Discovering XSD Keys from XML Data

Marcelo Arenas
PUC Chile &
University of Oxford
marenas@ing.puc.cl

Jonny Daenen
Hasselt University &
Transnational University of Limburg
jonny.daenen@uhasselt.be

Frank Neven
Hasselt University &
Transnational University of Limburg
frank.neven@uhasselt.be

Martin Ugarte
PUC Chile
mgugarte@uc.cl

Jan Van den Bussche
Hasselt University &
Transnational University of Limburg
jan.vandenbussche@uhasselt.be

Stijn Vansummeren
Université Libre de Bruxelles (ULB)
stijn.vansummeren@ulb.ac.be

ABSTRACT

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 paper 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 above mentioned properties 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*

Keywords

XML key mining

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., [22]). 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 [6, 7, 8, 9, 10, 18], the automatic inference of constraints has been left largely unexplored (we refer to Section 2 for a discussion on related work). In this paper, 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 [31]. 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 paper is devoted to a study of the complexity of testing these requirements.

EXAMPLE 1.1. *Consider the key,*

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

Here, the pair $(\textit{order}, q_{\texttt{order}})$ *is a* context *consisting of the label 'order' and the state or type* $q_{\texttt{order}}$[1]*, which 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* $R_{\phi,t}$*:*

$(\textit{order}, q_{\texttt{order}})$	`.//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*

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

holds in $R_{\phi,t}$*. That is, within the same context node, 'title' and 'year' uniquely determine the 'book' element.*

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

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

As a necessary condition for a key to be valid on a tree t, the XML key specification ([31, Section 3.11.4]) requires every key path to always select precisely one node 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* w.r.t. 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 (which 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 can not 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 paper is that of reasoning about keys. On the negative side, and in strong contrast to reasoning about relational keys [3, 29] or XML keys without an accompanying schema [14, 15], 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 above theoretical groundwork, 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 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 Manilla and Toivonen [24]) 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 w.r.t. 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 that 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 can not hold anymore given already obtained information.

We define a target path miner within the above framework 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 to be optimal w.r.t. the considered partial order. Furthermore, the search predicate can be solely evaluated on a much smaller prefix tree representation of the input document and, therefore, does not need access to the original document. In addition, we define a one-key path miner which searches for all consistent single key paths p (w.r.t 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 (w.r.t. 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 can not 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 above, a functional dependency miner can be used to determine the corresponding XML key (e.g., [11, 21, 23])).

Contributions. We make the following contributions:
(*i*) We characterize the complexity of the consistency problem for XML keys w.r.t. an XSD for different classes of target and key paths (Theorem 4.4). 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 1). In addition, we characterize the complexity of boundedness, satisfiability, universality, and implication of XML keys (Theorem 4.5) as well as equivalence of target paths (Theorem 4.6).

(*ii*) We develop a novel key mining algorithm leveraging on algorithms 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)

(*iii*) 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. 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. In Section 6, we experimentally validate our algorithm. We conclude in Section 7.

2. RELATED WORK

XML Keys. One of the first definitions of keys for XML was introduced by Buneman et al. [14, 15]. These keys are of the form $(Q, (Q', P))$ where Q is the context-path, Q' is the target path and P is 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

[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 paper can be easily extended to include attributes.

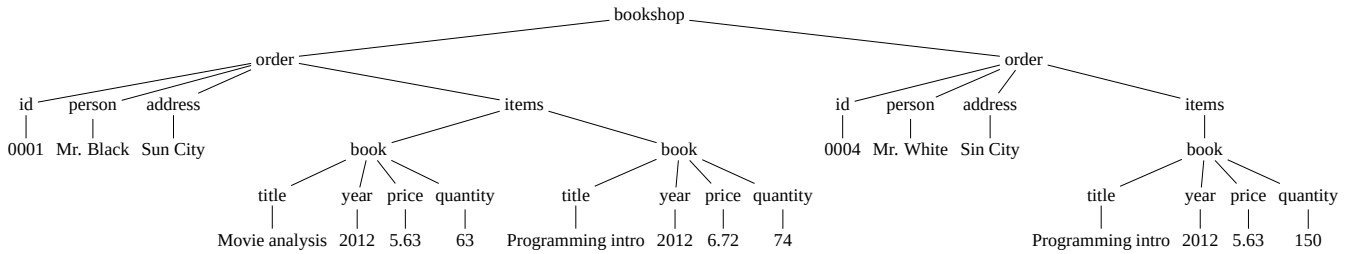

Figure 1: An example XML tree (order and book nodes are named o_1, o_2 and b_1, b_2, b_3 from left to right).

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 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 paper 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 [14, 15, 20]. Unfortunately, these results do not carry over to W3C keys as the latter are defined w.r.t. an XML Schema but the former are not.

