) *there exists $c_0 \in I$ with $c_0 \to^+ (\sigma_c, q_c)$;*
(2) *there exist $c_1, \ldots, c_k \in C$ such that c_i matches l_i, for $1 \leq i \leq k$ and $(\sigma_c, q_c) \to^+ c_1 \to c_2 \cdots \to c_k$; and*
(3) *at least one of the paths in items (1) and (2) traverses a $\xrightarrow{\infty}$ edge.*

The correctness of this claim follows essentially by a pumping argument: if the path in condition (1) traverses an $\xrightarrow{\infty}$ edge we can arbitrarily pump the number of nodes that have context (σ_c, q_c) (thereby increasing $|\text{TNodes}_t(\phi)|$); if the path in condition (2) traverses an $\xrightarrow{\infty}$ edge we can arbitrarily pump the number of nodes that are selected by τ (thereby also increasing $|\text{TNodes}_t(\phi)|$ to N). The proof of Claim 1 is provided in Appendix D.

Observe that condition (1) can easily be verified in polynomial time using a standard graph reachability algorithm. Condition (2) can be verified in polynomial time by computing the following sequence of sets $T_1, \ldots, T_k$ and verifying that T_k is nonempty.

$$T_1 := \{c_1 \mid (\sigma_c, q_c) \to^+ c_1 \text{ and } c_1 \text{ matches } l_1\}; \text{ and}$$
$$T_{i+1} := \{c_{i+1} \mid \exists c_i \in T_i, c_i \to c_{i+1} \text{ and } c_{i+1} \text{ matches } l_{i+2}\}$$

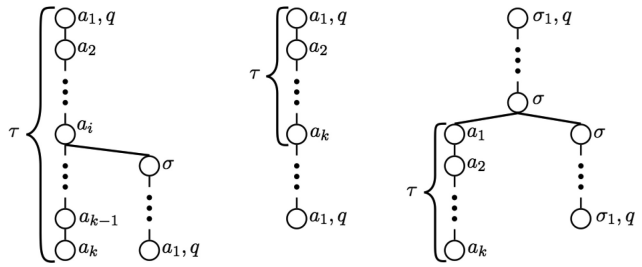
Finally, condition (3) can be verified in polynomial time by suitably keeping track of when a $\xrightarrow{\infty}$-edge is traversed in the prior algorithms for tracking (1) and (2). As such, horizontal unboundedness of (φ, X) can be checked in PTIME. □

PROPOSITION 4.42. *Boundedness($\mathcal{SE}$) is in* PTIME.

PROOF. First, we can decide in PTIME whether (ϕ, X) is horizontally unbouded by Proposition 4.41. If (ϕ, X) is horizontally unbouded, then it is clearly also unbouded.

Otherwise, we reason as follows. Since XSDs can be trimmed in polynomial time, we may assume without loss of generality that X is trimmed. Let $\phi = ((\sigma_c, q_c), \tau, P)$ $\tau \in \mathcal{SE}$ all $p \in P$ in $\mathcal{SE}$. We focus on the case where $\tau = .//a_1/a_2/\ldots/a_k$ (the case where τ starts with a child axis is similar). Let C be the set of all well-formed contexts of X. For an XML tree $t \in \mathcal{L}(X)$, let $\rho(t)$ be the C tree obtained by labeling each node v in t by its context according to X, that is, $\text{lab}_{\rho(t)}(v) = c$ where c is the unique context with $v \in \text{CNodes}_t(c)$. Finally, let $T = \{\rho(t) \mid t \in \mathcal{L}(X)\}$ be the set of all context-labeled versions of trees adhering to X.

We claim that (ϕ, X) is unbounded if and only if some $\rho(t) \in T$ contains a node v labeled by (σ_c, q_c) such that $t|_v$ contains one of the following three tree templates as a (partial) subtree.



Here, we have omitted states for nodes where the states are not important; q represents an arbitrary state; and σ, σ_1 represent arbitrary alphabet symbols. Also, the order among siblings is not important.

Indeed, if T contains such a tree $\rho(t)$ then we can show $\mathcal{L}(X)$ to be unbounded as follows. Suppose that $t|_v$ contains the first tree template as a subtree (the other two cases are similar). Let u_1 be the (a_1, q)-labeled node in t at which this subtree is rooted and let u_2 be the (a_1, q)-labeled descendant of u_1. Then let $t' := t[u_2 \leftarrow t|_{u_1}]$. By the Subtree Exchange Property (Propostion 4.39), $t' \in \mathcal{L}(X)$. Moreover, observe that $|\operatorname{TNodes}_{t'}(\phi)| > |\operatorname{TNodes}_t(\phi)| > 1$. Since t' still contains a copy of $t|_{u_1}$ as a subtree under v, we can iterate this reasoning until $|\operatorname{TNodes}_{t'}(\phi)|$ reaches the desired size N.

Conversely, suppose for the purpose of contradiction that (ϕ, X) is unbounded but T does not contain a tree $\rho(t)$ as described before. We then derive a contradiction by showing that (ϕ, X) is horizontally unbounded (which we assumed not to be the case). In particular, define for every well-formed context c the natural number $\mathit{mindepth}(c)$ as

$$\mathit{mindepth}(c) := \min\{\mathit{depth}(t|_u) \mid t \in \mathcal{L}(X), u \in \operatorname{CNodes}_t(c)\}.$$

Let $\mathit{mindepth}(X) := \max_{c \in C} . \mathit{mindepth}(c)$, fix $\ell := 2 \times |\Sigma| \times |\text{Types}| + \mathit{mindepth}(X)$, and let $N \in \mathbb{N}$ be arbitrary but fixed. We will show there exists a tree s in $\mathcal{L}(X)$ of depth at most ℓ with $|\operatorname{TNodes}_s(\phi)| > N$.

Hereto, we reason as follows. Since (ϕ, X) is unbounded, there exists a tree $t_N \in \mathcal{L}(X)$ with $|\operatorname{TNodes}_s(\phi)| > N$. For simplicity of illustration we will assume that all of the nodes in $\operatorname{TNodes}_{t_N}(\phi)$ are descendants of a single node v_N in t_N that has context (σ_c, q_c) (i.e., $\operatorname{lab}_{\rho(t_N)}(v_N) = (\sigma_c, q_c)$). The reasoning when $\operatorname{TNodes}_{t_N}(\phi)$ is distributed over multiple nodes with context (σ_c, q_c) in t_N is similar, but more verbose.

Observe that we may assume without loss of generality that the path from v_N to the root in t_N is of size at most $|\Sigma| \times |\text{Types}|$. Indeed, since there are only $|\Sigma| \times |\text{Types}|$ distinct contexts, there must exist nodes z and z', both on this path, that have the same context (in the sense that $\operatorname{lab}_{\rho(t_N)}(z) = \operatorname{lab}_{\rho(t_N)}(z')$) if this path is of larger size. Assume without loss of generality that z' is the descendant of z. Then $t_1 := t_N[z \leftarrow t_N|_{z'}]$ still adheres to X by the Subtree Exchange Property. Moreover, it has the same number of target nodes as the number of target nodes selected by ϕ in t_N since all of the latter target nodes belong to $t_N|z_n$, which is still present in t_1. By iterating this reasoning until no context is repeated on the path from v_N to the root, we obtain a tree in $\mathcal{L}(X)$ in which this path is of length at most $|\Sigma| \times |\text{Types}|$ and that has the same number of target nodes.

By our assumption that T does not contain a tree $\rho(t)$ as described earlier, we know in particular that $t_N|_{v_N}$ does not contain any of the preceding three tree templates as a subtree. This implies that, for all descendants u of v_N, if u has a descendant in $\operatorname{TNodes}_{t_N}(\phi)$ then u does not have another descendant u' with the same context.

In particular, since there are only $|\Sigma| \times |\text{Types}|$ distinct contexts, the length of the path from any node in $\operatorname{TNodes}_{t_N}(\phi)$ to v_N must be of length at most $|\Sigma| \times |\text{Types}|$. Then

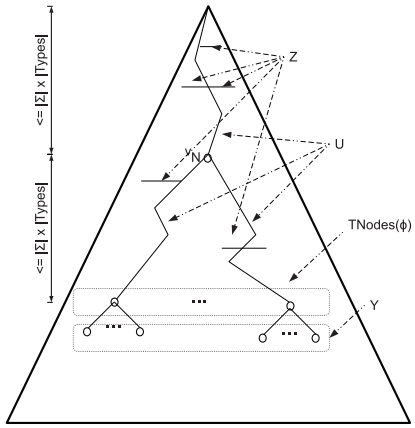


Fig. 4. Illustration of the construction in the proof of Proposition 4.42.

let U be the set of all nodes that occur on the path from a node in $\text{TNodes}_{t_N}(\phi)$ to t_N's root. Let Y be the set of child nodes of nodes in $\text{TNodes}_{t_N}(\phi)$. Let Z be the set of all sibling nodes of nodes in U, excluding the nodes in U themselves. Our situation so far is illustrated in Figure 4.

