Performance Analysis of Algorithms to Reason about XML Keys*

Flavio Ferrarotti[1], Sven Hartmann[2], Sebastian Link[3], Mauricio Marin[4], and Emir Muñoz[4,5,**]

[1] Victoria University of Wellington
[2] Clausthal University of Technology
[3] The University of Auckland
[4] Yahoo! Research
[5] University of Santiago de Chile
Flavio.Ferrarotti@vuw.ac.nz

Abstract. Keys are fundamental for database management, independently of the particular data model used. In particular, several notions of XML keys have been proposed over the last decade, and their expressiveness and computational properties have been analyzed in theory. In practice, however, expressive notions of XML keys with good reasoning capabilities have been widely ignored. In this paper we present an efficient implementation of an algorithm that decides the implication problem for a tractable and expressive class of XML keys. We also evaluate the performance of the proposed algorithm, demonstrating that reasoning about expressive notions of XML keys can be done efficiently in practice and scales well. Our work indicates that XML keys as those studied here have great potential for diverse areas such as schema design, query optimization, storage and updates, data exchange and integration. To exemplify this potential, we use the algorithm to calculate non-redundant covers for sets of XML keys, and show that these covers can significantly reduce the number of XML keys against which XML documents must be validated. This can result in enormous time savings.

1 Introduction

The increasing popularity of XML for persistent data storage and data processing has triggered the demand for efficient algorithms to manage XML data. Both industry and academia have long since recognized the importance of keys in XML data management. Over the last decade, several notions of XML keys have been proposed and discussed in the database community. The most influential proposal is due to Buneman et al. [3,4] who defined keys on the basis of an XML

* This research is supported by the Marsden Fund Council from Government funding, administered by the Royal Society of New Zealand.

** The contribution of this author was based on his Master's thesis, which was supported by grants from the University of Santiago de Chile and Yahoo! Labs.

tree model similar to the one suggested by DOM [1] and XPath [6]. While Buneman et al. studied keys as a concept orthogonal to schema specification (such as DTD or XSD), their proposal has been adopted by the W3C for the XML Schema standard [15] subject to some minor, though essential modifications (see [2] for a discussion). Today, all major XML-enabled DBMS, XML parsers and editors (such as XMLSpy) support keys.

Example 1. Figure 1 shows an XML tree, in which nodes are annotated by their type: E for element nodes, A for attribute nodes, and S for text nodes. For the data represented in Figure 1 we have the following keys: (a) A *project* node is identified by *pname*, no matter where the *project* node appears in the document. (b) A *team* node can be identified by *tname* relatively to a *project* node. (c) Within any given subtree rooted at *team*, an *employee* node is identified by *name*. The first key is an example of an *absolute key* since it must hold globally throughout the entire tree. The last two are examples of *relative keys* since they hold locally within some subtrees. Note that a given team of employees can work on several projects and thus a *team* node cannot be identified in the entire tree by its *tname*. However, it holds locally within each subtree rooted at a *project* node. Similarly, a given employee can work on different teams and thus cannot be identified in the entire tree by its *name*. □

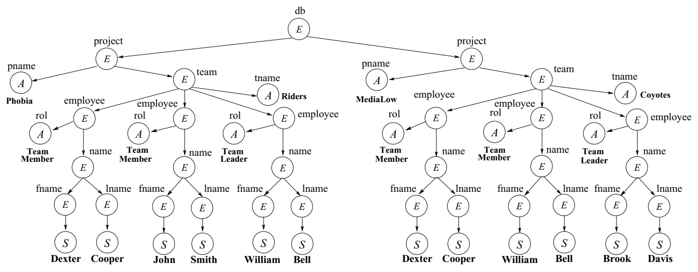


Fig. 1. An XML tree representing an XML document

For relational data, keys have been widely used to improve the performance of many perennial tasks in database management, ranging from consistency checking to query answering. The hope is that keys will turn out to be equally beneficial for XML. One of the most fundamental questions on keys is that of logical implication, that is, deciding if a new key holds given a set of known keys. Among other things, this is important for minimizing the cost of validating that an XML document satisfies a set of keys gathered as business rules during requirements engineering.

Example 2. Suppose, the database designer has already specified the keys (a), (b) and (c). Now she considers a further key (d) which expresses that a *project* node can be identified by its child nodes *pname* and *team*. By key (a) one already knows that a *project* node can also be identified by just its *pname*. It is easy to see that (a) actually implies (d), in the sense that every XML tree that satisfies (a) also satisfies (d). Thus, instead of checking whether an XML tree T satisfies (a) and (d), we could just check whether T satisfies (a). We would like to emphasize that *project* nodes have complex content. Thus, checking whether two *project* nodes in T violate (d) is quite costly in terms of time, since it involves testing whether the subtrees rooted at their *team* nodes are isomorphic to one another with the identity on string values. In contrast, checking whether two *project* nodes violate (a) only involves checking equality on text of their respective *pname* attribute nodes. □

Example 3. No less important, the implication of XML keys is of interest for semantically rich data exchange. Suppose, the company wants to share part of their project data with a business partner. For that they generate a view over the XML tree T but skip the *lname* nodes for the sake of privacy. Thus, key (c) is no longer meaningful. To provide the business partner with relevant semantic information it should be checked whether the specified keys allow one to conclude a further key stating that an *employee* node is identified by their remaining descendant nodes *fname* and *role* within any given *team* subtree. □

The definition of keys adopted by the W3C for XML Schema [15] is currently the industry standard for specifying keys. However, Arenas et al. [2] have shown the computational intractability of the associated consistency problem, i.e., the question whether there exists an XML document that conforms to a given XSD and satisfies the specified keys. A further issue pointed out by Buneman et al. [3] is the fact that XML Schema restricts value equality to string-valued data items. But there are cases in which keys are not so restricted (see Section 7.1 of [3] for discussion). In particular, keys (c) and (d) in our examples utilize a less restricted notion of equality, since they require to test equality between *name* nodes and *team* nodes, respectively, none of which are string-valued. On the other hand, the expressiveness and computational properties of XML keys with good reasoning capabilities have been deeply studied from a theoretical perspective [3,4,9]. In practice, however, expressive yet tractable notions of XML keys have been ignored so far.