Decision problems in the presence of a schema. A number of consistency problems of XML keys w.r.t. a DTD have been considered by Fan and Libkin [17]. 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 paper. 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 [17] contains many more results on the interplay between keys, foreign keys, inclusion dependencies and DTDs. Arenas et al. [4] discuss satisfiability[3] of XML keys w.r.t. a DTD. The result most relevant to the present paper is NP-hardness of satisfiability w.r.t. a non-recursive 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 [19] considered mining of approximate keys and proposed an Apriori style algorithm which uses the inference rules of [15] for optimization. Necaský and Mlýnková [28] ignore the XML data but present an approach to infer keys and foreign keys from element/element joins in XQuery logs. Fajt et al [16] consider the inference of keys and foreign keys building further on algorithms for the relational model. The above algorithms can not 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 [32] 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 w.r.t. a selector expression as opposed to w.r.t. a DFA. For these reasons, W3C keys can not be encoded as a special case of FDs. Barbosa and Mendelzon [5] proposed algorithms to find ID and IDREFs attributes in XML documents. They show that the natural decision problem associated to this discovery problem is NP-complete, and present a heuristic algorithm. Abiteboul et al. [2] consider probabilistic generators for XML collections in the presence of integrity constraints but do not consider mining of such constraints.

[3]We note that satisfiability is called consistency in [4].

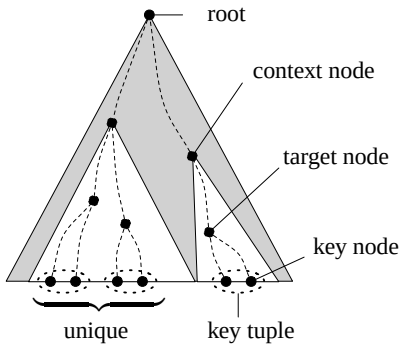

Figure 2: Schematic representation of a key.

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 that trees are sibling-ordered. We fix a finite set of element names Σ and an infinite set $\mathbf{Data}$ of data elements. An *XML-tree* is a $(\Sigma \cup \mathbf{Data})$-tree where non-leaf nodes are labeled with Σ and leaf nodes are labeled with elements from $(\Sigma \cup \mathbf{Data})$. As the XSD specification does not allow mixed content models for fields in keys [31], we ignore 'mixed' content models altogether to simplify presentation, and assume that when a node is labeled with a $\mathbf{Data}$-element it is the only child of its parent. We then

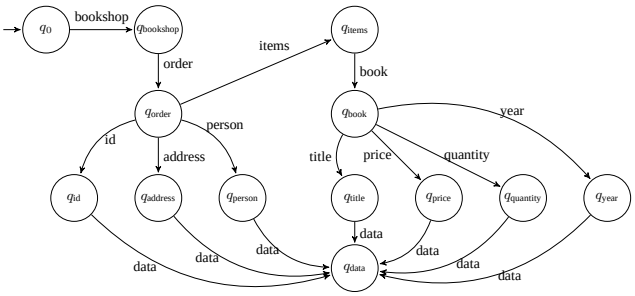

Figure 3: The type automaton of X_{bookshop}.

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. [25]. 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 $q \in \text{Types}$, $\mathit{Out}(q) = \mathit{Symb}(\lambda(q))$, where $\mathit{Out}(q) = \{\sigma \in \Sigma \mid \delta(q, \sigma) \text{ is defined}\}$ and $\mathit{Symb}(r)$ consists of all Σ-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.

- $\text{value}_t(v) \in \textbf{Data}$ 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 $\lambda(q)$. We stress that this DFA-based characterization of XSDs corresponds precisely to the more traditional abstraction in terms of single-type grammars [26, 27]. We let $\mathcal{L}(X)$ denote the set of all trees adhering to XSD X.

[4]Also referred to as 1-unambiguous regular expressions [13].

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

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

For all other types q, $\lambda(q) =$ `data`.

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

3.3 XML keys

A *selector expression* is a restricted XPath-expression of one of the three forms '.' (the dot symbol) or $./l_1/l_2/\ldots/l_k$ (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 $\tau(t, v)$ is defined as follows. If $\tau =$ '.', then $\tau(t, v) = \{v\}$. Otherwise τ is of the form either $./l_1/l_2/\ldots/l_k$ or $.//l_1/l_2/\ldots/l_k$, and $\tau(t, v)$ 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.

DEFINITION 3.3. *An* XML key, *defined w.r.t. 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 w.r.t. X, we sometimes write a key simply as a pair (ϕ, X).

We stress that the definition of XML keys given above, corresponds to the definition of keys in XML Schema [31]. 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* [31, 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 which 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 those trees, i.e., 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 above in Figure 2.

Note that $p_j(t, v)$ might select more than one node or might select a node u for which $\text{value}_t(u)$ is undefined; both are disallowed by the XML Schema specification:

DEFINITION 3.4. *A key $\phi = (c, \tau, P)$* qualifies *in a document t if for every $v \in \mathrm{CNodes}_t(c)$, 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 w.r.t. a document:

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

Notice that, 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 w.r.t. 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 cause is semantical and more informative.

In this paper, 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 w.r.t. 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 w.r.t. the tree in Figure 1 but ϕ is not consistent w.r.t. $X_{bookshop}$. 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 w.r.t. t.[5] We define the support of a key as the quantity measuring the number of nodes captured by the key. Define $\mathrm{TNodes}_t(\phi)$ as the set of target nodes selected by $\phi = (c, \tau, P)$ on t. That is,

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

Then, following Grahne and Zhu [19], we define the *support* of ϕ on t to be the total number of selected target nodes: $\mathrm{supp}(\phi, t) = |\mathrm{TNodes}_t(\phi)|$. Since this support only depends on the context c and the target path τ of ϕ, we also write $\mathrm{supp}(c, \tau, t)$ for $\mathrm{supp}(\phi, t)$.