Now fix, for every node $u \in Y \cup Z$, a tree $t_u \in \mathcal{L}(X)$ and a node $v_u \in \text{CNodes}_{t_u}(c_u)$ with c_u the context of u in t_N, $c_u = \text{lab}_{\rho(t_N)}(u)$, such that $\mathit{depth}(t_u|_{v_u}) \leq \mathit{mindepth}(X)$. Such t_u and v_u exist by the definition of $\mathit{mindepth}(X)$ and the fact that X is trimmed.

Then let s be the tree obtained from t_N by simultaneously replacing, for every $u \in Y \cup Z$, the tree rooted at u in t_N by $t_u|_{v_u}$. By the Subtree Exchange Property, $s \in \mathcal{L}(X)$. Moreover, since we did not touch any of the nodes in t_N on the paths from nodes in $\text{TNodes}_{t_N}(\phi)$ to t_N's root, we have $\text{TNodes}_{t_N}(\phi) \subseteq \text{TNodes}_s(\phi)$. Therefore $N < |\,\text{TNodes}_{t_N}(\phi)| \leq \text{TNodes}_s(\phi)$. Finally, we have replaced all children of nodes in $\text{TNodes}_{t_N}(\phi)$ by trees of depth at most $\mathit{mindepth}(X)$. In addition, we have replaced all siblings of nodes on the path from nodes in $\text{TNodes}_{t_N}(\phi)$ also by trees of depth at most $\mathit{mindepth}(X)$. Then, since all paths from nodes $\text{TNodes}_{t_N}(\phi)$ to the root are of length at most $2 \times |\Sigma| \times |\text{Types}|$, the resulting tree s must be of depth at most $2 \times |\Sigma| \times |\text{Types}| + \mathit{mindepth}(X) = \ell$, as desired.

To complete the proof, we need to show that we can check whether T contains a tree $\rho(t)$ as stated earlier in polynomial time. Hereto, we construct four (unranked) tree automata, A_1, A_2, A_3, and A_4. The first automaton A_1 is constructed to recognize exactly T. It is not hard to verify that this automaton can be constructed from X in polynomial time. The other three automata A_2,A_3, and A_4 are constructed to accept trees in which the first, respectively second and third, tree template given before occurs as a subtree beneath a (σ_c, q_c)-labeled node. It is not hard to verify that these automatons can be constructed from (ϕ, X) in polynomial time. Essentially, it suffices to construct the automatons nondeterministically, guessing the context that needs to be repeated, and verifying the target path τ. Finally, we check emptiness of $\mathcal{L}(A_1) \cap (\mathcal{L}(A_2) \cup \mathcal{L}(A_3) \cup \mathcal{L}(A_4))$, that can be done in polynomial time in the size of A_1, A_2, A_3, and A_4 (that are all of size polynomial in (ϕ, X)). If the result is empty then the key is bounded, otherwise it is unbounded. □

In the following proposition, we reduce BOUNDEDNESS($\mathcal{DSE}$) to BOUNDEDNESS($\mathcal{SE}$).

PROPOSITION 4.43. BOUNDEDNESS($\mathcal{DSE}$) *is in* PTIME.

PROOF. Let $(\phi = (c, \tau, P), X)$ be a key with a target path $\tau = \tau_1 \mid \cdots \mid \tau_n$ that consists of a union of selector expressions. Then, clearly (ϕ, X) is bounded if and only if $((c, \tau_i, P), X)$ is bounded for every $i \leq n$. Hence BOUNDEDNESS($\mathcal{DSE}$) is in PTIME. □

Next, we prove that TARGET PATH CONTAINMENT($\mathcal{SE}$) can be solved in PTIME, from which it immediately follows that TARGET PATH EQUIVALENCE($\mathcal{SE}$) is also in PTIME.

We start by observing the following characterization of target path containment. Given a tree t and a context c, define $paths(t, c)$ as the set of strings s for which there exist nodes u, v in t such that $u \in \text{CNodes}_t(c)$, v is a descendant of u in t, and s is the string formed by the labels on the unique path form u to v (including the labels of u and v). Moreover, given an XSD X, define $paths(X, c)$ as $\bigcup_{t \in \mathcal{L}(X)} paths(t, c)$.

LEMMA 4.44. *Let X be an XSD, c a context, and τ, τ' selector expressions. Then $\tau \subseteq_{X,c} \tau'$ if, and only if for every $s \in paths(X, c)$, it holds that if $s \in \mathcal{L}(\tau)$, then also $s \in \mathcal{L}(\tau')$.*

PROOF. ($\Rightarrow$) Assume that (τ, c) is contained in (τ', c) with respect to X, and let $s \in paths(X, c)$ be such that $s \in \mathcal{L}(\tau)$. We need to prove $s \in \mathcal{L}(\tau')$. Given that $s \in paths(X, c)$, there exists a tree $t \in \mathcal{L}(X)$ such that $s \in paths(t, c)$. Thus, there exist nodes u, v in t such that $u \in \text{CNodes}_t(c)$, v is a descendant of u in t, and s is the string formed by the labels on the unique path from u to v (including the labels of u and v). We conclude that $v \in \tau(t, u)$ and therefore we have that $v \in \tau'(t, u)$ as (τ, c) is contained in (τ', c) with respect to X. Hence we conclude by definition of $paths(t, c)$ that $s \in \mathcal{L}(\tau')$.

($\Leftarrow$) Assume that (τ, c) is not contained in (τ', c) with respect to X. Then we need to show that there exists $s \in paths(X, c)$ such that $s \in \mathcal{L}(\tau)$ and $s \notin \mathcal{L}(\tau')$. Given that (τ, c) is not contained in (τ', c) with respect to X, there exists a tree $t \in \mathcal{L}(X)$ and nodes u, v in t such that $u \in \text{CNodes}_t(c)$, $v \in \tau(t, u)$ and $v \notin \tau'(t, u)$. Let s be the string formed by the labels on the unique path from u to v (including the labels of u and v). Then we have that $s \in paths(t, c)$, from which we conclude that $s \in paths(X, c)$. Moreover, given that $v \in \tau(t, u)$, we have that $s \in \mathcal{L}(\tau)$, and given that $v \notin \tau'(t, u)$, we have that $s \notin \mathcal{L}(\tau')$. This concludes the proof of the lemma. □

PROPOSITION 4.45. TARGET PATH CONTAINMENT($\mathcal{SE}$) *is in* PTIME.

PROOF. Assume given an XSD X, a context c, and selector expressions τ, τ'. To check that $\tau \subseteq_{X,c} \tau'$ it suffices by Lemma 4.44 to check that $(paths(X, c) \cap \mathcal{L}(\tau)) \subseteq \mathcal{L}(\tau')$. We prove in Appendix D that we can construct in polynomial time a DFA $A_{X,c}$ and UFAs A_τ and $A_{\tau'}$ such that $\mathcal{L}(A_{X,c}) = paths(X, c)$; $\mathcal{L}(A_\tau) = \mathcal{L}(\tau)$; and $\mathcal{L}(A_{\tau'}) = \mathcal{L}(\tau')$. Thus, the problem of verifying whether $(paths(X, c) \cap \mathcal{L}(\tau)) \subseteq \mathcal{L}(\tau')$ can be reduced in polynomial time to the problem of verifying whether $\mathcal{L}(A_{X,c} \times A_\tau) \subseteq \mathcal{L}(A_{\tau'})$, where $A_{X,c} \times A_\tau$ is the (usual) product automaton of $A_{X,c}$ and A_τ, which can be computed in polynomial time and accepts $\mathcal{L}(A_{X,c}) \cap \mathcal{L}(A_\tau)$. Given that $A_{X,c}$ is a DFA and A_τ is a UFA, we have that $A_{X,c} \times A_\tau$ is a UFA. Thus, given that $A_{\tau'}$ is also a UFA, we conclude that $\mathcal{L}(A_{X,c} \times A_\tau) \subseteq \mathcal{L}(A_{\tau'})$ can be checked in polynomial time, as the containment problem for UFAs can be solved in polynomial time by Theorem 4.13. □

The following theorem then readily follows.

PROPOSITION 4.46. TARGET PATH EQUIVALENCE($\mathcal{SE}$) *is in* PTIME.