Aiming to fill this gap between theory and practice, we initiate in this work an empirical study of an expressive XML key fragment, namely the fragment of XML keys with nonempty sets of simple key paths. As shown in [4,9], automated reasoning about this XML key fragment can be done efficiently, in theoretical terms. Our work confirms this fact in practice. Incidentally, note that all the examples of XML keys described above belong to this fragment.

In this paper, we describe an efficient implementation of an algorithm that decides the implication problem for an expressive fragment of XML keys and thoroughly evaluate its performance. Our performance tests give first empirical evidence that reasoning about expressive notions of XML keys is practically efficient and scales well. Our work indicates that XML keys have great potential for database management tasks similar to their counterparts for relational data.

Exploiting our algorithm we compute non-redundant covers for sets of XML keys. A set Σ of keys is non-redundant if there is no key σ in Σ such that σ is implied by $\Sigma - \{\sigma\}$. Thus, considering such covers has the potential to reduce significantly the number of keys against which an XML document must be validated. This can result in enormous time savings. Our experiments show that the time to compute a cover for a given set of keys is just a small fraction of the average time needed to validate an XML document against a single key. Surprisingly, even though several algorithms that validate XML documents against sets of certain XML keys have been proposed and tested with promising results (see e.g. [5,12]), none of them makes use of the reasoning capabilities of XML keys as proposed in our work.

The paper is organized as follows. We recall basic notions in Section 2, including the central notion of XML keys which is used through this work. In Section 3, we present the algorithm for deciding XML key implication, and describe an implementation thereof in Section 4. In Section 5, we discuss how this implementation can be reused to speed up the validation of XML documents against sets of XML keys. Section 6 summarizes experimental results obtained from applying our implementations to publicly available XML data, including DBLP, the SIGMOD Record, and the Mondial database. We conclude the paper in Section 7 with final remarks.

2 Keys for XML

We use the common representation of XML data as ordered, node-labelled trees. Thus, an *XML tree* is a 6-tuple $T = (V, lab, ele, att, val, r)$ where V is a set of nodes, and lab is a mapping $V \to \mathcal{L} = \mathbf{E} \cup \mathbf{A} \cup \{S\}$ assigning a label to every node in V. A node $v \in V$ is an *element node* if $lab(v) \in \mathbf{E}$, an *attribute node* if $lab(v) \in \mathbf{A}$, and a *text node* if $lab(v) = S$. Here $\mathbf{E} \cup \mathbf{A} \cup \{S\}$ form a partition of $\mathcal{L}$. Moreover, ele and att are partial mappings defining the edge relation of T: for any node $v \in V$, if v is an element node, then $ele(v)$ is a list of element and text nodes in V and $att(v)$ is a set of attribute nodes in V. The partial mapping val assigns a string to each attribute and text node. Finally, r is the unique and distinguished root node.

The XML keys studied in this work are defined using the path language PL consisting of expressions given by the following grammar: $Q \to \ell \mid \varepsilon \mid Q.Q \mid \_^*$. Here $\ell \in \mathcal{L}$ is any label, ε denotes the empty path expression, "." denotes the concatenation of two path expressions, and "$\_^*$" denotes the *variable length "don't care"* wildcard. Let Q be a word from PL. A path $v_1, \ldots, v_n$ in an XML tree T is called a Q-path if $lab(v_1). \cdots .lab(v_n)$ can be obtained from Q by replacing variable length wildcards in Q by words from PL. For a node $v \in V$, $v[\![Q]\!]$ denotes the set of nodes in T that are reachable from v following any Q-path. We use $[\![Q]\!]$ as an abbreviation for $r[\![Q]\!]$ where r is the root node of T. We denote as PL_s the subset of PL expressions containing all words over the alphabet $\mathcal{L}$, i.e., we do not allow wildcards in PL_s expressions. $Q \in PL$ is *valid* if it does not have labels $\ell \in \mathbf{A}$ or $\ell = S$ in a position other than the last one.

We define formally the concept of XML key following [4]. For that, we need the concept value equality. Two nodes $u, v \in V$ are *value equal*, denoted by $u =_v v$, iff the subtrees rooted at u and v are isomorphic by an isomorphism that is the identity on string values. As an example, the third and fifth *employee*-nodes are not value equal while their respective child nodes labeled as *name* are.

Definition 1. *An XML key φ in the class $\mathcal{K}$ is an expression of the form $(Q_\varphi, (Q'_\varphi, \{P^\varphi_1, \ldots, P^\varphi_{k_\varphi}\}))$ where $k_\varphi \geq 1$, Q_φ and Q'_φ are PL expressions, and for all $i = 1, \ldots, k_\varphi$, P^φ_i is a PL_s expressions such that $Q_\varphi.Q'_\varphi.P^\varphi_i$ is a valid PL expression. An XML tree T satisfies the key $(Q, (Q', \{P_1, \ldots, P_k\}))$ if and only if for every node $q \in [\![Q]\!]$ and all nodes $q'_1, q'_2 \in q[\![Q']\!]$ such that there are nodes $x_i \in q'_1[\![Q_i]\!], y_i \in q'_2[\![P_i]\!]$ with $x_i =_v y_i$ for all $i = 1, \ldots, k$, then $q'_1 = q'_2$. Therefore, Q_φ is called the* context path*, Q'_φ is called the* target path*, and $P^\varphi_1, \ldots, P^\varphi_{k_\varphi}$ are called the* key paths *of φ.*