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

DEFINITION 3.8. **(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 $\mathrm{supp}(\phi, t) > N$.*

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

4. BASIC 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 which can then be removed from the output of the key mining algorithm. Specifically, we consider testing for consistency and show that the problem becomes tractable when top-level disjunction is disallowed. We also study universality and boundedness, and show that they are tractable. Finally, we show that testing for satisfiability and implication of keys is EXPTIME-hard, even when disallowing disjunction, which complicates the inference of minimal keys.

[5]W.l.o.g. 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.

$\mathcal{P}$	$\forall_{\text{tree}}^{>k,\mathcal{P}}$	$\forall_{\text{tree}}^{<k,\mathcal{P}}$	$\forall_{\text{tree}}^{=k,\mathcal{P}}$
$\mathcal{RE}$	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

Table 1: Complexity of $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$.

4.1 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 w.r.t. 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 paper. 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 should 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 $\forall_{\text{tree}}^{\bullet k}$ 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 4.3 that CONSISTENCY can be easily reduced to $\forall_{\text{tree}}^{=1}$. Although Bjorklund, Martens, and Schwentick [12] showed that $\forall_{\text{tree}}^{>k}$ is EXPTIME-complete, we obtain below that $\forall_{\text{tree}}^{=1}$ can in fact be solved in polynomial time through an intricate translation to the equivalence test for unambiguous tree automata [30].

4.1.1 Cardinality of XPath result sets

Because of its relevance to cardinality estimation of XPath result sets, we investigate in more detail the complexity of $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ and it restriction to strings, denoted by $\forall_{\text{string}}^{\bullet k,\mathcal{P}}$, relative to the XPath-fragment $\mathcal{P}$.

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 $\mathcal{SE}$ the set of all selector expressions and by $\mathcal{SE}^{//}$ and $\mathcal{SE}^*$, the set of all selector expressions *without* descendant and wildcard, respectively. For a class of patterns $\mathcal{P} \in \{\mathcal{RE}, \mathcal{DSE}, \mathcal{SE}, \mathcal{SE}^{//}, \mathcal{SE}^*\}$, we denote by $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ the problem $\forall_{\text{tree}}^{\bullet k}$ where expressions are restricted to the class $\mathcal{P}$.

THEOREM 4.1. *The complexity of the problem $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ is as stated in Table 1.*

Notice that the problem $\forall_{\text{tree}}^{=k,\mathcal{RE}}$ is PSPACE-hard for every value $k \geq 1$, while $\forall_{\text{tree}}^{=k,\mathcal{DSE}}$ is CONP-hard for every value $k \geq 1$. On

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

the other hand, $\forall_{\text{tree}}^{=0,\mathcal{RE}} = \forall_{\text{tree}}^{<1,\mathcal{RE}}$ and $\forall_{\text{tree}}^{=0,\mathcal{DSE}} = \forall_{\text{tree}}^{<1,\mathcal{DSE}}$ and, thus, these two problems can be solved in polynomial time given the results in the second column of Table 1. Below we provide a sketch of the proof that $\forall_{\text{tree}}^{=1,\mathcal{SE}}$ can be solved in polynomial time, which is a key problem in our study of consistency.

PROOF SKETCH OF $\forall_{\text{tree}}^{=1,\mathcal{SE}} \in$ PTIME. In this proof, we need an encoding of XML trees as binary trees. More precisely, for an XML tree t, denote by $\text{fcns}(t)$ a binary tree such that for every node v of t: (1) v is a node in $\text{fcns}(t)$; (2) the left child of v in $\text{fcns}(t)$ is the first child of v in t (if v is a leaf in t, then a node with label $\#$ is placed as the left child of v in $\text{fcns}(t)$); and (3) the right child of v in $\text{fcns}(t)$ is the next sibling of v in t (if such a sibling does not exist in t, then a node with label $\#$ is placed as the right child of v in $\text{fcns}(t)$). Moreover, in this proof we also make use of tree automata which operate in a top-down fashion over binary trees, which are called binary tree automata (BTA). Given a BTA A, we denote by $\mathcal{L}(A)$ the set of trees accepted by A, and we say that A is unambiguous if for every $t \in \mathcal{L}(A)$, there is only one accepting run of A on t, but there could many non-accepting ones.

In order to show that $\forall_{\text{tree}}^{=1,\mathcal{SE}} \in$ PTIME, we first need the following results: (1) given an XSD X, one can construct in polynomial time a deterministic BTA A_X such that for every tree t: $\text{fcns}(t) \in \mathcal{L}(A_X)$ if and only if $t \in \mathcal{L}(X)$; (2) given a selector expression p, one can construct in polynomial time a non-deterministic BTA B_p such that for every XML tree t: $\text{fcns}(t) \in \mathcal{L}(B_p)$ if and only if $|p(t)| > 0$; and (3) there is a deterministic BTA $A_\#$ such that $t' \in \mathcal{L}(A_\#)$ if and only if $t' = \text{fcns}(t)$ for some XML tree t. With these ingredients, the polynomial time algorithm for $\forall_{\text{tree}}^{=1,\mathcal{SE}}$ works as follows.