5. XML KEY MINING ALGORITHM

In this section, we provide an algorithm for solving the XML key mining problem. Recall from Definition 3.9 that the input to this algorithm is an XSD X, an XML tree t, and a minimum support threshold N, and that it should output keys that are consistent with

ALGORITHM 1: XML Key Mining Algorithm

```
for all c ∈ ContextMiner_{t,X} do
    for all τ ∈ TargetPathMiner_{t,X}(c) do
        S = OneKeyPathMiner_{t,X}(c, τ)
        P = MinimalKeyPathSetMiner_{t,X}(c, τ, S)
        for each P ∈ P return (c, τ, P)
```

X, are satisfied by t, and whose support exceeds N.[7] For the remainder, let $X = (A_X, \lambda_X)$ with the type automaton $A_X = (\text{Types}, \Sigma \cup \{\texttt{data}\}, \delta, q_0)$.

The overall structure of the XML key mining algorithm is outlined in Algorithm 1. Basically the algorithm consists of four components.

—$\text{ContextMiner}_{t,X}$ returns a list of possible contexts based on t and X.
—$\text{TargetPathMiner}_{t,X}(c)$ returns a list of target paths with minimal support in t given a context c.
—$\text{OneKeyPathMiner}_{t,X}(c, \tau)$ returns a maximal set S of key paths for which $(c, \tau, \{p\})$ is consistent for every $p \in S$.
—$\text{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$ returns a set $\mathcal{P}$ of minimal subsets P of S for which $t \models (c, \tau, P)$.

$\text{TargetPathMiner}_{t,X}(c)$ and $\text{OneKeyPathMiner}_{t,X}(c, \tau)$ are different instantiations of levelwise search [Mannila and Toivonen 1997], while the function $\text{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$ leverages on discovery algorithms for functional dependencies in the relational model. In the remainder, we explain each function in detail. We will only consider target and key paths up to a given length k_{max} which can be at most the maximum depth of the document. Since the presence of top-level disjunction renders testing for consistency intractable (see Theorem 4.6), we focus on a key mining algorithm that disregards the union operator.

To illustrate the different parts of the mining algorithm, we will use the XML document t depicted in Figure 1 as a running example.

5.1. Prefix Tree and Context Miner

We first define a basic data structure that is used to speed up various parts of the mining algorithm. Denote by $\text{PT}(t)$ the prefix tree obtained from t by collapsing all nodes with the same ancestor string. Recall that the ancestor string of a node is the string obtained by concatenating all labels on the unique path from the root to (and including) this node. Let h be the function mapping each node in t to its corresponding node in $\text{PT}(t)$. Then, we label every node m in $\text{PT}(t)$ with a pair (q, i), where q is the state assigned to m by the type automaton A_X and i is the number of nodes in t mapped to m, that is, $|h^{-1}(m)|$. Note that $\text{PT}(t)$ does not contain data nodes. The prefix tree can be computed in time linear in the size of t (see, e.g., [Grahne and Zhu 2002]).

We next discuss the context miner. Clearly, the set of all contexts $c = (\sigma, q)$ with $\sigma \in \Sigma$ and $q \in \text{Types}$ can be directly inferred from the given XSD. But, since only contexts that are effectively materialized in t can give rise to a nonzero support, the context miner enumerates all unique contexts c occurring in $\text{PT}(t)$ through a depth-first traversal.

Example 5.1. The prefix tree for the XML tree in Figure 1 is shown in Figure 5. The type automaton depicted in Figure 3 is used to assign a unique state to each node in the prefix tree. The set of materialised contexts can now be derived by combining,

[7] If no XSD is available, one can be derived, for example, using algorithms from Bex et al. [2007].

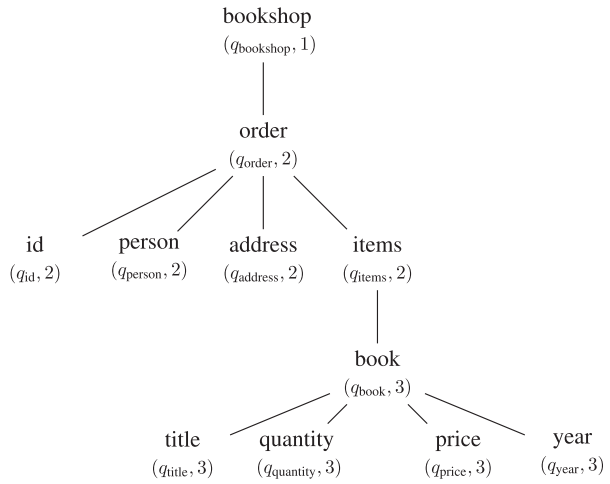


Fig. 5. Prefix tree for the XML tree in Figure 1. Each node has an associated state and number of matches.

for each node, its label with the assigned context. In this case the context miner yields the following set:

$$\{(q_{\text{book}}, \text{book}), (q_{\text{bookshop}}, \text{bookshop}), (q_{\text{order}}, \text{order}), (q_{\text{quantity}}, \text{quantity}), (q_{\text{year}}, \text{year}), (q_{\text{title}}, \text{title}), (q_{\text{person}}, \text{person}), (q_{\text{price}}, \text{price}), (q_{\text{id}}, \text{id}), (q_{\text{address}}, \text{address}), (q_{\text{items}}, \text{items})\}.$$

5.2. Target Path Miner

Next, we describe the target path miner which finds all target paths exceeding the support threshold for a given context c. The algorithm follows the framework of *levelwise search* described by Mannila and Toivonen [1997]. In brief, the algorithm is of a generate-and-test style that starts from the most general target path, .//* in our case, and generates increasingly more specific paths while avoiding paths that cannot be interesting given the information obtained in earlier iterations.

The components of any levelwise search algorithm consist of a set U called the *search space*, a predicate q on U called the *search predicate*, and a partial order $\preceq$ on U called the *specialization relation*. The goal is to find all elements of U that satisfy the search predicate. Obviously, U in our case is the set of selector expressions up to length k_{max}. A standard approach is to use a support threshold for the search predicate. Accordingly, we define the search predicate as $q(\tau) := \text{supp}(c, \tau, t) > N$, for the given input threshold N. That is, τ is deemed interesting when its support exceeds N.

For levelwise search to work correctly, q should be *monotone* (actually, monotonically decreasing) with respect to $\preceq$, meaning that if $\tau' \preceq \tau$ and $q(\tau)$ holds, then $q(\tau')$ holds as well. The intuition of $\tau' \preceq \tau$ is that τ is more specific than τ', or, in other words, that τ' is more general than τ. For our purposes, it would be ideal to use the semantic containment relation $\tau \subseteq_{X,c} \tau'$ in context c (as defined in Section 4.5). Although this containment relation is shown tractable (Theorem 4.46) through a translation to the inclusion test of unambiguous string automata, it is not well suited to be used within the framework of levelwise search that requires fast testing of specialization due to the large number of such tests. In strong contrast, as we show shortly, the containment of selector expressions, that disregards the presence of a schema, has a syntactic counterpart which can be implemented efficiently. Therefore we define $\tau' \preceq \tau$ if and only if, for every XML tree t, the set $\tau(t)$ is a subset of $\tau'(t)$. With respect to this definition

ALGORITHM 2: Basic algorithm for levelwise search [Mannila and Toivonen 1997]

$C_0 :=$ set of most general elements of U;
$i := 0$;
while $C_i \neq \emptyset$ **do**
  $F_i := \{\tau \in C_i \mid q(\tau)\}$;
  $C_{i+1} := \{\tau \in U \mid \forall \tau' \in U : \tau' \prec \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\} \setminus \bigcup_{j \leq i} C_j$;
  $i := i + 1$;
return $\bigcup_{j<i} F_j$;

it is obvious that the search predicate q is monotone. Notice also that $\tau \subseteq \tau'$ implies $\tau \subseteq_{X,c} \tau'$.

Now, levelwise search computes sets F_i iteratively as shown in Algorithm 2. Here, $\prec$ is the strict version of $\preceq$, so $\tau' \prec \tau$ if $\tau' \preceq \tau$ and $\tau' \neq \tau$. The step computing C_{i+1} is called *candidate generation*; those candidates that satisfy q then end up in the corresponding set F_{i+1} (the letter F is a shorthand for "frequent", referring to the support threshold). It can formally be shown that the union of all sets F_i indeed equals the set of all elements of U satisfying q [Mannila and Toivonen 1997]. Moreover, the algorithm is terminated as soon as C_i is empty, because then all later sets F_j and C_j with $j \geq i$ will be empty as well.

The abstract framework as given, however, leaves questions to be answered.

(i) How can we efficiently evaluate the search predicate $q(\tau)$?
(ii) *How can we efficiently generate candidate sets C_{i+1}?*

We will next answer these questions in detail.