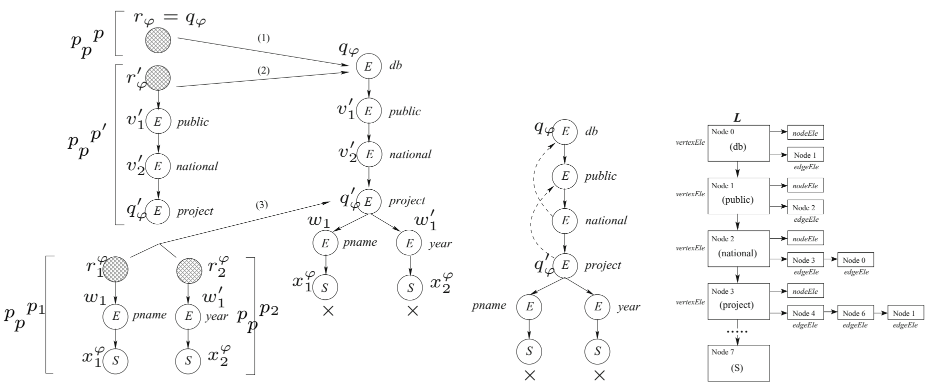
In particular, the four keys described informally in the introduction, belong to this class and can be expressed formally as follows: (a) $(\varepsilon, (project, \{pname\}))$; (b) $(project, (team\{tname\}))$; (c) $(\_^*.team, (employee, \{name\}))$; (d) $(\varepsilon, (project, \{pname, team\}))$.

3 Deciding XML Key Implication

Let $\Sigma \cup \{\varphi\}$ be a finite set of XML keys in a class $\mathcal{C}$. We say that Σ *implies* φ, denoted by $\Sigma \models \varphi$, if and only if every finite XML tree T that satisfies all $\sigma \in \Sigma$ also satisfies φ. The *implication problem* for $\mathcal{C}$ is to decide, given any finite set $\Sigma \cup \{\varphi\}$ of keys in $\mathcal{C}$, whether $\Sigma \models \varphi$.

A finite axiomatization for the implication of keys in the class of XML keys with nonempty sets of simple key paths $\mathcal{K}$, was established in [9]. The completeness proof of this axiomatization is based on a characterization of key implication in terms of the reachability problem for fixed nodes in a suitable digraph. This characterization, together with the efficient evaluation of Core XPath [8], resulted in a compact algorithm to decide XML key implication in time quadratic in the size of the input key. This algorithm, which is described next, forms the basis for our implementation. We need the following technical concepts.

Mini-trees and Witness Graphs. Let $\Sigma \cup \{\varphi\}$ be a finite set of keys in $\mathcal{K}$. Let $\mathcal{L}_{\Sigma,\varphi}$ denote the set of all labels $\ell \in \mathcal{L}$ that occur in path expressions of keys in $\Sigma \cup \{\varphi\}$, and fix a label $\ell_0 \in \mathbf{E} - \mathcal{L}_{\Sigma,\varphi}$. Let O_φ and O'_φ be the PL_s expressions obtained from the PL expressions Q_φ and Q'_φ, respectively, by replacing each wildcard "$\_^*$" by ℓ_0. Let p be an O_φ-path from a node r_φ to a node q_φ, let p' be an O'_φ-path from a node r'_φ to a node q'_φ and, for each $i = 1, \ldots, k_\varphi$, let p_i be a P^φ_i-path from a node r^φ_i to a node x^φ_i, such that the paths $p, p', p_1, \ldots, p_{k_\varphi}$ are mutually node-disjoint. From the paths $p, p', p_1, \ldots, p_{k_\varphi}$ we obtain the *mini-tree* $T_{\Sigma,\varphi}$ by identifying the node r'_φ with q_φ, and by identifying each of the nodes r^φ_i with q'_φ. The *marking* of the mini-tree $T_{\Sigma,\varphi}$ is a subset $\mathcal{M}$ of the node set of $T_{\Sigma,\varphi}$: if for all $i = 1, \ldots, k_\varphi$ we have $P^\varphi_i \neq \varepsilon$, then $\mathcal{M}$ consists of the leaves of $T_{\Sigma,\varphi}$, and otherwise $\mathcal{M}$ consists of all descendant nodes of q'_φ in $T_{\Sigma,\varphi}$.



(a) Mini tree $T_{\Sigma,\varphi}$ (b) Witness graph $G_{\Sigma,\varphi}$ (c) Adj. list for $G_{\Sigma,\varphi}$

Fig. 2. Mini-tree, Witness-graph and Adjacency list

Example 4. Let $\Sigma = \{\sigma_1, \sigma_2\}$ where σ_1 and σ_2 are the XML keys $(\varepsilon, (\mathit{public}.\_^*, \{\mathit{project.pname.S}, \mathit{project.year.S}\}))$ and $(\mathit{public}, (\_^*.\mathit{project}, \{\mathit{pname}.\mathrm{S}, \mathit{year.S}\}))$, respectively. Let $\varphi = (\varepsilon, (\mathit{public}.\_^*.\mathit{project}, \{\mathit{pname.S}, \mathit{year.S}\}))$. The construction of the mini-tree $T_{\Sigma,\varphi}$ is schematized in Figure 2 (a).

The mini-trees are used in the algorithm as a base to calculate the impact of a key in Σ on a possible counter-example tree for the implication of φ by Σ. To distinguish keys that have an impact from those that do not, the following notion of *applicability* is needed. Let $T_{\Sigma,\varphi}$ be the mini-tree of the key φ with respect to Σ, and let $\mathcal{M}$ be its marking. A key σ is said to be *applicable* to φ if and only if there are nodes $w_\sigma \in [\![Q_\sigma]\!]$ and $w'_\sigma \in w_\sigma[\![Q'_\sigma]\!]$ in $T_{\Sigma,\varphi}$ such that $w'_\sigma[\![P^\sigma_i]\!] \cap \mathcal{M} \neq \emptyset$ for all $i = 1, \ldots, k_\sigma$. We say that w_σ and w'_σ *witness* the applicability of σ to φ.

