The Finite Implication Problem for Expressive XML Keys: Foundations, Applications, and Performance Evaluation

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

[1] Victoria University of Wellington
flavio.ferrarotti@vuw.ac.nz
[2] Clausthal University of Technology
sven.hartmann@tu-clausthal.de
[3] The University of Auckland
s.link@auckland.ac.nz
[4] Yahoo! Research
mmarin@yahoo-inc.com
[5] DERI, National University of Ireland Galway
emir.munoz@deri.org

Abstract. The increasing popularity of XML for persistent data storage, processing and exchange 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. In this paper we make a theoretical as well as a practical contribution to this area. This endeavour is ambitious given the multitude of intractability results that have been established. Our theoretical contribution is based in the definition of a new fragment of XML keys that keeps the right balance between expressiveness and efficiency of maintenance. More precisely, we characterize the associated implication problem axiomatically and develop a low-degree polynomial time decision algorithm. In comparison to previous work, this new fragment of XML keys provides designers with an enhanced ability to capture properties of XML data that are significant for the application at hand. Our practical contribution includes an efficient implementation of this decision algorithm and a thorough evaluation of its performance, demonstrating that reasoning about expressive notions of XML keys can be done efficiently in practice, and scales well. Our results promote the use of XML keys on real-world XML practice, where a little more semantics makes applications a lot more effective. To exemplify this potential, we use the decision algorithm to calculate non-redundant covers for sets of XML keys. In turn, this allow us to reduce significantly the time required to validate large XML documents against keys from the proposed fragment.

1 Introduction

Keys are the most important class of integrity constraints used in database management. First of all, they provide a mechanism for identifying objects in

A. Hameurlain et al. (Eds.): TLDKS X, LNCS 8220, pp. 60–94, 2013.
© Springer-Verlag Berlin Heidelberg 2013

database instances, thus establishing an invariant relationship between objects in the real world and their representation in the database. The increasing popularity of XML [6] 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. (See [14] for a brief overview). The most influential proposal is due to Buneman et al. [7,8] who defined keys on the basis of an XML tree model similar to the one suggested by DOM [3] and XPath [10]. Figure 1 shows such a representation in which nodes are annotated by their type: E for element nodes, A for attribute nodes, and S for text nodes (PCDATA). 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 [23] subject to some minor, though essential modifications (see [4] for a discussion). Today, all major XML-enabled DBMSs, XML parsers and editors (such as XMLSpy) support keys.

The XML keys considered in this work uniquely identify nodes in an XML tree by (complex) values on some selected descendant nodes. These keys are defined using path expressions to select the relevant sets of nodes. In Figure 1, a clear example of a key is that a *bank* node can be uniquely identified by the value of its *name* attribute. This is an instance of an absolute key which is satisfied if it holds globally in the entire XML tree. We also work with relative keys, which are satisfied if they hold locally within some subtrees. For instance, in every subtree rooted at an *account* node, a *transaction* node can be uniquely identified by the value of its child node, which can either be a *withdrawal* node or a *deposit* node. This XML key might not hold in the entire tree since there might well be different withdrawals or deposits made at the same time, on the same day and for the same amount, as long as they do not belong to the same account. We emphasize that the element nodes *withdrawal* and *deposit* have complex content. Thus, checking whether two *transaction* nodes violate this latter key involves testing whether the subtrees rooted at their respective child nodes are isomorphic to one another, with the identity on string values.

1.1 Motivation

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. If the implication of XML keys can be decided efficiently, then it is possible to take advantage not only of those keys that were specified explicitly by the database designer, but also of those ones that were specified implicitly. Among other things, this is important for minimizing the cost of validating that an