Search predicate. The search predicate $\text{supp}(c, \tau, t)$ can be entirely evaluated on the prefix tree $\text{PT}(t)$ and does not need access to the original document t. A single XPath expression can be used to aggregate the counts of all nodes matching τ below nodes in context c.[8] Indeed, for $c = (\sigma, q)$, the support can be obtained from $\text{PT}(t)$ using the XPath expression

$$\texttt{sum(//}\sigma\texttt{[@state=}id_q\texttt{]/}\tau\texttt{/@matches)},$$

where id_q is the internally used id of the state q. The attributes `@state` and `@matches` contain, respectively, the state id assigned to the node in the prefix tree and the number of nodes with the same ancestor path in t.

Specialization relation and candidate generation. Since our chosen specialization relation is purely semantic, we need an equivalent algorithmic definition to show that containment can be effectively decided. Thereto, we define a *one-step specialization relation* whose repeated application corresponds to the semantic specialization relation as follows: $\tau' \prec_1 \tau$ if τ is obtained from τ' by one of the following operations: (a) if τ' starts with the descendant axis, replace it by the child axis; (b) if τ' starts with the descendant axis, insert a wildcard step right after it; or (c) replacing a wildcard with an element name.

We establish that $\tau' \preceq \tau$ if and only if τ' can be transformed into τ by a sequence of $\prec_1$-steps, or, more formally (a proof can be found in Appendix E), as follows.

PROPOSITION 5.2. *The relation $\preceq$ equals the reflexive and transitive closure of the relation $\prec_1$.*

Note that the definition of $\prec_1$ makes it impossible that $\tau' \prec_1 \tau'' \prec_1 \tau$ while at the same time $\tau' \prec_1 \tau$. Hence, Proposition 5.2 implies that $\prec_1$ as defined before really is

[8]Recall that in the prefix tree every node contains its corresponding context and count.

the "successor" relation of $\preceq$. More formally, $\tau' \prec_1 \tau$ holds precisely if and only if $\tau' \prec \tau$ and there exists no intermediate τ'' such that $\tau' \prec \tau'' \prec \tau$. Moreover, $\prec_1$ is very efficient to compute. Thus armed, we can perform candidate generation in a effective manner as given in Algorithm 3. Here, candidate generation is split up into two steps, which in practice can be interleaved. The set G_{i+1} takes all successors of the current set F_i; the set C_{i+1} then prunes away those elements that have a predecessor that does not satisfy q. It can be shown formally that the sets F_i computed in this concrete manner are exactly the same as those prescribed by the levelwise algorithm (a proof can be found in Appendix E).

ALGORITHM 3: $\text{TargetPathMiner}_{t,X}(c)$

$C_0 :=$ set of minimal elements of U;
$i := 0$;
while $C_i \neq \emptyset$ **do**
  $F_i := \{\tau \in C_i \mid q(\tau)\}$;
  $G_{i+1} := \{\tau \in U \mid \exists \tau' \in F_i : \tau' \prec_1 \tau\}$;
  $C_{i+1} := \{\tau \in G_{i+1} \mid \forall \tau' : \tau' \prec_1 \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\}$;
  $i := i + 1$;
return $\bigcup_{j<i} F_j$;

THEOREM 5.3. *Algorithms 2 and 3 are equivalent.*

Example 5.4. We now illustrate the first iteration of Algorithm 3. Consider the context $c = (\text{order}, q_{\text{order}})$ from our running example, together with a support threshold of 3.

The target miner starts with the minimal elements as a set of candidates. In this case this corresponds to the most general path: $C_0 = \{.//*\}$. The predicate q is then checked for $.//*$. As this path selects all 23 nondata nodes below the `order` nodes, it is supported and the predicate evaluates to `true`. Hence $F_0 = \{.//*\}$. Next, the specialization relation is used to generate the next level. Applying the three possible specialization operations on $.//*$ yields the following set.

$$G_1 = \{./*,\ .//*/*,\ .//\texttt{items},\ .//\texttt{book}, \ldots\}$$

For each of these paths it is checked whether all the parent paths are supported (only $.//*$ in this case). In the next iteration, the predicate will be evaluated for these new candidates and a new level will be generated (if possible). Note that the sets C_i in Algorithm 2 would have the same value.

Duplicate elimination. Often, a nuisance in mining logical formulas such as selector expressions is duplicate elimination: different expressions may be logically equivalent. Fortunately, in our setting, it follows from Proposition 5.2 that only identical selector expressions can be equivalent.

Regardless, it can happen that two derived, and therefore inequivalent, target paths τ and τ' select precisely the same set of target nodes on the given document t. As these paths are equivalent from the perspective of t, it holds that $t \models (c, \tau, P)$ iff $t \models (c, \tau', P)$ for all sets P. Therefore, with respect to generation of key paths P, it does not make sense to consider all of these equivalent path separately. Rather, we should choose among them one canonical path. One possibility, for instance, is to opt for the most specific path according to $\prec_1$ by minimizing the length and number of wildcards. Notice that equivalence of target paths on t can be tested on the prefix tree $\text{PT}(t)$ without access to the original document.

Example 5.5. Consider the context q_{order} and the target paths `.//book` and `./items/book`. When evaluating these target paths on the small prefix tree in Figure 5, we notice that the same nodes are selected and we can therefore conclude they are equivalent on the document. Indeed, both target paths will select all the book nodes in the larger XML document (see Figure 1).

Boundedness elimination. The quality of the mining result can be improved using the results of Section 4.5. Indeed, target paths that are bounded but still have passed the support threshold N, which may happen with low values of N, may be eliminated at this stage.

Example 5.6. Suppose the XSD in our running example would limit the number of `order` nodes in a document to a maximum of 10. We could then eliminate those target paths that select (descendants of) these nodes.

5.3. One-Key Path Miner

Our task here is to find all key paths p for which $(c, \tau, (p))$ is consistent on the given document, that is, for every $v \in \text{CNodes}_t(c)$ and every $u \in \tau(t, v)$, it holds that $p(t, u)$ is a singleton containing a **Data** node. In a second step, the key paths p for which $(c, \tau, (p))$ is consistent with respect to X are selected for further processing. The reason for this two-step approach is to reduce the number of costly consistency tests. Although testing for consistency with respect to a schema is in polynomial time (refer to Theorem 4.6), it can be slow for large schemas and is ill suited to be used directly as a search predicate. Therefore, we test for document consistency in a first step and make use of the fact that inconsistency on t implies inconsistency on X. That is, key paths which are not consistent on t and which are therefore pruned in the first step can never be consistent with respect to X.

It turns out that, again, a levelwise search may be used, utilizing the converse of the specialization relation $\preceq$ for target path mining. So, define $p' \preceq^{\text{key}} p$ iff $p \preceq p'$, that is, $p' \preceq^{\text{key}} p$ iff $p' \subseteq p$. The search predicate $q_\tau^{\text{key}}(p)$ is now defined to hold if p selects at most one node in t for each of the target nodes selected by τ in context c. This q_τ^{key} is indeed monotonically decreasing with respect to the converse of containment among selector expressions: $p' \preceq^{\text{key}} p \equiv p' \subseteq p$ and $q_\tau^{\text{key}}(p)$ together imply $q_\tau^{\text{key}}(p')$. We note that consistency requires the selection of exactly one (rather than at most one) node. However, this mismatch can be solved by confining the search space U_{key} to all selector expressions up to length k_{max} that from a target node select a leaf node in the prefix tree: these expressions select at least one node by virtue of their being present in the prefix tree. The "most general" elements from which the levelwise search is started are then those paths in the prefix tree from target nodes to leafs. Obviously, U_{key} can be computed directly from $\text{PT}(t)$.

It remains to discuss how to compute q_τ^{key} efficiently. Unfortunately, q_τ^{key} cannot always be computed solely on $\text{PT}(t)$. Indeed, consider the documents $t_1 = a(b(d), b(d))$ and $t_2 = a(b(d, d), b)$, where each d node is a **Data** node. Then, $\text{PT}(t_1) = \text{PT}(t_2)$, yet ϕ is consistent on t_1 but inconsistent on t_2 for $\phi = (c_{\text{root}}, ./b, (./d))$ with c_{root} the root context.

We next present a sufficient condition for inconsistency that can be tested on the prefix tree. Thereto, consider $\phi = (c, \tau, (p))$ and let $t' = \text{PT}(t)$. For a node m in t', we denote by $\#_{t'}(m)$ the number assigned to m in t', that is, $|h^{-1}(m)|$ for h as defined in Section 5.1. Define the following conditions.

—(C1) There exists a $v \in \text{CNodes}_{t'}(c)$ and a $u \in \tau(t', v)$ such that $\#_{t'}(u) < \sum_{w \in p(t', u)} \#_{t'}(w)$.
—(C2) There exists a $v \in \text{CNodes}_{t'}(c)$, a $u \in \tau(t', v)$, a $w \in p(t', u)$, and a node m on the path from u to w such that $\#_{t'}(m) < \#_{t'}(w)$.