We define the *witness graph* $G_{\Sigma,\varphi}$ as the node-labeled digraph obtained from $T_{\Sigma,\varphi}$ by inserting additional edges: for each key $\sigma \in \Sigma$ that is applicable to φ and for each pair of nodes $w_\sigma \in [\![Q_\sigma]\!]$ and $w'_\sigma \in w_\sigma[\![Q'_\sigma]\!]$ that witness the applicability of σ to φ, $G_{\Sigma,\varphi}$ contains the directed edge (w'_σ, w_σ) from w'_σ to w_σ.

Example 5. Let Σ and φ be as in Example 4. Both keys in Σ are applicable to φ. The witness graph $G_{\Sigma,\varphi}$ is shown in Figure 2 (b). It contains a witness edge from *national* to *db* that arises from σ_1 and a witness edge from *project* to *public* that arises from σ_2.

The algorithm. Algorithm 1 decides XML key implication. Its correctness is an immediate consequence of Theorem 1.

Theorem 1. ([9]) *Let $\Sigma \cup \{\varphi\}$ be a finite set of keys in the class $\mathcal{K}$. We have $\Sigma \models \varphi$ if and only if q_φ is reachable from q'_φ in $G_{\Sigma,\varphi}$.*

Algorithm 1. (XML key implication in $\mathcal{K}$)

```
Input: finite set of XML keys $\Sigma \cup \{\varphi\}$ in $\mathcal{K}$
Output: yes, if $\Sigma \models \varphi$; no, otherwise
 1: Construct $G_{\Sigma,\varphi}$ for $\Sigma$ and $\varphi$;
 2: if $q_\varphi$ is reachable from $q'_\varphi$ in $G$ then return yes;
    else return no; end if
```

4 An Efficient Implementation

In this section we discuss our implementation of Algorithm 1 and analyze its theoretical complexity. The implementation was developed in C++ using *gcc* version 4.4.3 from the GNU compiler collection.

Data Structures. We need data structures suitable to represent mini-trees and witness-graphs. The obvious candidates are adjacency matrices and adjacency lists. Since the algorithm does not require frequent determination of edge existence, we choose the latter in order to minimize the memory requirements. In our implementation, a mini-tree $T_{\Sigma,\varphi}$ is represented by using a list L of length $n = |V|$ where V is the vertex set of $T_{\Sigma,\varphi}$. Each element $e_i \in L$ is represented by an object of type *vertexEle* that has a pointer to the adjacency list of the i-th vertex v_i in some fixed enumeration of the vertices in V, a pointer to the data component of the vertex v_i, and a pointer to the next element e_{i+1} in the list. In turn, the data component of a vertex v_i is represented by an object of type *nodeEle*, and an element in the adjacency list of a vertex v_i is represented by an object of type *edgeEle*. An object of type *nodeEle* has an *id* component that uniquely identifies v_i, a *label* component with the label of v_i, a flag *visited*, and a *type* component with the type E (element), A (attribute) or S (PCDATA) of v_i. An object of type *edgeEle* has a pointer to an object of type *vertexEle* and a pointer to the next object of type *edgeEle* in the adjacency list. Witness graphs are represented likewise. Figure 2(b) shows a witness graph and Figure 2(c) a corresponding representation using adjacency lists.

The Implementation. We implemented Step 1 of Algorithm 1, using the following strategy:

i. Construct $T_{\Sigma,\varphi}$;
ii. Determine the marking of $T_{\Sigma,\varphi}$;
iii. For each $\sigma \in \Sigma$, add the edge (w'_σ, w_σ) to $T_{\Sigma,\varphi}$ whenever w_σ and w'_σ witness the applicability of σ to φ.

Substep (i) involves constructing the mini-tree $T_{\Sigma,\varphi}$ using the data structures defined at the beginning of this section. Note that we can find a label ℓ_0 that is not among the labels used in the XML keys in $\Sigma \cup \{\varphi\}$ in time $\sum_{\sigma_i \in \Sigma} |\sigma_i| + |\varphi|$, where $|\sigma_i|$ and $|\varphi|$ denote the sum of the lengths of all path expressions in σ_i and φ, respectively. Once we have got a suitable label, ℓ_0, $T_{\Sigma,\varphi}$ can be built in time $\mathcal{O}(|\varphi|)$, since the mini-tree $T_{\Sigma,\varphi}$ has only $|\varphi| + 1$ nodes.

Regarding Substep (ii), if $P_i^{\varphi} \neq \varepsilon$ we can determine the marking of the mini-tree $T_{\Sigma,\varphi}$ by simply traversing the list L marking the nodes whose adjacency list is empty. Note that those nodes correspond to leaves in $T_{\Sigma,\varphi}$. Otherwise, we mark all nodes in the adjacency list of the element e_i in L that represents q'_{φ}, and recursively mark all descendants of those nodes. This step takes $\mathcal{O}(|\varphi|)$ time.