⟨**bank name=** *"ANZ"*⟩⟨**branch name=** *"Downtown"*⟩
⟨**client cno=** *"JOH23144"*⟩⟨**account no=** *"2144964"*, **kind=** *"savings"*⟩
⟨**transaction**⟩⟨**withdrawal**⟩
⟨**date**⟩ *"2012.02.04"*⟨**/date**⟩⟨**time**⟩ *"13:01:03"*⟨**/time**⟩⟨**amount**⟩ *"$500.00"*⟨**/amount**⟩
⟨**/withdrawal**⟩⟨**/transaction**⟩
⟨**transaction**⟩⟨**deposit**⟩
⟨**date**⟩ *"2012.02.04"*⟨**/date**⟩⟨**time**⟩ *"14:15:03"*⟨**/time**⟩⟨**amount**⟩ *"$302.34"*⟨**/amount**⟩
⟨**/deposit**⟩⟨**/transaction**⟩
⟨**/account**⟩⟨**/client**⟩
⟨**/branch**⟩⟨**/bank**⟩
⟨**bank name=** *"BNZ"*⟩⟨**branch name=** *"Northshore"*⟩
...
⟨**/branch**⟩⟨**/bank**⟩

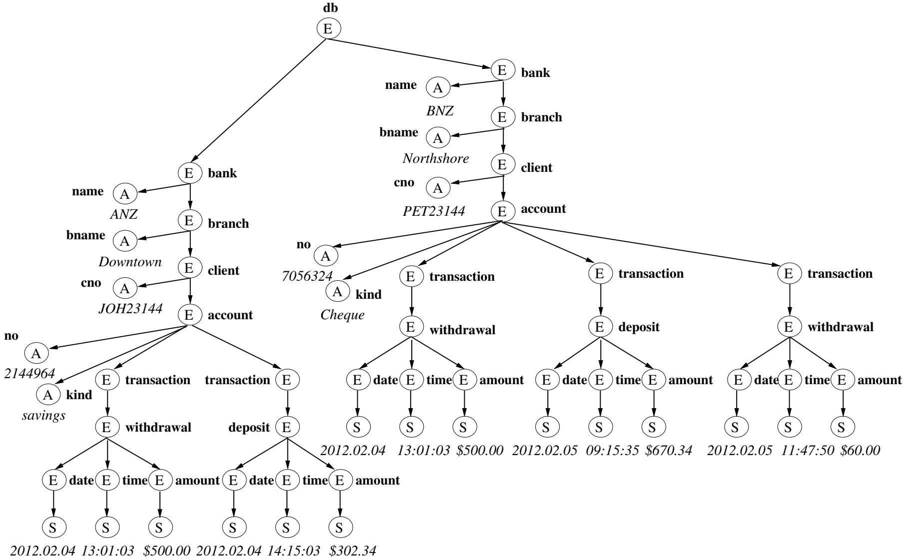


Fig. 1. XML data fragment and its tree representation

XML document satisfies a set of keys gathered as business rules during requirements engineering.

Example 1. Suppose, a database designer has already specified an XML key φ_a which expresses that in every subtree rooted at a *bank* node, a *client* node can be uniquely identified by its child attribute *cno* together with its child node *account*. Later on, another of the database designers discovers that the *cno* attribute can by itself be used to identify a *client* node relatively to a *bank* node, and thus defines a new suitable XML key φ_b. It is easy to see that if φ_b is satisfied by an XML tree T, then also the former key φ_a is satisfied by T. That is, φ_b implies φ_a. Thus, we can validate an XML tree T against both keys by just checking whether T satisfies φ_b, since T will satisfy both φ_a and φ_b iff it satisfies φ_b. We would like to emphasize that the *account* nodes have complex content. Thus, checking whether two *client* nodes in T violate φ_a is quite costly in terms

of time, since it involves testing whether the subtrees rooted at their *account* nodes are isomorphic to one another, with the identity on string values. In contrast, checking whether two *client* nodes violate φ_b only involves checking equality on the text nodes of their respective *cno* attributes. □

No less important, facilities for reasoning about XML keys present numerous opportunities for designing XML databases and views that permit a more efficient processing of frequent queries and updates.