Here, (C1) says that the number of target nodes u is strictly smaller than the number of nodes selected by p, and (C2) says that there is a leaf node selected by p and an ancestor with a smaller number of corresponding nodes in t. Both conditions imply there are at least two nodes selected by p that belong to the same target node in t and which contradict consistency.

The following proposition hence follows.

PROPOSITION 5.7. *Given $\phi = (c, \tau, (p))$ and a document t. If condition C1 or C2 holds on* PT(t)*, then ϕ is inconsistent on t.*

So, only when the tests for the two preceding conditions fail do we evaluate p on t to determine the value of $q_\tau^{\text{key}}(p)$.

Finally, define $\prec_1^{\text{key}}$ as the inverse of $\prec_1$, that is, $p' \prec_1^{\text{key}} p$ iff $p \prec_1 p'$. Then, the first step of OneKeyPathMiner$_{t,X}(c, \tau)$ is the same algorithm as depicted in Algorithm 3 with U, q, and $\prec_1$, replaced by U_{key}, q_τ^{key}, and $\prec_1^{\text{key}}$, respectively. The second step in OneKeyPathMiner$_{t,X}(c, \tau)$ retains, from all of the returned key paths p, those for which $(c, \tau, (p))$ is consistent with respect to X employing the algorithm of Theorem 4.6. A duplicate elimination step similar to the one of the previous section is performed as well.

Example 5.8. When considering the context $(\text{order}, q_{\text{order}})$ and the target path `.//book`, the one-key path miner will generate candidate paths starting from the leaf paths in the prefix tree:

$$\{\texttt{./quantity}, \texttt{./title}, \texttt{./year}, \texttt{./price}\}.$$

None of these is found inconsistent by either (C1) or (C2), nor by the XML document itself (see Figure 1). This is because all of them appear exactly once. But, after the XSD consistency check, the path `./year` is removed. Indeed, when we inspect the XSD more closely (see Example 3.2), we see that year is optional. This means there are XML documents that satisfy the XSD, but for which the key is inconsistent. In the next iterations, the algorithm will generate more general paths by applying the converse specialization relation, as described earlier. In this case, paths such as `./*` will violate the consistency requirement, while paths of the form `.//quantity` are equivalent to their nondescendant counterparts. The final output of this phase is therefore:

$$\{\texttt{./quantity}, \texttt{./title}, \texttt{./price}\}.$$

5.4. Minimal Key Path Set Miner

At this point, we have computed the maximal set S for which every $p \in S$, $(c, \tau, (p))$ is consistent with respect to X. Next, we are looking for minimal and meaningful sets $P \subseteq S$ such that $t \models (c, \tau, P)$, that is, such that (c, τ, P) is a key for t. Note that such a set P can be trivially converted to a sequence to satisfy the definition of an XML key as defined in Section 3.3.

We capitalize on existing relational techniques for mining functional dependencies (e.g., [Bitton et al. 1989; Mannila and Räihä 1989, 1994]). To this end, we define a relation $R_{S,t}$ with the schema

$$(CID, TID, p_1, p_2, \ldots, p_{|S|}),$$

where *CID* and *TID* are columns for the selected context nodes and target nodes, respectively, and every p_i corresponds to the unique **Data** value selected by the corresponding key path p_i. Then, $(v, u, \bar{o}) \in R_{S,t}$ if and only if $v \in \text{CNodes}_t(c)$, $u \in \tau(t, v)$ and $\text{record}_S(t, u) = \bar{o}$. Now, it follows that $t \models (c, \tau, P)$ iff

$$CID, p_1, p_2, \ldots, p_n \rightarrow TID$$

Table III. Data Table for Key
((order, q_{order}), .//book, {./title, ./price, ./quantity})
and the XML Document in Figure 1

CID	TID	title	price	quantity
o1	b1	Movie analysis	5.63	63
o1	b2	Programming intro	6.72	63
o2	b3	Programming intro	5.63	150

is a functional dependency in $R_{S,t}$ for $P = (p_1, \ldots, p_n)$. We can now plug in any existing functional dependency discovery algorithm.

Example 5.9. From previous phases we obtain the following consistent candidate key:

$$((\text{order}, q_{\text{order}}),\ \texttt{.//book},\ \{\texttt{./title},\ \texttt{./price},\ \texttt{./quantity}\}),$$

yielding the relation in Table III. We observe that

$$\begin{aligned} \mathit{CID}, \mathit{title} &\rightarrow \mathit{TID}, \\ \mathit{CID}, \mathit{price} &\rightarrow \mathit{TID}, \\ \mathit{CID}, \mathit{quantity} &\not\rightarrow \mathit{TID}, \end{aligned}$$

and can hence derive the final XML keys for the considered context and target path as

$$\begin{aligned} &((\text{order}, q_{\text{order}}),\ \texttt{.//book}, (\texttt{./title})), \\ &((\text{order}, q_{\text{order}}),\ \texttt{.//book}, (\texttt{./price})). \end{aligned}$$

6. EXPERIMENTS

In this section, we analyse the performance of different parts of the mining algorithm and also look at different optimizations to understand their impact on the execution time and number of derived keys.

For our experiments, we use a corpus of 90 high-quality XML documents and associated XSDs obtained from Grijzenhout and Marx [2010]. The input can therefore be seen as 90 pairs, each consisting of a unique XML document and a unique XSD. The maximal and average number of elements occurring in documents is 91K and 5K, respectively, while the maximal and average number of elements occurring in XSDs is 532 and 52, respectively. All experiments are with respect to to this corpus and were run on a 3 GHz Mac Pro with 2GB of RAM. In all experiments, we set k_{max} to 4 for target paths and to 2 for key paths, unless explicitly mentioned otherwise.

Choosing constants. We need to determine meaningful values for k_{max} and the support threshold. In this section we derive suitable bounds. To determine k_{max}, we examine the distribution of the *target paths' lengths*. To this end, we generated for all documents the set of target paths up to length 10, having a minimal support of 1, that is, each target path must select at least one node in the XML document. Already 88.03% of the (canonical) target paths have a length up to 4, and 96.39% up to 6. But more importantly, when looking at the removal rates from canonization (removal of equivalent target paths) in Table IV, we see that larger paths have a much higher removal rate. For example, for length-6 paths, 98.35% are covered by canonical paths of length 6 or smaller. This means that, especially for larger path lengths, a significant portion of the candidate target paths are superfluous. The latter is exemplified in Figure 6, which shows the distribution of the target path length before and after canonization. This huge amount of unnecessarily generated paths motivates us to pick 2 or 4 as values for

Table IV. Fraction of Target Paths Removed After a Path Equivalence Test (without a schema)

length	target paths	canonical	rem. rate
1	10974	3799	65.38%
2	17500	3344	80.89%
3	21929	2318	89.43%
4	27212	1584	94.18%
5	27379	784	97.14%
6	29771	491	98.35%
7	28942	280	99.03%
8	29178	170	99.42%
9	24349	73	99.70%
10	21492	28	99.87%
Total	241105	15250	93.67%

Table V. Fraction of Target Paths Removed After a Path Equivalence Test (without a schema)

length	key paths	canonical	rem. rate
1	1505	681	54.75%
2	638	130	79.62%
3	421	38	90.97%
4	436	20	95.41%

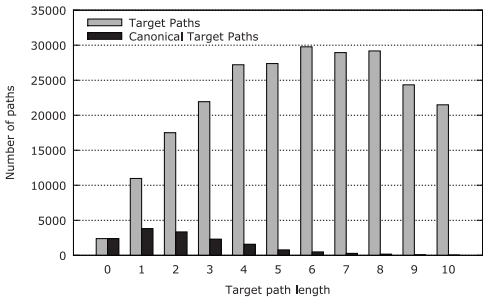


Fig. 6. Spread of the target paths and their canonical versions for a support threshold of 1 and a maximal length of 10.

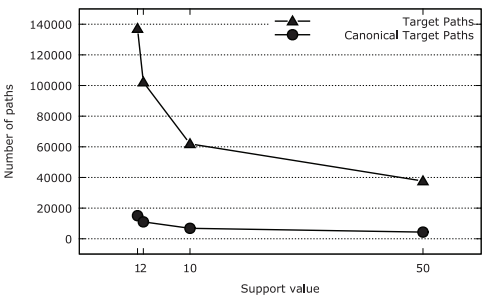


Fig. 7. Total amount of target paths found for support thresholds 1, 2, 10, and 50 and a maximal length of 6.

k_{max}. Finally, we think that, together with all possible contexts, paths of length up to 4 will provide enough freedom for selecting nodes in the XML document.

For key paths, we make similar observations. An inspection of the percentage of key paths that are pruned away using path equivalence reveals the numbers in Table V. These observations motivate us to restrict to maximal key path lengths of 2 or 4.