In principle, Substep (iii) requires, for each $\sigma \in \Sigma$, to evaluate $w'_{\sigma}[\![P_i^{\sigma}]\!]$ for $i = 1, \ldots, k_{\sigma}$, for all $w'_{\sigma} \in w_{\sigma}[\![Q'_{\sigma}]\!]$ and all $w_{\sigma} \in [\![Q_{\sigma}]\!]$. However, we do not need to determine *all* witness edges (w', w) to decide whether q_{φ} is reachable from q'_{φ} in the witness graph $G_{\Sigma,\varphi}$. Let W'_{σ} be the set of all nodes w' in $T_{\Sigma,\varphi}$ for which there exists some node w in $T_{\Sigma,\varphi}$ such that w and w' witness the applicability of σ to φ. Further, for each $w' \in W'_{\sigma}$, let $W_{\sigma}(w')$ be the set of all nodes w in $T_{\Sigma,\varphi}$ such that w and w' witness the applicability of σ to φ. The witness edges are just the pairs (w', w) with $w' \in W'_{\sigma}$ and $w \in W_{\sigma}(w')$. As shown in [9], it is not necessary to determine the entire set $W_{\sigma}(w')$ for each $w' \in W_{\sigma}$. We can actually restrict ourselves to the top-most ancestor of q'_{φ} in $T_{\Sigma,\varphi}$ that belongs to $W_{\sigma}(w')$, which we denote by $w_{\sigma}^{top}(w')$ (if it exists).

So, we need first to determine W'_{σ}, and then, for each $w' \in W'_{\sigma}$, we need to determine $w_{\sigma}^{top}(w')$ (if it exists). By definition, W'_{σ} consists of all nodes $w' \in [\![Q_{\sigma}.Q'_{\sigma}]\!]$ in $T_{\Sigma,\varphi}$ such that, for each $i = 1, \ldots, k_{\sigma}$, there is a marked node in $w'[\![P_i^{\sigma}]\!]$. Since a query of the form $v[\![Q]\!]$ is a Core XPath query and can be evaluated on a node-labelled tree T in $\mathcal{O}(|T| \times |Q|)$ time, it follows that $[\![Q_{\sigma}.Q'_{\sigma}]\!]$ can be evaluated in $T_{\Sigma,\varphi}$ in $\mathcal{O}(|\varphi| \times |Q_{\sigma}.Q'_{\sigma}|)$ time. Next, fix some $i \in \{1, \ldots, k_{\sigma}\}$. Let v be a marked node, and let u denote the ancestor of v that resides $|P_i^{\sigma}|$ levels atop of v in $T_{\Sigma,\varphi}$ (if it exists). We can then check whether $v \in u[\![P_i^{\sigma}]\!]$, that is, whether the unique path from u to v is a P_i^{σ}-path. This can be done in $\mathcal{O}(\min\{|P_i^{\sigma}|, |\varphi|\})$ time, since P_i^{σ} is a PL_s expression. By inspecting all nodes $v \in \mathcal{M}$, we obtain the set U_i^{σ} of all nodes u in $T_{\Sigma,\varphi}$ for which $u[\![P_i^{\sigma}]\!] \cap \mathcal{M} \neq \emptyset$. Overall, this takes $\mathcal{O}(|\mathcal{M}| \times |P_i^{\sigma}|\})$ time. Since W'_{σ} is the intersection of $[\![Q_{\sigma}.Q'_{\sigma}]\!]$ with the sets U_i^{σ}, $i = 1, \ldots, k_{\sigma}$, we get that W'_{σ} can be determined in $\mathcal{O}(|\varphi| \times |\sigma|)$ time. Regarding $w_{\sigma}^{top}(w')$, note that if Q'_{σ} is a PL_s expression, then $w_{\sigma}^{top}(w')$ is the node $|Q'_{\sigma}|$ levels atop of w' in $T_{\Sigma,\varphi}$. Otherwise Q'_{σ} contains a $\_^*$, and thus has the form $A.\_^*.B$ where A is a PL_s expression and B is a PL expression. In this case, as shown in [9], $w_{\sigma}^{top}(w')$ is the top-most ancestor w of q'_{φ} in $T_{\Sigma,\varphi}$ that belongs to $[\![Q_{\sigma}]\!]$ and for which $w[\![A]\!]$ is non-empty. In particular, $w_{\sigma}^{top}(w')$ is independent from the choice of w' in W'_{σ}. Thus, we propose Algorithm 2 to determine $w_{\sigma}^{top}(w')$ for a given node w'.

Since $[\![Q_{\sigma}.A]\!]$ can be evaluated in $\mathcal{O}(|\varphi| \times |Q_{\sigma}.A|)$ time, we can conclude from the previous algorithm that $w_{\sigma}^{top}(w')$ for a given w' can be determined in $\mathcal{O}(|\varphi| \times |Q_{\sigma}.A|)$ time. Thus, it takes us $\mathcal{O}(|\varphi| \times |\sigma|)$ time to determine all the witness edges arising from σ that are needed for deciding the reachability of q_{φ} from q'_{φ} in $G_{\Sigma,\varphi}$. Finally, Step 2 of Algorithm 1 can be implemented by applying a depth-first search algorithm to $G_{\Sigma,\varphi}$ with root q'_{φ}. This algorithm works in time linear in the number of edges of $G_{\Sigma,\varphi}$ [11]. Over all, our implementation can decide the implication problem $\Sigma \models \varphi$ in $\mathcal{O}(|\varphi| \times (\sum_{\sigma_i \in \Sigma} |\sigma_i| + |\varphi|))$ time.

Algorithm 2. (Determine $w_\sigma^{top}(w')$)

```
Input: a mini-tree $T_{\Sigma,\varphi}$, a set $W'_\sigma$, and a node $w' \in W'_\sigma$.
Output: $w_\sigma^{top}(w')$
 1: if $Q'_\sigma$ is a $PL_s$ expression then
 2:    return The node $|Q'_\sigma|$ levels atop of $w'$ in $T_{\Sigma,\varphi}$
 3: else
 4:    Determine the set $[\![Q_\sigma.A]\!]$ of nodes in $T_{\Sigma,\varphi}$
 5:    if $[\![Q_\sigma.A]\!] \neq \emptyset$ then
 6:       Choose a topmost node $v$
 7:       Select the node $w$ that is $|A|$ levels atop of $v$ in $T_{\Sigma,\varphi}$
 8:       if $w$ is an ancestor of $q'_\varphi$ then
 9:          return $w$
10:       else
11:          return $\perp$ //$w_\sigma^{top}(w')$ does not exist.
12:       end if
13:    end if
14: end if
```

5 Applying XML Key Reasoning to Document Validation

Fast algorithms for the validation of XML documents against keys are crucial to ensure the consistency and semantic correctness of data stored in databases or exchanged between applications. In this section we explain how our implementation of the implication algorithm for XML keys can be used to compute non-redundant cover sets of XML keys, which in turn can be used to significantly speed up the process of XML document validation against sets of XML keys. This is, up to our knowledge, the first time that the reasoning capabilities of XML keys are used in this context.