Let X be an XSD and p a selector expression. In order to test whether $(X,p) \in \forall_{\text{tree}}^{=1,\mathcal{SE}}$, we first verify whether $(X,p) \in \forall_{\text{tree}}^{<2,\mathcal{SE}}$, which can be done in polynomial time (see Table 1). If this is not the case, then we know that $(X,p) \notin \forall_{\text{tree}}^{=1,\mathcal{SE}}$, so the algorithm returns *false*. Otherwise, the algorithm continues by computing deterministic BTAs A_X, $A_\#$ and non-deterministic BTA B_p. Then to check whether $(X,p) \in \forall_{\text{tree}}^{=1,\mathcal{SE}}$, the algorithm needs to verify whether $\mathcal{L}(A_X \times A_\#) \subseteq \mathcal{L}(B_p)$, where $A_X \times A_\#$ is the usual product of BTAs that accepts $\mathcal{L}(A_X) \cap \mathcal{L}(A_\#)$ and can be computed in polynomial time. The key observations to make here are: (1) testing whether $\mathcal{L}(A_X \times A_\#) \subseteq \mathcal{L}(B_p)$ is equivalent to verifying whether $\mathcal{L}(A_X \times A_\#) \subseteq \mathcal{L}(A_X \times B_p)$; (2) containment for BTAs is an intractable problem, but it becomes tractable for unambiguous BTAs [30]; (3) $A_X \times A_\#$ is an unambiguous BTA as it is deterministic; and (4) although $A_X \times B_p$ is non-deterministic, by using the fact that $(X,p) \in \forall_{\text{tree}}^{<2,\mathcal{SE}}$, it is possible to prove that $A_X \times B_p$ is an unambiguous BTA. Therefore, we can test in polynomial time whether $\mathcal{L}(A_X \times A_\#) \subseteq \mathcal{L}(A_X \times B_p)$ and, thus, we can test in polynomial time whether $(X,p) \in \forall_{\text{tree}}^{=1,\mathcal{SE}}$. □

Finally, we consider the corresponding problem for strings as well. We denote by $\forall_{\text{string}}^{\bullet k,\mathcal{P}}$ the problem to decide whether, given a DFA A and a pattern $p \in \mathcal{P}$, $|p(s)| \bullet k$ for every $s \in \mathcal{L}(A)$. Here, as every string can be viewed as a unary tree, $p(s)$ simply denotes the nodes selected by p when evaluated from the root.

THEOREM 4.2. *The complexity of the problem $\forall_{\text{string}}^{\bullet k,\mathcal{P}}$ is as for $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$ with the exception that:*

1. *$\forall_{\text{string}}^{>k,\mathcal{RE}}$, $\forall_{\text{string}}^{>k,\mathcal{DSE}}$, $\forall_{\text{string}}^{>k,\mathcal{SE}}$ are* PSPACE*-complete and $\forall_{\text{string}}^{>k,\mathcal{SE}^*}$ is in* PTIME*;*
2. *$\forall_{\text{string}}^{=k,\mathcal{RE}}$ is* PSPACE*-complete for every $k \geq 1$; and*
3. *$\forall_{\text{string}}^{=k,\mathcal{DSE}}$ is* CONP*-complete for every $k \geq 1$.*

The results in Theorem 4.2 are important for our investigation not only because they can be used to obtain lower bounds for the complexity of the problems $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$, 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 $\forall_{\text{tree}}^{\bullet k,\mathcal{P}}$, most notably in the case of $\forall_{\text{tree}}^{=k,\mathcal{SE}}$.

4.1.2 Main result on consistency

For a class of patterns $\mathcal{P}$, we denote by CONSISTENCY($\mathcal{P}$) the problem CONSISTENCY restricted to keys using expressions in $\mathcal{P}$. We introduce the following 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 node u in $\tau(t,v)$.

Let *root* stand for the class containing only the selector expressions '.', that is, the expression which selects the root. The following lemma now allows to transfer upper and lower bounds from the previous section:

LEMMA 4.3. *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 $\forall_{\text{key}}^{\bullet k,root,\mathcal{P}}$.*

The main result of this section immediately follows from Theorem 4.1 and Lemma 4.3:

THEOREM 4.4.
1. CONSISTENCY($\mathcal{SE}$) *is in* PTIME*;*
2. CONSISTENCY($\mathcal{DSE}$) *is* CONP*-hard and in* EXPTIME*;*
3. CONSISTENCY($\mathcal{RE}$) *is* PSPACE*-hard and in* EXPTIME*;*

4.2 Determining the quality of keys

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, that only address a bounded number of target nodes, and keys that are implied by keys that have already been found.

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 w.r.t. 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 $\Psi \sqsubseteq \phi$, 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 $t \models \phi$;

Intuitively, a bounded key can only select a bounded number of target nodes independent of the size of the input document. Since the main purpose of a key is to ensure uniqueness of nodes within a collection of nodes, bounded keys are not very interesting.

We next 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 (aka, a cover) is practically infeasible. Note that, while the EXPTIME-completeness of SATISFIABILITY is discouraging, it does not pose a problem for key mining algorithms in practice. Indeed, by Definition 3.8 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). As such, the keys ϕ it returns are necessarily satisfiable.

Similar to the previous section, we parametrize the problems above by a class $\mathcal{P}$ of expressions, to restrict attention to input keys that only use expressions in $\mathcal{P}$.