Next, we derive a suitable value for the *support threshold*. In Figure 7, we see the effects of an increasing support threshold on the number of target paths and on the number of canonical target paths. For larger values, the number of target paths passing the support threshold stabilizes quickly. This indicates that a large number of paths only select a few target nodes and that even small support thresholds will prune away significant parts of the XML key search space. To decide on a good support value, we should strike a balance between removing paths with low support while still keeping paths that select a significant portion of small documents. Indeed, small documents can yield low support values for a large portion of paths. For this reason, we mostly use support values of 2 and 10.

Finally, based on the observations made, we may already conclude that path equivalence without a schema, in conjunction with our canonisation algorithm, is an effective way of avoiding an explosion of paths.

Prefix tree. As different parts of the algorithm can avoid access to the input document t by operating directly on PT(t), it is instrumental to investigate the compression rate of PT(t) over t. Figure 8 plots the number of nodes in documents versus the number of nodes in the corresponding prefix trees. Note the scale is logarithmic. In essence, every

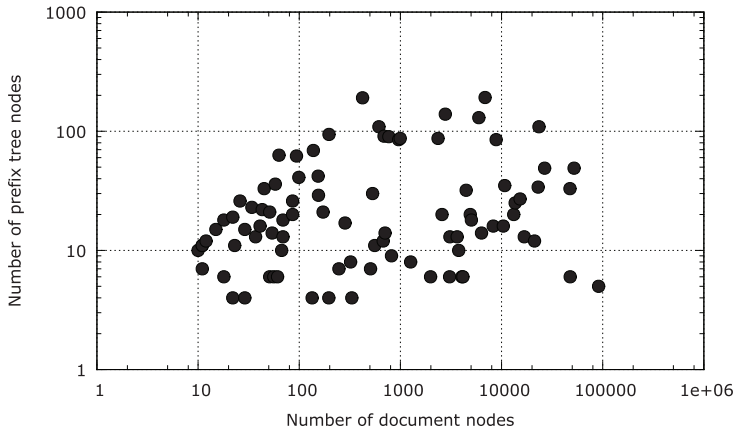


Fig. 8. Number of document nodes versus number of nodes in prefix trees. The prefix trees are considerably smaller than their full-sized counterparts.

document is compressed to a prefix tree with at most 200 nodes, even large documents containing up to 91K nodes.

Contexts. A key $\phi = (c, \tau, P)$ consists of three interdependent components: target paths need only be considered with respect to a context, and key paths need only be considered with respect to a context and a target path. To avoid an explosion of the size of the search space, it is paramount to reduce the number of considered contexts, target paths, and key paths. We next assess the effectiveness of the algorithm in this respect.

We start with the number of contexts considered by the algorithm. An analysis comparing the number of contexts allowed by XSDs with those actually used in the XML documents shows that, for 40% of the documents, all allowable contexts materialize in the corresponding XML documents, that is, there is no improvement as no allowable context can be omitted. Nevertheless, it appears that this mostly happens for smaller XSDs. Indeed, the total sum of allowable contexts over all 90 documents is 4,639, while the total sum of contexts found in actual documents is 2,217, indicating that over the complete dataset 52% of all possible contexts do not have to be considered. Keeping in mind that every context that can be removed in this step eliminates a call to the target path *and* key path miner underlines the effectiveness of context search driven by the XML data at hand.

Target paths. Next, we discuss the behavior of the target path miner when the support threshold N equals 10. The results are illustrated in Figure 9 (cases with $k_{max} = 5$ and/or lower support threshold were also tested but are similar and therefore not shown). For presentation purposes, the x-axis enumerates all document-XSD pairs increasingly ordered by the size of the XSD. The figure then shows, per pair, the number of candidate, supported, and nonequivalent derived target paths. Its purpose is to provide a visual inspection of the considered quantities on a per-document basis. By candidate target paths we mean those that occurred in a candidate set C_i during the execution of Algorithm 3, nonequivalent target paths are those which remain after duplicate elimination (as explained in Section 5.2). The number of possible target paths to consider (that is, the cardinality of the search space U times the number of allowable contexts) is not shown as the target path miner only considers a small fraction thereof (to be precise, only 3% on average). Furthermore, on average, only 7% of all candidate target paths turn out to be supported and of all supported paths only 27% remain after

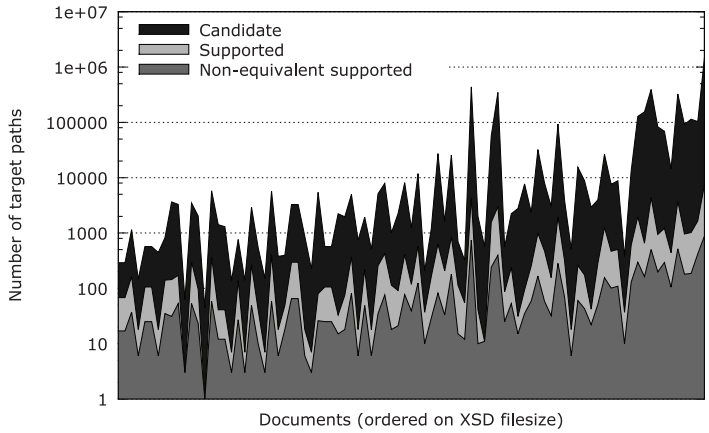


Fig. 9. Behavior of the target path miner. Number of candidate, supported, and nonequivalent target paths per document for $k_{max} = 4$ and support threshold of 10.

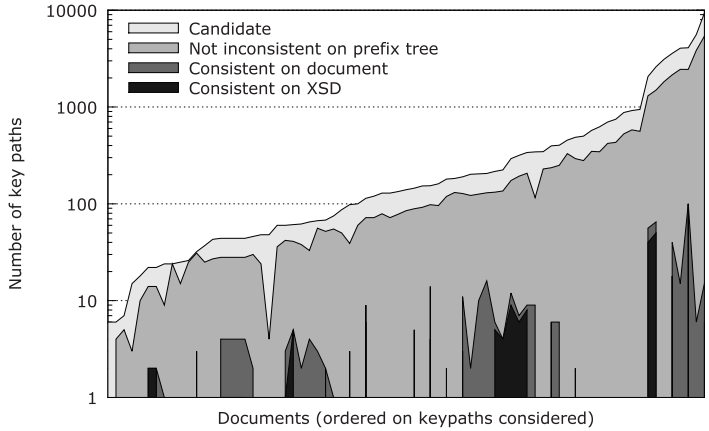


Fig. 10. Behavior of one-key path miner for support threshold 10, max target path length 4, and max key path length 2.

duplicate elimination. To get a feeling for the magnitude of the reduction in Target Paths (TPs) provided by the algorithm, Table VI shows the absolute numbers, which are summed up over the whole dataset of document-XSD pairs.

One-key paths. Figure 10 provides a visual interpretation of the reduction in number of key paths by the consecutive steps of the one-key path miner as described in Section 5.3. Again, for presentation purposes, the x-axis enumerates all document-XSD pairs ordered increasingly by the number of resulting candidate key paths. Specifically, the figure plots on a per-document basis the following numbers: candidate key paths, paths for which the inconsistency test fails on the prefix tree, paths that are consistent on the document, and paths consistent with respect to the XSD. We first discuss the average improvement on a per-document basis. Specifically, on average, 39% of candidate paths are inconsistent over the prefix tree. This means that, for 61% of the remaining key paths, consistency needs to be tested on the document. On average, only 6% of key paths are consistent with respect to the document, and of these 68% turn out to be consistent with respect to the XSD. Table VII shows the absolute numbers

Table VI. Number of Target Paths in the Search Space and Those that are Effectively Considered in Different Stages of the Target Path Miner

possible TPs	2.4×10^{11}
candidate TPs	6.7×10^{6}
supported TPs	8.4×10^{4}
unique TPs	1.3×10^{4}

The values shown are summed up over the entire dataset.

Table VII. Number of Key Paths (KPs) that are Effectively Considered in Different Stages of the One-Key Path Miner

candidate KPs	48144
inconsistent KPs on prefix tree	29190
consistent KPs on document	484
consistent KPs on XSD	288

The values shown are summed up over the entire dataset.

summed up over the whole dataset of document-XSD pairs, to give an indication of the effectiveness of the different stages of the one-key path miner.

It is interesting to observe that on the considered sample of real-world documents, consistency on the document does not always imply consistency with respect to the associated XSD. Specifically, Table VII shows that, overall, only roughly 60% of key paths consistent on documents are consistent on the XSD as well.