Cover Sets for XML Keys. We define the concept of *cover* set of XML keys following the notion given in [13] for functional dependencies in the relational model.

Definition 2. *Let Σ^* denote the set of all XML keys implied by a given set Σ. Two sets Σ_1 and Σ_2 of XML keys are* equivalent*, denoted by $\Sigma_1 \equiv \Sigma_2$, if $\Sigma_1^* = \Sigma_2^*$. If Σ_1 and Σ_2 are equivalent we call them a* cover *of one another. This means that Σ_1 and Σ_2 imply exactly the same XML keys.*

For all cover Σ_2 of Σ_1, if an XML tree T_D satisfies Σ_2 ($T_D \models \Sigma_2$), then $T_D \models \Sigma_1$ too. If $\Sigma_1 \equiv \Sigma_2$, then for each XML key ψ in Σ_1^*, $\Sigma_2 \models \psi$, because $\Sigma_2^* = \Sigma_1^*$. In particular, $\Sigma_2 \models \psi$ for each key ψ in Σ_1.

Definition 3. *A set Σ_2 of XML keys is* non-redundant *if it is not equivalent to any of its proper subsets. Σ_2 is a* non-redundant cover *for a set Σ_1 of XML keys if Σ_2 is non-redundant and a cover for Σ_1.*

An important property is that a non-redundant cover set has in most cases fewer keys that the original one (in the extreme case both sets are equal). This can result in enormous time saving when validating an XML document against a set of XML keys, as we will show in the experimental results.

A characterization of non-redundancy is that Σ is non-redundant if there is no key ψ in Σ such that $\Sigma - \{\psi\} \models \psi$. A key $\psi \in \Sigma$ is called redundant if $\Sigma - \{\psi\} \models \psi$. Thus, we propose Algorithm 3 to compute, given a set Σ of XML keys, a non-redundant cover Θ of Σ.

Algorithm 3. (Non-redundant Cover for XML keys)

```
Input: finite set $\Sigma$ of XML keys
Output: a non-redundant cover for $\Sigma$
1: $\Theta = \Sigma$;
2: for each key $\psi \in \Sigma$ do
3:     if $\Theta - \{\psi\} \models \psi$ then
4:         $\Theta = \Theta - \{\psi\}$;
5:     end if
6: end for
7: return $\Theta$;
```

It is important to note that a set Σ can have more than one non-redundant cover set and there can exist non-redundant cover sets that are not included in Σ.

The complexity of Algorithm 3 is determined by the complexity of the implication algorithm which is executed once for every key in Σ. Thus a non-redundant cover set for a set Σ of XML keys in $\mathcal{K}$ can be computed in $\mathcal{O}(|\Sigma| \times (max\{|\psi| : \psi \in \Sigma\})^2)$ time.

6 Experimental Results

In the following we present a performance analysis of the algorithms proposed in this work. Up to our knowledge, this is the first time that the theory on automated reasoning about XML keys is tested in practice. The running time results were obtained in an Intel Core 2 Duo 2.0 GHz machine, 3GB RAM, and Linux kernel 2.6.32.

The Data Set. We used a collection of large XML documents from [14]. The collection consists of the following XML documents. A characterization of the documents is shown in Table 1.

- *321gone.xml and yahoo.xml.* Auction data converted to XML.
- *dblp.xml.* Bibliographic information on computer science.
- *nasa.xml* Astronomical Data converted from legacy flat-file format into XML.
- *SigmodRecord.xml.* Index of articles from SIGMOD Record.
- *mondial-3.0.xml.* World geographic database from several sources.

We defined, for each document in the collection, a corresponding set of 5 to 10 appropriate (in the context of the document) XML keys.

Table 1. XML Documents

Doc ID	Document	No. of Elements	No. of Attributes	Size	Max. Depth	Average Depth
Doc1	321gone.xml	311	0	23 KB	5	3.76527
Doc2	yahoo.xml	342	0	24 KB	5	3.76608
Doc3	dblp.xml	29,494	3,247	1.6 MB	6	2.90228
Doc4	nasa.xml	476,646	56,317	23 MB	8	5.58314
Doc5	SigmodRecord.xml	11,526	3,737	476 KB	6	5.14107
Doc6	mondial-3.0.xml	22,423	47,423	1 MB	5	3.59274