THEOREM 4.5. *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.*

Next, we consider *target path containment* and *equivalence*. 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 $v \in \mathrm{CNodes}_t(c)$, $\tau(t, v) \subseteq \tau'(t, v)$. We denote the latter condition by $\tau \subseteq_{X,c} \tau'$. By TARGET PATH EQUIVALENCE we denote the corresponding equivalence problem.

THEOREM 4.6. TARGET PATH CONTAINMENT *and* TARGET PATH EQUIVALENCE *are in* PTIME.

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, but which can be solved efficiently.

5. XML KEY MINING ALGORITHM

In this section, we provide an algorithm for solving the XML key mining problem. Recall from Definition 3.8 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 X, are satisfied by t, and whose support exceeds N[7]. For the remainder, let $X = (A_X, \lambda_X)$ with the type-automaton $A_X = (\mathrm{Types}, \Sigma \cup \{\texttt{data}\}, \delta, q_0)$.

The overall structure of the XML key mining algorithm is outlined in Algorithm 1.

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)
```

Basically the algorithm consists of four components:

- $\mathrm{ContextMiner}_{t,X}$ returns a list of possible contexts based on t and X;
- $\mathrm{TargetPathMiner}_{t,X}(c)$ returns a list of unique target paths with minimal support in t given a context c;

[7]If no XSD is available, one can be derived, e.g., using algorithms from [9].

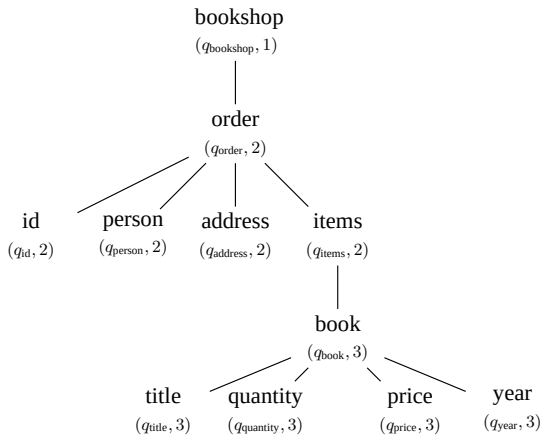

Figure 4: Prefix tree for the XML tree in Figure 1.

- $\mathrm{OneKeyPathMiner}_{t,X}(c, \tau)$ returns a maximal set S of unique key paths for which $(c, \tau, \{p\})$ is consistent for every $p \in S$; and,
- $\mathrm{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$ returns a set $\mathcal{P}$ of minimal subsets P of S for which $t \models (c, \tau, P)$.

$\mathrm{TargetPathMiner}_{t,X}(c)$ and $\mathrm{OneKeyPathMiner}_{t,X}(c, \tau)$ are different instantiations of levelwise search [24], while the function $\mathrm{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$ 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 (cf. Theorem 4.4), we focus on a key mining algorithm that disregards the union operator.

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 $\mathrm{PT}(t)$ 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) that node. Let h be the function mapping each node in t to its corresponding node in $\mathrm{PT}(t)$. Then, we label every node m in $\mathrm{PT}(t)$ with the number of nodes in t mapped to m, i.e., $|h^{-1}(m)|$ together with the state assigned to m by the type-automaton A_X. For example, the prefix tree for the XML tree in Figure 1 is shown in Figure 4. Note that $\mathrm{PT}(t)$ does not contain data nodes. The prefix tree can be computed in time linear in the size of t (see, e.g., [19]).

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

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 [24]. In brief, the algorithm is of a generate-and-test style that starts from the most general target path, .//*

Algorithm 2 Basic algorithm for levelwise search [24]

$C_0 :=$ 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(F_i);

Algorithm 3 TargetPathMiner$_{t,X}(c)$

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

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 $q(\tau) := \mathrm{supp}(c, \tau, t) > N$, 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 4.2). Although this containment relation is shown to be tractable (Theorem 4.6), 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 which requires fast testing of specialization due to the large number of such tests. In strong contrast, as we show below, 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 it is obvious that 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 $\tau' \neq \tau$. Each 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 [24]. Moreover, the algorithm is terminated as soon C_i is empty, because then all later sets F_j and C_j with $j \geq i$ will be empty as well.

The above abstract framework, however, leaves a number of questions to be answered: (i) *How can we efficiently evaluate the search predicate $q(\tau)$?*; and, (ii) *How can we efficiently generate candidate sets C_{i+1}?*. We will next answer these questions in detail.

Search predicate. The search predicate $\mathrm{supp}(c, \tau, t)$ can be entirely evaluated on the prefix tree $\mathrm{PT}(t)$ 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 $\mathrm{PT}(t)$ using the following XPath expression:

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

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" 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:

PROPOSITION 5.1. *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 5.1 implies that $\prec_1$ as defined above really is the "successor" relation of $\preceq$. More formally, $\tau' \prec_1 \tau$ holds precisely if and only $\tau' \prec \tau$ 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 in 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:

THEOREM 5.2. *Algorithms 2 and 3 are equivalent.*

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 5.1 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

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

$t \models (c, \tau', P)$ for all sets P. Therefore, w.r.t generation of key paths P, it does not makes sense to consider all of these equivalent path separately. Rather we should choose among them one canonical path. One possibility, e.g., is to opt for the most specific path according to $\prec_1$ minimizing the length and number of wildcards. Notice that equivalence of target paths on t can be tested on the prefix tree $\mathrm{PT}(t)$ without access to the original document.

Boundedness elimination. The quality of the mining result can be improved using the results of Section 4.2. Indeed, target paths that are bounded but that have still passed the support threshold N, which may happen with low values of N, may be eliminated at this stage.

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 $v \in \mathrm{CNodes}_t(c)$ and every $u \in \tau(t, v)$, it holds that $p(t, u)$ is a singleton containing a $\mathbf{Data}$-node. Afterwards, only those key paths p are retained for which $(c, \tau, (p))$ is consistent w.r.t. X. The reason for this two-step approach is to reduce the number of costly consistency tests. Although testing for consistency w.r.t. a schema is in polynomial time (cf., Theorem 4.4), 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 w.r.t. 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 w.r.t. 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 the paths in the prefix tree from target nodes to leafs. Obviously, U_{key} can be computed directly from $\mathrm{PT}(t)$.

It remains to discuss how to compute q_τ^{key} efficiently. Unfortunately, q_τ^{key} can not always be computed solely on $\mathrm{PT}(t)$. 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 $\mathbf{Data}$-node. Then, $\mathrm{PT}(t_1) = \mathrm{PT}(t_2)$ yet ϕ is consistent on t_1 but inconsistent on t_2 for $\phi = (c_{\text{root}}, ./a/b, (./d))$ with c_{root} the root context.

We next present a sufficient condition for inconsistency which can be tested on the prefix tree. Thereto, consider $\phi = (c, \tau, (p))$ and let $t' = \mathrm{PT}(t)$. 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 $v \in \mathrm{CNodes}_{t'}(c)$ and a $u \in \tau(t', v)$ such that $\#_{t'}(u) < \sum_{w \in p(t', u)} \#_{t'}(w)$; and, (C2) There exists a $v \in \mathrm{CNodes}_{t'}(c)$, 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 that there are at least two nodes selected by p which belong to the same target node in t and which contradict consistency.

Formally, we have that:

PROPOSITION 5.3. *Given $\phi = (c, \tau, (p))$ and a document t. If condition C1 or C2 holds on $\mathrm{PT}(t)$, then ϕ is inconsistent on t.*

So, only when the tests for the above two conditions fail, 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 $\text{OneKeyPathMiner}_{t,X}(c, \tau)$ 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 $\text{OneKeyPathMiner}_{t,X}(c, \tau)$ retains from all of the returned key paths p, those for which $(c, \tau, (p))$ is consistent w.r.t. X employing the algorithm of Theorem 4.4. A duplicate elimination step similar to the one of the previous section is performed as well.

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 w.r.t. 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.

We capitalize on existing relational techniques for mining functional dependencies (e.g., [11, 21, 23]). To this end, we define a relation $R_{S,t}$ with the following 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 $\mathbf{Data}$-value selected by the corresponding key path p_i. Then, $(v, u, \bar{o}) \in R_{S,t}$ if and only if $v \in \mathrm{CNodes}_t(c)$, $u \in \tau(t, v)$ and $\mathrm{record}_S(t, u) = \bar{o}$. Now, it follows that $t \models (c, \tau, P)$ iff

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