Keys. Next, we discuss the keys returned by our algorithm. We use the hypergraph transversal algorithm to mine relational functional dependencies as, for instance, described in Mannila and Räihä [1991], but any such algorithm can be readily plugged in. We consider keys with target path length at most 4 and key path length at most 2. In the following, we refer to testing the consistency of a key with respect to its XSD, that is, by applying the algorithm of Theorem 4.6, as the schema consistency test. Tables VIII and IX then gather some statistics of discovered keys, both without and with the schema consistency test. First of all, it can be observed that not every document contains a key with the required support: only 30% and 16% of all documents using support 10 and 100, respectively (Table VIII). The latter might seem strange at first sight, but note that not all XML documents are in fact databases and that the requirement for a key to qualify (recall, Definition 3.4) is a severe one. Indeed, even lowering the support threshold to a value of two (experiment not shown here) only provides a key for 60% of the documents but, of course, a key with support two is not very relevant. We note that the average support for discovered keys in this section is 404 and 612 for support thresholds equal to 10 and 100, respectively, while the maximum support encountered is 2011, indicating that the discovered keys indeed cover a large number of elements.

The figures in the two tables nicely illustrate the effectiveness of schema consistency as a quality measure. Indeed, without schema consistency, Table VIII shows that 107 and 54 keys are derived for support thresholds 10 and 100, respectively. Interestingly, in both cases there is a document with a rather large number of keys: 23, to be specific. But after the schema consistency test, each of these keys is removed as they all contain a key path that selects elements declared optional in the schema. Of course, one could debate about whether the schema is actually always correct or may be too liberal. One could always opt to offer keys that do not pass schema consistency to the user. However, after an inspection of the derived keys from our corpus, it becomes apparent that in many cases keys rejected by the schema are probably not keys at all. As an illustrative example, consider the following three derived keys (all with support 340, and where *root* refers to the root context).

(root, `./Products`, (`./ID`))

(root, `./Products`, (`./Other_Information`, `./Catalogue-Name`))

(root, `./Products`, (`./Type`, `./Other_Information`))

Table VIII. Statistics of Mined Keys for Support Thresholds 10 and 100 *Without* the Requirement to be Consistent with Respect to the Associated XSD

	sup10	sup100
derived keys	107	54
docs with keys	27	15
avg nr. of keys per doc	4	3.6
max nr. of keys per doc	23	23
avg nr. of key paths	1.3	1.3
max key nr. of key paths	2	2

Table IX. Statistics of Mined Keys for Support Thresholds 10 and 100 *With* the Requirement to be Consistent with Respect to the Associated XSD

	sup10	sup100
derived keys	43	16
docs with keys	19	10
avg nr. of keys per doc	2.2	1.6
max nr. of keys per doc	9	4
avg nr. of key paths	1.3	1.2
max key nr. of key paths	2	2

After the schema consistency test, only the first key remains. In this case, it should be clear that the second and third keys are not accurate but are glitches in the data. Therefore, one could say that the reduction from 107 to 43 and from 54 to 16 keys in Tables VIII and IX actually improves the quality at the expense of lowering the quantity, which, in our opinion, can be seen as a good thing as most data mining problem suffer an explosion in derived patterns.

Quality. It remains to discuss the quality of the keys. When the provided schema is accurate, the schema consistency test as discussed previously, provides a quality criterion in its own. A second quality criterion can be the high support of derived keys: as mentioned earlier, the found support of derived keys is on average 404 and 612 for support thresholds equal to 10 and 100, respectively, while the maximum support encountered is 2011. Furthermore, when inspecting found keys it appeared that in many cases keys select elements whose name contains "ID".

We finish with a discussion on implication of keys. Usually, in key discovery, the goal is to find a minimal set of keys, called cover, from which all other keys can be derived. For instance, to this end Grahne and Zhu [2002] make use of the inference algorithms for XML keys investigated and shown polynomially computable by Buneman et al. [2003]. Unfortunately, Theorem 4.32 shows that key implication in the presence of a schema is EXPTIME-hard. Still, there is opportunity to detect duplicate keys. For instance, the next pair of discovered keys turn out to be equivalent (both with support 90).

((State: 188,Symbol: `ConstraintID`),`./*`,(`./*`))

((State: 167,Symbol: `PureOrMixtureData`), `./Constraint/ConstraintID/*`,(`./*`))

ConstraintID can only occur under a `Constraint` element. We can therefore consider the keys equivalent as they select precisely the same set of target nodes.

Running time. We next discuss the running time of the algorithm. Of course, the previous sections have already illustrated how the different mining steps succeed in reducing the number of considered contexts, target paths, and key paths and how every such reduction induces a gain in speed. Figure 12 gives insight on the overall running time. Here, we can see that a large fraction of the time is taken up by the schema consistency test. Furthermore, Figure 11 gives an indication of the proportion of time taken by the schema consistency test with respect to the overall running time. For presentation purposes, the x-axis enumerates all document-XSD pairs ordered increasingly by the time required for the schema consistency test. Note that the figure does not imply an exponential growth of the running time. In fact, as the x-axis does not correspond to a quantity, no inference can be made about the asymptotic growth of the running time.

To get more insight on what part of the input controls the running time, we checked several metrics of the data and the schema's:

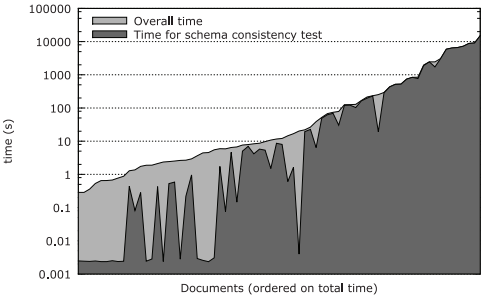


Fig. 11. Part of the overall running time spent on the schema consistency test; support threshold 2, max target path length 2, and max key path length 2.

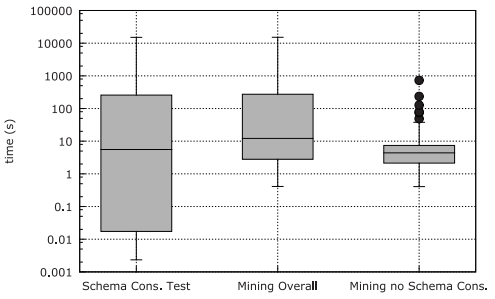


Fig. 12. Boxplots indicating average running times for schema consistency by itself, the entire mining algorithm, and mining without schema consistency; support threshold 2, max target path length 2, and max key path length 2, The plots presented use 1.5× the inter-quartile range for determining the whiskers. The black dots are outliers, each depicting a document-XSD pair.

—XML metrics as the number of nodes, depth, average children, number of labels, number of leafs, and number of prefix tree nodes;
—XSD metrics as the number of labels, number of states, to and number of contexts.

For the target miner, we found the number of nodes in the prefix tree to be correlated to the running time and the number of target paths. This is as expected, since the prefix tree is used for support calculation and equivalence tests; both are used continuously during in this part. For the key miner, we found that, without XSD consistency tests, the same correlation is observed. Also, there is a correlation between the number of labels used in the document and the running time. Sadly, none of these metrics showed a clear connection to the running time when XSD consistency is used. We stress that key discovery is not a time-critical task and that the algorithm only has to be run once for an XML document and XSD. Nevertheless, the figures also show that the most room for improvement lies within a speedup of the schema consistency test as opposed to other components of the algorithm.

Optimizations. The execution of the algorithm can be tailored by switching several optimizations on or off. In this section, we take a look at some key optimizations and their effect on the running time.

We first focus on optimizations of the target miner. One of the most important optimizations is the *duplicate elimination* as described in Section 5.2. We consider three options: (1) no duplicate elimination; (2) duplicate elimination without a schema; and (3) using duplicate elimination with a schema. Figure 13 shows the running times for each of these, per document, ordered by (2). Option (1) is the fastest in almost every case, while (3) is the slowest in almost every case, sometimes even by several orders of magnitude. Notice that the time for options (2) and (3) is bounded by 10 seconds.

Although (1) may seem the best option for the target miner, it will produce a very large set of target paths (see the preceding), each of them invoking a new run of the key path miner. Figure 14 shows the running time of the key miner for the same documents, this time ordered by the default key miner time. When we compare the default time to the time where duplicate elimination of the target miner is switched off, we see the running time increase a lot, sometimes even by an order of magnitude. Note that the key path miner typically takes up a lot more time, whence it is advised to retain the duplicate elimination check (2).

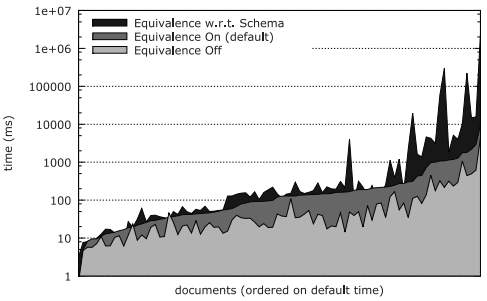


Fig. 13. Per-document running time of equivalence optimizations in the target miner; XSD consistency off, support threshold 10, max target path length 4.