Then, in order to test the scalability of the implication algorithm, we generated large sets of XML keys in the following two systematic ways. Firstly, using the manually defined sets of XML keys as seeds, we computed new implied keys by successively applying the inference rules from the axiomatization of XML keys presented in [9]. For instance, by applying the interaction rule to $(listing, (auction\_info, \{high\_bidder.\,bidder\_name.S, high\_bidder.bidder\_rating.S\}))$ and $(listing.auction\_info, (high\_bidder, \{bidder\_name.S, bidder\_rating.S\}))$, we derived the implied key $(listing, (auction\_info.high\_bidder, \{bidder\_name.S, bidder\_rating.S\}))$. Each key generated by this method was added to the original set. We applied the interaction, context-target, subnodes, context-path containment, target-path containment, subnodes-epsilon and prefix-epsilon rules whenever possible, since those are the rules which can produce implied keys with corresponding non trivial witness graphs (see the proof of Lemma 3.6 in [9]). Secondly, we defined some non-implied (by the keys defined previously) XML keys. We did that by taking non-implied XML keys φ, building their corresponding mini-trees $T_{\Sigma,\varphi}$, adding several witness edges to it while keeping q_φ not reachable from q'_φ, and finally defining new non-implied XML keys corresponding to those witness edges. As an example, let us take the mini-tree in Figure 3 which corresponds to the key $\varphi = (conference, (issue.\_^*.articles.article.author, \{first.S, last.S\}))$. From the witness edges (a), (b) and (c), we obtained the keys, $(conference.issue, (\_^*, \{articles.article.author.first.S\}))$, $(conference.issue.\_^*, (articles.article, \{author.first.S\}))$ and $(conference.issue.\_^*.articles, (article.author\{first.S\}))$, respectively.

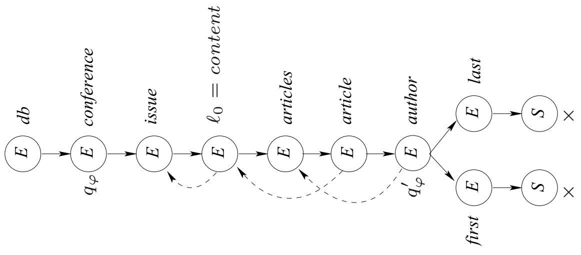


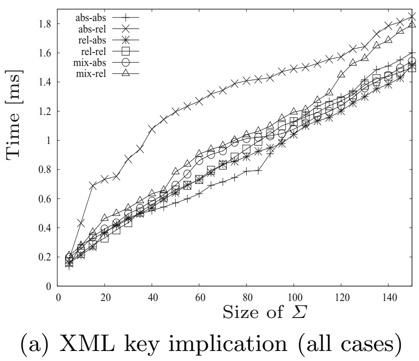
Fig. 3. Mini-tree corresponding to a non-implied key

This process gave us a robust collection of XML keys to thoroughly test the performance of the implication algorithm.

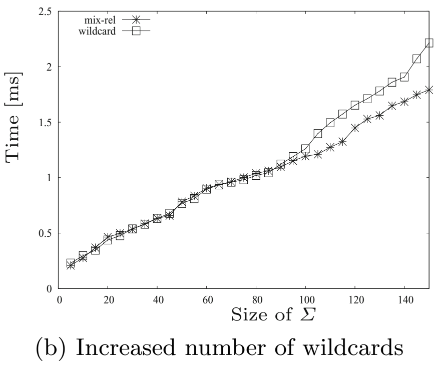
Deciding Implication of XML Keys: Tests Results. The results regarding running times for deciding the implication of XML keys are shown in Figures 4(a) and 4(b). In both figures, the x-axis corresponds to the number of keys in Σ, and the y-axis corresponds to the *average* running time required to decide whether Σ implies a given key φ. More precisely, let $time(\Sigma, \varphi)$ be the running time required to decide $\Sigma \models \varphi$ and let Φ be a set of XML keys such that $\Sigma \cap \Phi = \emptyset$, the running time shown in Figures 4(a) and 4(b), corresponds to $\left(\sum_{\varphi_i \in \Phi} time(\Sigma, \varphi_i)\right)/|\Phi|$. In our experiments the sets Φ were composed of 20 fixed XML keys each. We tested the scalability of the algorithm by adding, in each iteration, 5 new XML key to the corresponding Σ sets. The actual XML keys included in all these sets were created using the strategy explained above.

We consider Σ sets composed by (i) only absolute keys ("abs"), (ii) only relative keys ("rel") or (iii) both types of keys ("mix"). Given that an input key φ can be either absolute or relative, we have a total of six test cases. The results obtained in these experiments are summarized in Figure 4(a). For a small set Σ with about 5 XML keys, the execution takes 0.2ms in average, whereas for a large set of about 100 XML keys, the execution takes 1.7ms in average. This indicates that our implementation of the implication algorithm is practically efficient and scales well regardless of the type of XML keys considered.

Note that the resulting running time is slightly lower when φ is an absolute key and Σ is composed by either absolute or relative keys. This is mainly due to the fact that $Q_\varphi = \varepsilon$, which means that the construction of the mini-tree involves less steps and that the q_φ node corresponds to the root node, making it unnecessary to perform a search for such node. On the other hand, the performance shown by the "abs-rel" curve in Figure 4(a) is slightly degraded due to the fact that, in general, the algorithm needs to traverse more nodes to determine whether q_φ is reachable from q'_φ. This is consistent with the way in which the witness graphs are defined.



(a) XML key implication (all cases)



(b) Increased number of wildcards

Fig. 4. Performance of the Algorithm for the Implication of XML Keys

To isolate the effect of wildcards in the performance of the algorithm, we duplicated the number of wildcards in the keys of the test case "mix-rel", replacing some of the labels in the context and target paths of those keys by the variable length wildcard. Figure 4(b) shows the running times for the original set of keys (curve "mix-rel") and the set with increased number of wildcards (curve "wildcard"). The results show that the presence of wildcards in the path expressions increases the running time for the test sets with large number of keys, but such increase is not significant in practice.

Document Validation: Tests Results. We use the same data set as before. The aim is to determine the viability of computing non-redundant cover sets to speed up the validation of XML documents against XML keys. By validating an XML document against a set of XML key, we refer to the task of checking, for every XML key in the set, whether the document satisfies such key.