Example 2. Let us consider the following XQuery, which expresses over XML trees with the structure of the tree in Figure 1, a client request for her transactions on an specific account and date.

```
for $a in doc("transactions.xml")//account[@no]="2144964"
where $a//date/text() ="2012.02.04"
return ⟨transactions⟩{$a/transaction/*}⟨/transactions⟩
```

Also, let us assume that this is the most frequent type of query. It is expensive to process an XQuery such as this one over an XML tree with the layout of the tree in Figure 1, since for all *transaction* nodes of the selected account, it has to be decided if it has a descendant *date* node which corresponds to the specified date. This is due to the fact that there can be many transactions on the selected account for the given date, and thus a *transaction* node cannot be uniquely identified by its descendant *date* node. Note that this key is therefore not implied by the keys that apply to the XML tree. Also note that if the *transaction* nodes are encrypted, then the evaluation of the query requires us to decrypt every single one of them for the selected account. In this case, the layout illustrated in Figure 2 represents a better choice. The target *transaction* nodes are now grouped together as child nodes of a target *txdate* node. Redundancies with respect to *dates* for a given account are eliminated, which results in a better overall design. Moreover, only those *transaction* nodes which are child nodes of the selected *txdate* node need to be decrypted. Note that these improvements are due to the existence of a new relative key which specifies that, in every subtree rooted at an *account* node, a *txdate* node can be uniquely identified by the value of its *date* attribute. The original XQuery is rewritten into

```
for $a in doc("transactions.xml")//account[@no]="2144964"
where $a/txdate[@date] ="2012.02.04"
return ⟨transactions⟩{$a/txdate/transaction/*}⟨/transactions⟩
```

□

A further example of an area in which the implication of XML keys is of tremendous benefit is semantically rich data exchange.

Example 3. Suppose we need to share part of the information on bank transactions with the Reserve Bank, and that due to legal requirements we need to eliminate the information regarding clients and individual accounts. Thus, we

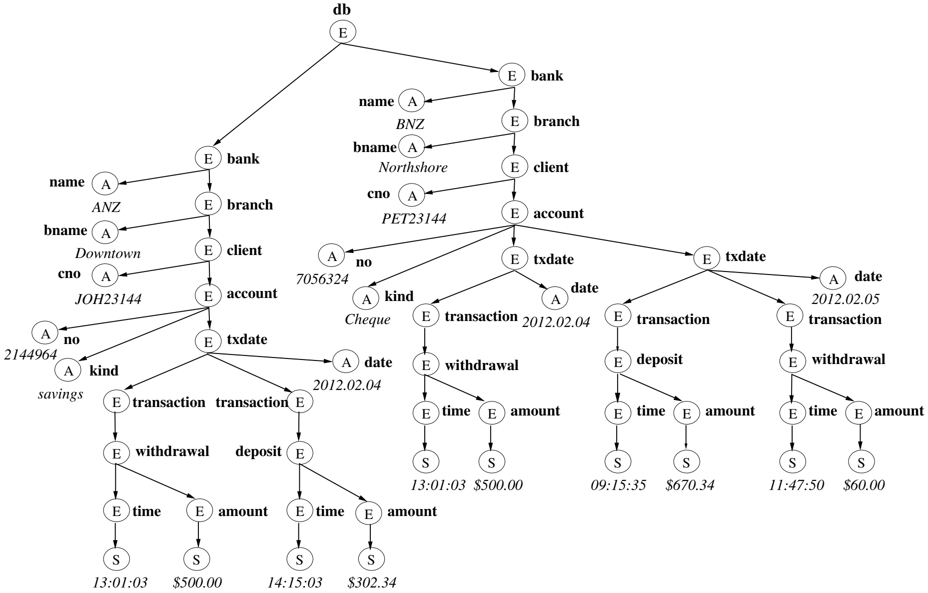


Fig. 2. XML data fragment with improved layout

generate a view over the XML tree which skips the *no* attribute of the *account* nodes and the *cno* attribute of the *client* nodes. Clearly, it would be useful to provide the Reserve Bank with the set of keys defined for the original XML tree. In light of this additional semantic information, they could better interpret the XML tree. For instance, they could deduce from those keys that a *client* node was identifiable in the original XML document by a *cno* attribute relatively to a *bank* node. Furthermore, they could check whether the specified keys allow one to conclude further keys which are useful for extraction and processing of the data in the provided XML document. For instance, they could check whether the specified keys allow one to conclude a further key stating that a *client* node can be identified by its descendant *account* nodes relatively to a *bank* node. □

1.2 Related Work

The definition of keys, adopted by the W3C for XML Schema [23], is currently the industry standard for specifying keys. However, Arenas et al. [4] 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. [7] 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 [7] for discussion). In particular, we have given examples of XML keys that require

a less restricted notion of equality, since they require us to test equality on element nodes such as *withdrawal* and *deposit* which are not 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 [7,8,15,16,11].