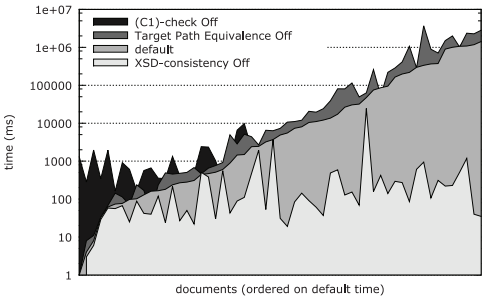


Fig. 14. Per-document running time obtained by switching off different optimizations of the key path miner; support threshold 2, max target path length 2, max key path length 2.

As we have seen before, the XSD consistency test takes up a large portion of the running time. When switching this test off and resorting to XML consistency, we see that the running time improves by several orders of magnitude (Figure 14). Because of the applicability of the XSD consistency measure as a quality measure, however, we conclude that future work on the key miner software should make optimizing this part a priority.

7. DISCUSSION

In this article, we initiated a fundamental study of properties of W3C XML keys in the presence of a schema and introduced an effective novel key mining algorithm leveraging on the formalism of levelwise search and on algorithms for the discovery of functional dependencies in the relational model.

A number of interesting issues remain open and require further investigation. The most direct one is to close the gaps between some of the obtained lower and upper bounds. It would be interesting to investigate tractable subcases, especially with respect to key implication. An observed bottleneck of the proposed approach is to check consistency of a derived key with respect to the associated schema, even though the number of keys that have to be tested is greatly reduced by testing for inconsistency on the XML document, it should be investigated how schema consistency can be accelerated. This would require advances in string and tree automata theory. Another approach would be to try to find fast heuristic algorithms, or to study the problem for subclasses of XSDs.

Another possible direction would be to investigate how the mining framework could be extended to top-level union in keys. It would be especially important to avoid an explosion of the size of the search space. The latter would also require to find heuristics for consistency testing in the presence of disjunction and a schema, as this problem is CONP-hard.

It would also be interesting to see how the present framework can be extended to discover approximate keys. For this, we need a measure f over multisets that expresses how closely a multiset resembles a set. Then the *confidence* of the key $\phi = (c, \tau, P)$ can, for instance, be obtained by $\text{agg}_{v \in \text{CNodes}_t(c)}\, f(\{\text{record}_P(u) \mid u \in \tau(t, v)\})$, where agg is an aggregate operator (as, e.g., sum). Our framework would then allow to plug in any algorithm for deriving relational approximate functional dependencies.

ELECTRONIC APPENDIX

The electronic appendix to this article can be accessed in the ACM Digital Library.

REFERENCES

Serge Abiteboul, Yael Amsterdamer, Daniel Deutch, Tova Milo, and Pierre Senellart. 2012. Finding optimal probabilistic generators for XML collections. In *Proceedings of the 15th International Conference on Database Theory (ICDT'12)*. ACM Press, New York, 127–139.

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

Marcelo Arenas, Jonny Daenen, Frank Neven, Martin Ugarte, Jan Van den Bussche, and Stijn Vansummeren. 2013. Discovering XSD keys from XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 61–72.

Marcelo Arenas, Wenfei Fan, and Leonid Libkin. 2002. What's hard about XML schema constraints? In *Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA'02)*. Lecture Notes in Computer Science, vol. 2453, Springer, 269–278.

Denilson Barbosa and Alberto O. Mendelzon. 2003. Finding id attributes in XML documents. In *Proceedings of the 1st International XML Database Symposium (XSym'03)*. Lecture Notes in Computer Science, vol. 2824, Springer, 180–194.

Geert Jan Bex, Wouter Gelade, Wim Martens, and Frank Neven. 2009. Simplifying XML schema: Effortless handling of nondeterministic regular expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. 731–744.

Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. 2010a. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Trans. Web* 4, 4.

Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. 2010b. Inference of concise regular expressions and dtds. *ACM Trans. Database Syst.* 35, 2.

Geert Jan Bex, Frank Neven, and Stijn Vansummeren. 2007. Inferring XML schema definitions from XML data. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*. 998–1009.

Geert Jan Bex, Frank Neven, and Stijn Vansummeren. 2008. SchemaScope: A system for inferring and cleaning XML schemas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 1259–1262.

Dina Bitton, Jeffrey Millman, and Solveig Torgersen. 1989. A feasibility and performance study of dependency inference. In *Proceedings of the International Conference on Data Engineering (ICDE'89)*. 635–641.

Henrik Björklund, Wim Martens, and Thomas Schwentick. 2013. Validity of tree pattern queries with respect to schema information. In *Proceedings of the 38th International Symposium on Mathematical Foundations of Computer Science (MFCS'13)*. Lecture Notes in Computer Science, vol. 8087, Springer, 171–182.

Mikolaj Bojanczyk. 2008. Tree-walking automata. In *Proceedings of the 2nd International Conference on Language and Automata Theory and Applications (LATA'08)*. Lecture Notes in Computer Science, vol. 5196, Springer, 1–2.

Anne Bruggemann-Klein and Derick Wood. 1998. One-unambiguous regular languages. *Inf. Comput.* 140, 2, 229–253.

Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. 2002. Keys for XML. *Comput. Netw.* 39, 5, 473–487.

Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. 2003. Reasoning about keys for XML. *Inf. Syst.* 28, 8, 1037–1063.

Stanislav Fajt, Irena Mlynkova, and Martin Necasky. 2011. On mining xml integrity constraints. In *Proceedings of the 6th IEEE International Conference on Digital Information Management (ICDIM'11)*. 23–29.

Wenfei Fan and Leonid Libkin. 2002. On XML integrity constraints in the presence of dtds. *J. ACM* 49, 3, 368–406.

Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, Sridhar Seshadri, and Kyuseok Shim. 2003. XTRACT: Learning document type descriptors from XML document collections. *Data Min. Knowl. Discov.* 7, 1, 23–56.

Gösta Grahne and Jianfei Zhu. 2002. Discovering approximate keys in XML data. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM'02)*. 453–460.

Steven Grijzenhout and Maarten Marx. 2010. University of amsterdam XML web collection. http://data.politicalmashup.nl/sgrijzen/xmlweb/.

Sven Hartmann and Sebastian Link. 2009. Efficient reasoning about a robust XML key fragment. *ACM Trans. Database Syst.* 34, 2.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2003. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

Donald E. Knuth, James H. Morris Jr., and Vaughan R. PRATT. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6, 2, 323–350.

Heiki Mannila and Kari-Jouko Räihä. 1989. Practical algorithms for finding prime attributes and testing normal forms. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS'89)*. ACM Press, New York, 128–133.

Heikki Mannila and Kari-Jouko Räihä. 1991. *The Design of Relational Databases*. Addison-Wesley.

Heikki Mannila and Kari-Jouko Räihä. 1994. Algorithms for inferring functional dependencies from relations. *Data Knowl. Engin.* 12, 1, 83–99.

Heikki Mannila and Hannu Toivonen. 1997. Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.* 1, 3, 241–258.

Wim Martens, Frank Neven, and Thomas Schwentick. 2007. Simple off the shelf abstractions for XML schema. *SIGMOD Rec.* 36, 3, 15–22.

Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. 2006. Expressiveness and complexity of XML schema. *ACM Trans. Database Syst.* 31, 3, 770–813.

Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. 2005. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.* 5, 4, 660–704.

Martin Necaský and Irena Mlýnkova. 2009. Discovering XML keys and foreign keys in queries. In *Proceedings of the ACM Symposium on Applied Computing (SAC'09)*. ACM Press, New York, 632–638.

Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* 3rd Ed. McGraw-Hill.

Helmut Seidl. 1990. Deciding equivalence of finite tree automata. *SIAM J. Comput.* 19, 3, 424–437.

Richard Edwin Stearns and Harry B. Hunt Iii. 1985. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM J. Comput.* 14, 3, 598–611.

Larry J. Stockmeyer and Albert R. Meyer. 1973. Word problems requiring exponential time: Preliminary report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC'73)*. 1–9.

Peter Van Emde Boas. 1997. The convenience of tilings. In *Complexity, Logic, and Recursion Theory,* Marcel Dekker, 331–363.

W3C. 2004. *XML Schema Part 1: Structures* 2nd Ed. http://www.w3.org/TR/xmlschema-1/#cIdentity-constraint

Cong Yu and H. V. Jagadish. 2008. XML schema refinement through redundancy detection and normalization. *VLDB J.* 17, 2, 203–223.

Received October 2013; revised April 2014; accepted June 2014