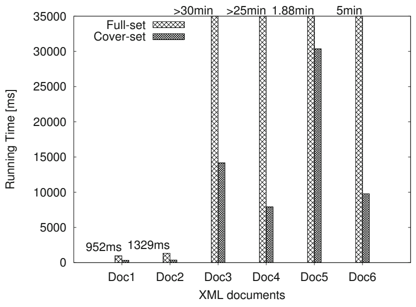
To validates XML keys, we use a naive algorithm that parses the XML document into a DOM tree and then evaluates the XML keys on the resulting tree, by using XPath queries to express their context, target and key paths. We do need to use sophisticated validation algorithms such as [5,12], since the proposed optimization based in cover sets is independent of the particular algorithm used for XML key validation.

The results (in milliseconds (ms)) obtained from the computation of non-redundant cover sets is summarized in Figure 5 (a). We emphasize that the behavior of the Algorithm 3 is linear in practice. For example, for a set of 146 keys, calculating a non-redundant cover set takes around 155ms. A total of 102 keys are discarded reducing the set to 44 keys.

Figure 5 (b) shows the optimization achieved by pre-calculating non-redundant cover sets during the validation process of the documents in Table 1. The results

XML doc.	Key Set	Time[ms]
321gone & yahoo (Doc1)	Processed Keys: 23 Discarded keys: 15 Cover set: 8 keys	3.458
DBLP (Doc2)	Processed Keys: 36 Discarded keys: 24 Cover set: 12 keys	12.757
nasa (Doc3)	Processed Keys: 35 Discarded keys: 28 Cover set size: 7 keys	9.23
Sigmod Record (Doc4)	Processed Keys: 24 Discarded Keys: 19 Cover set: 5 keys	5.294
mondial (Doc5)	Processed Keys: 26 Discarded Keys: 16 Cover set: 10	4.342

(a) Non-redundant Cover Sets.



(b) Validation Against Cover Sets.

Fig. 5. Non-redundant Cover Sets of XML keys and Validation of XML Documents

indicate that the running time required to compute a non-redundant cover set is just a tiny fraction of the overall running time required to validate a single XML document against a key. Note that in most cases the validation time can be significantly reduced by pre-computing the non-redundant covers. This can be clearly observed in the case of the DBLP document ('Doc3') and Nasa document ('Doc4'). In these cases the running time of the validation against the original set of XML keys is approximately 63 times greater than the running time of the validation against its non-redundant cover-set.

7 Conclusion

Our research was motivated by two objectives. Firstly, we wanted to demonstrate that there are expressive classes of XML keys that are not only tractable in theory but can be reasoned about efficiently in practice. For that we studied a fragment of XML keys as originally introduced by Buneman et al. [3,4], namely the class $\mathcal{K}$ of XML keys with nonempty sets of simple key paths. For these keys it was known that their implication problem can be decided in quadratic time in theory [9]. Here we have presented an efficient implementation thereof, and our experiments show that it also runs fast in practice and scales well.

Secondly, we wanted to show that our observations on the problem of deciding implication is not only of interest for the problem itself but has immediate consequences for other perennial tasks in XML database management. As an example we study the problem of validating an XML document against a set of XML keys. We have presented an optimization method for this validation that computes a non-redundant cover for the set of XML keys given as input so that satisfaction only needs to be checked for the keys in this cover. This can reduce the number of keys significantly, and our experiments show that enormous time savings can be achieved in practice. This holds true even though the validation procedure is able to decide value equality among element nodes with complex content as this is required for the XML keys studied here (and distinguishes them from the keys defined in XML Schema). This illustrates the advantage of having efficient reasoning capabilities at hand for integrity constraints.

We would like to emphasize that the use of non-redundant covers does not depend on the particular choice of the XML fragment but can be tailored to any class of XML constraints for which the implication problem can be solved efficiently. We plan to extend our studies to other expressive classes of XML keys and related constraints such as those studied in [10,7].

References

1. Apparao, V., et al.: Document object model (DOM) level 1 specification, W3C recommendation (1998), `http://www.w3.org/TR/REC-DOM-Level-1/`
2. Arenas, M., Fan, W., Libkin, L.: What's Hard about XML Schema Constraints? In: Hameurlain, A., Cicchetti, R., Traunmüller, R. (eds.) DEXA 2002. LNCS, vol. 2453, pp. 269–278. Springer, Heidelberg (2002)

3. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.: Keys for XML. Computer Networks 39(5), 473–487 (2002)
4. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.: Reasoning about keys for XML. Inf. Syst. 28(8), 1037–1063 (2003)
5. Chen, Y., Davidson, S., Zheng, Y.: Xkvalidator: a constraint validator for XML. In: CIKM 2002: Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, pp. 446–452. ACM (2002)
6. Clark, J., DeRose, S.: XML path language (XPath) version 1.0, W3C recommendation (1999), `http://www.w3.org/TR/xpath`
7. Ferrarotti, F., Hartmann, S., Link, S.: A Precious Class of Cardinality Constraints for Flexible XML Data Processing. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 175–188. Springer, Heidelberg (2011)
8. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. Trans. Database Syst. 30(2), 444–491 (2005)
9. Hartmann, S., Link, S.: Efficient reasoning about a robust XML key fragment. ACM Trans. Database Syst. 34(2) (2009)
10. Hartmann, S., Link, S.: Numerical constraints on XML data. Inf. Comput. 208(5), 521–544 (2010)
11. Jungnickel, D.: Graphs, Networks and Algorithms. Springer (1999)
12. Liu, Y., Yang, D., Tang, S., Wang, T., Gao, J.: Validating key constraints over XML document using XPath and structure checking. Future Generation Comp. Syst. 21(4), 583–595 (2005)
13. Maier, D.: Minimum Covers in the Relational Database Model. J. ACM 27, 664–674 (1980)
14. Suciu, D.: XML Data Repository, University of Washington (2002), `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`
15. Thompson, H., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1: Structures Second Edition, W3C Recommendation (2004), `http://www.w3.org/TR/xmlschema-1/`