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.

6. EXPERIMENTS

For our experiments, we use a corpus of 90 high quality XML documents and associated XSDs obtained from [1]. The input can therefore be seen as 90 pairs consisting each 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 w.r.t. to this corpus and were run on a 3GHz 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.

Prefix tree. As different parts of the algorithm can avoid access to the input document t by operating directly on $\mathrm{PT}(t)$, it is instrumental to investigate the compression rate of $\mathrm{PT}(t)$ over t. Figure 5 plots the number of nodes in documents versus the number of nodes in the corresponding prefix trees. Note that the scale is logarithmic. In essence, every document is compressed to a prefix tree with at most 200 nodes even for large documents containing ten or even hundred thousand of nodes.

Contexts. A key $\phi = (c, \tau, P)$ consists of three interdependent components: target paths need only to be considered w.r.t. a context, and key paths need only to be considered w.r.t. a context and a target path. To avoid an explosion of the size of the search space

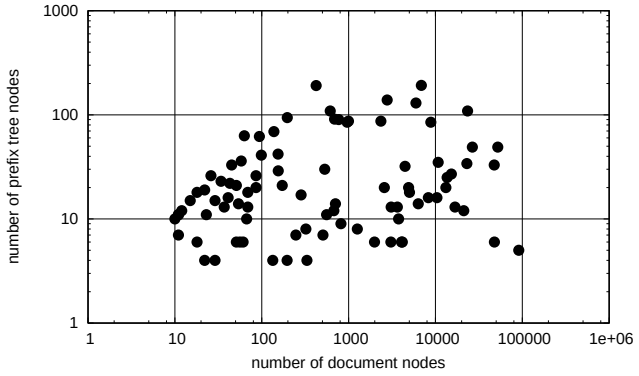

Figure 5: Number of documents versus number of nodes in prefix trees.

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 the number of contexts actually used in the XML documents, shows that for 40% of the documents all allowable contexts materialize in the corresponding XML documents, i.e., 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 4639 while the total sum of contexts found in actual documents is 2217 which indicates that over the complete data set 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.

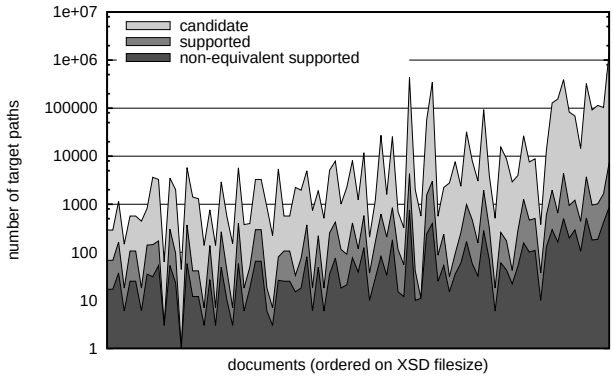

Figure 6: Behavior of the target path miner.

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 6 (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 non-equivalent derived target paths. Its purpose is to provide a visual inspection on the considered quantities on a per document basis.

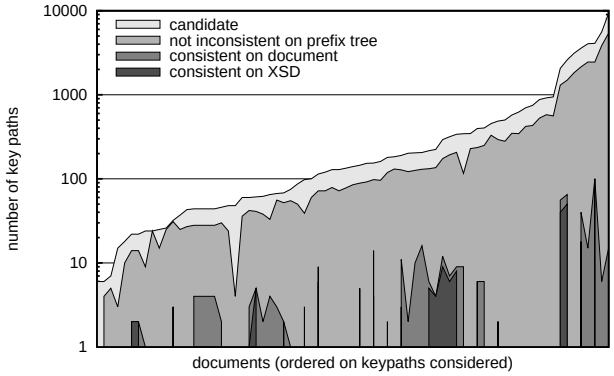

Figure 7: Behavior of one-key path miner.

By candidate target paths we mean those that occurred in a candidate set C_i during the execution of Algorithm 3. Non-equivalent 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 of those, 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 duplicate elimination. To get a feeling for the magnitude of the reduction in target paths (TPs) provided by the algorithm, we give the following table of absolute numbers which are summed up over the whole data set of document-XSD pairs:

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

One-key paths. Figure 7 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 increasingly ordered 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 which are consistent on the document, and paths which are consistent w.r.t. the XSD. We first discuss the average improvement on a per document basis. Specifically, on average 29% 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 w.r.t. the document and of these 68% turn out to be consistent w.r.t. the XSD. Absolute numbers summed up over the whole data set of document-XSD pairs, give the following picture for key paths (KPs):

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

It is interesting to observe that on the considered sample of real-world documents, consistency on the document does not always imply consistency w.r.t. the associated XSD. Specifically, the above

table shows that overall only roughly 60% of KP which are 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 [22], 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 consistency of a key w.r.t. its XSD, that is, by applying the algorithm of Theorem 4.4, as the schema consistency test. Table 2 and Table 3 then gather some statistics of discovered keys 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 2). 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 (cf., 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 supports 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 2 shows that 107 and 54 keys are derived for support threshold 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 which select elements of which the schema says they are optional. 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 which 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 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})
```

where 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 a glitch in the data. Therefore, one could say that the reduction from 107 to 43 and from 54 to 16 keys in Tables 2 and 3 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 from 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 above, provides a quality criterion in its own. A second quality criterion can be the high support of derived keys: as mentioned above 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,

	sup = 10	sup = 100
derived keys	107	54
docs with keys	27	15
average nr. of keys per doc	4	3.6
max nr. of keys per doc	23	23
average nr. of key paths	1.3	1.3
max key nr. of key paths	2	2

Table 2: Statistics of mined keys *without* the requirement to be consistent w.r.t. the associated XSD.

	sup = 10	sup = 100
derived keys	43	16
docs with keys	19	10
average nr. of keys per doc	2.2	1.6
max nr. of keys per doc	9	4
average nr. of key paths	1.3	1.2
max key nr. of key paths	2	2

Table 3: Statistics of mined keys *with* the requirement to be consistent w.r.t. the associated XSD.

from which all other keys can be derived. For instance, to this end Grahne and Zhu [19] make use of the inference algorithms for XML keys investigated and shown to be polynomially computable by Buneman et al. [15]. Unfortunately, Theorem 4.5 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/*,{/*})
```

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

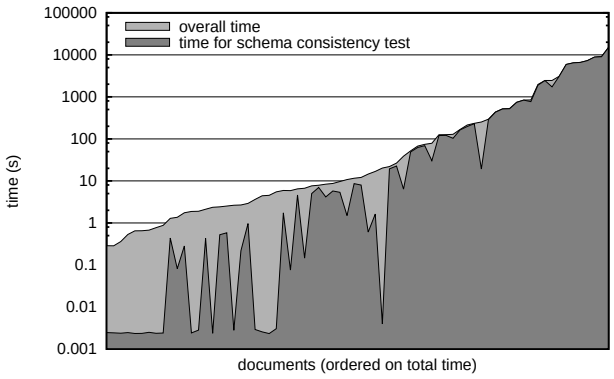

Figure 8: Proportion of the running time consumed by the schema consistency check.

Running time. We next discuss the runing 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 every such reduction

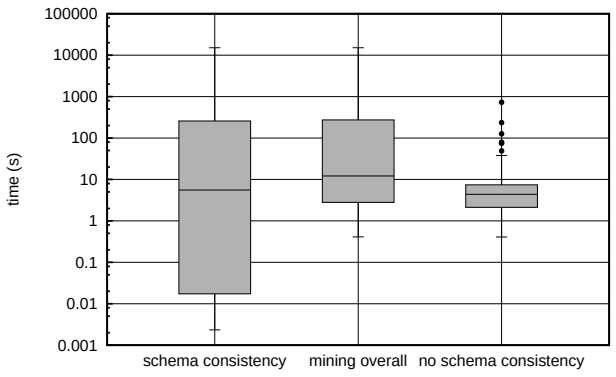

Figure 9: Boxplots indicating average run times.

induces a gain in speed. Figure 9 gives insight in the overall running time. Here, a large fraction of the time is taken up by the schema consistency test. Furthermore, Figure 8 gives an indication of the proportion of time taken by the schema consistency test w.r.t. the overall running time. For presentation purposes, the X-axis enumerates all document-XSD pairs increasingly ordered 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 corresponds to a quantity, no inference can be made about the asymptotic growth of the running time.

We want to 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 above figures also show that the most room for improvement lies within a speed up of the schema consistency test and less in other components of the algorithm.

7. DISCUSSION

In this paper, 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 w.r.t. key implication. An observed bottleneck of the proposed approach is to check consistency of a derived key w.r.t. the associated schema, even though the number of keys which 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.

8. ACKNOWLEDGEMENTS

We used the infrastructure of the VSC - Flemish Supercomputer Center, funded by the Hercules foundation and the Flemish Government. We acknowledge financial support of the Fondecyt Grant #1131049, FP7-ICT-233599 and ERC grant agreement DIADEM, no. 246858.

9. REFERENCES

[1] University of Amsterdam XML web collection. `http://data.politicalmashup.nl/sgrijzen/xmlweb/`.

[2] S. Abiteboul, Y. Amsterdamer, D. Deutch, T. Milo, and P. Senellart. Finding optimal probabilistic generators for XML collections. In *ICDT*, pages 127–139, 2012.

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

[4] M. Arenas, W. Fan, and L. Libkin. What's hard about XML schema constraints? In *DEXA*, pages 269–278, 2002.

[5] D. Barbosa and A. O. Mendelzon. Finding id attributes in XML documents. In *XSym*, pages 180–194, 2003.

[6] G. J. Bex, W. Gelade, W. Martens, and F. Neven. Simplifying XML schema: effortless handling of nondeterministic regular expressions. In *SIGMOD*, pages 731–744, 2009.

[7] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *TWEB*, 4(4), 2010.

[8] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM TODS*, 35(2), 2010.

[9] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, pages 998–1009, 2007.

[10] G. J. Bex, F. Neven, and S. Vansummeren. Schemascope: a system for inferring and cleaning XML schemas. In *SIGMOD*, pages 1259–1262, 2008.

[11] D. Bitton, J. Millman, and S. Torgersen. A feasibility and performance study of dependency inference. In *ICDE*, pages 635–641, 1989.

[12] H. Björklund, W. Martens, and T. Schwentick. Validity of tree pattern queries with respect to schema information. 2012.

[13] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Inf. Comput.*, 140(2):229–253, 1998.

[14] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Keys for XML. *Computer Networks*, 39(5):473–487, 2002.

[15] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Reasoning about keys for XML. *Inf. Syst.*, 28(8):1037–1063, 2003.

[16] S. Fajt, I. Mlynkova, and M. Necasky. On mining XML integrity constraints. In *ICDIM*, pages 23–29, 2011.

[17] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *J. ACM*, 49(3):368–406, 2002.

[18] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning document type descriptors from XML document collections. *Data Min. Knowl. Discov.*, 7(1):23–56, 2003.

[19] G. Grahne and J. Zhu. Discovering approximate keys in XML data. *CIKM*, page 453-460, 2002.

[20] S. Hartmann and S. Link. Efficient reasoning about a robust XML key fragment. *ACM TODS*, 34(2), 2009.

[21] H. Mannila and K.-J. Raiha. Practical algorithms for finding prime attributes and testing normal forms. In *PODS*, 1989.

[22] H. Mannila and K.-J. Räihä. *The design of relational databases*. Addison- Wesley, 1991.

[23] H. Mannila and K.-J. Räihä. Algorithms for inferring functional dependencies from relations. *Data Knowl. Eng.*, 12(1):83–99, 1994.

[24] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Min. Knowl. Discov.*, 1(3):241–258, 1997.

[25] W. Martens, F. Neven, and T. Schwentick. Simple off the shelf abstractions for XML schema. *SIGMOD Record*, 36(3):15–22, 2007.

[26] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML schema. *ACM TODS*, 31(3):770–813, 2006.

[27] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Techn.*, 5(4):660–704, 2005.

[28] M. Necaský and I. Mlýnková. Discovering XML keys and foreign keys in queries. In *SAC*, pages 632–638. ACM, 2009.

[29] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[30] H. Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19(3):424–437, 1990.

[31] W3C. XML schema part 1: Structures, 2nd edition.

[32] C. Yu and H. V. Jagadish. XML schema refinement through redundancy detection and normalization. *VLDB J.*, 17(2):203–223, 2008.