In practice, however, expressive yet tractable notions of XML keys have been mostly ignored. 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. [9,18]), none of them make use of the reasoning capabilities of XML keys as suggested in Example 1.

Aiming to fill this gap between theory and practice, we initiated in [12] an empirical study of an XML key fragment, namely the fragment of XML keys with nonempty sets of simple key paths. As shown in [8,15], automated reasoning about this XML key fragment can be done efficiently, in theoretical terms. Our work confirmed this fact in practice. This article extends these earlier works by considering a strictly more expressive fragment of XML keys which allows the use of single- and variable-length wildcards in the path language used to specify the keys, as well as empty sets of key paths for the specification of structural keys which identify nodes (within subtrees) by unique paths instead of unique data value. In fact, we establish a fragment that strictly includes the already expressive fragments of XML keys explored in [16,11] without resigning efficiency.

1.3 Contributions

Evidently, the expressiveness (but also the tractability) of a fragment of XML keys will depend on the query language used to navigate between nodes in trees. As common in XML data processing, path expressions are used to select the nodes of interest. The preferred languages for selection queries against XML data are XPath queries [10] and fragments thereof. The flexibility provided by these languages is probably not needed for every application, but there are many applications where flexibility is essential. Taken that XML is widely used to represent heterogeneous data, the expressiveness that comes with wildcards and descendant queries is highly appreciated. One may think of data integration where data from different sources are stored in the same document. Though not uniformly structured, the data should still be analyzed using the same 'one fits all' query to avoid extensive maintenance costs. This can only be achieved when using a sufficiently rich query language.

Thus, our first contribution establishes a new fragment of XML keys (namely Max-Keys) that keeps the right balance between expressiveness and efficiency of maintenance. More precisely, we characterize the associated implication problem axiomatically, and propose a low-degree polynomial time decision algorithm. The set of XML keys that are expressible in this new fragment includes strictly the sets of XML keys that are expressible in the fragments considered in

previous proposals [15,16,11,12]. The first source of expressiveness results from the very general notion of value equality: two element nodes v and w are considered value equal, if the subtrees rooted at v and w are isomorphic by an isomorphism that is the identity on string values. This contrasts with more restrictive notions, for instance with value equality on leaf nodes. The second source of expressiveness is a result of the path language we use to select nodes. This includes the single-label wildcard, child navigation, and descendant navigation as known from XPath. Note that in [12] we have considered XML keys whose context and target nodes can be selected by a path language that uses child navigation and descendant navigation, and key nodes by using only child navigation. While this fragment of XML keys can already capture many desirable properties, the Max-Keys fragment proposed in this paper captures a considerably more expressive class of properties by allowing the use of single-label wildcards, a controlled use of variable-length wildcards in the key paths and XML keys without key paths that allow one to express structural keys.

Our second contribution concerns the exploration of this theoretical ideas in practice. We develop and implement an efficient algorithm that decides the implication problem for Max-Keys and thoroughly evaluate its performance. Our performance tests give empirical evidence that reasoning about expressive notions of XML keys is practically efficient, and scales well. In fact, the performance of this new algorithm is comparable to the performance of the algorithm presented in [12] for a strictly less expressive fragment of XML keys. Our results unleash XML keys on real-world XML practice, where a little more semantics makes applications a lot more effective.

There is indeed great potential for practical uses of the proposed decision algorithm. For instance, the process of checking XML data integrity against XML keys can benefit a lot from the ability to decide implication efficiently. Clearly, if a set Σ of XML keys implies an XML key φ, and we have already checked that an XML data tree satisfies Σ, then there is no need to test φ any more, saving considerable resources. Thus, 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 $\Sigma - \{\sigma\}$. Same than for the less expressive fragment of XML keys studied in [12], our experiments show that the time to compute a non-redundant cover for a given set of keys in the fragment of Max-Keys, is just a small fraction of the average time needed to validate an XML document against a single key.

We would like to note that the experiments carried out in this article extend the experiments presented in [12] in several ways. Firstly, we consider XML keys which are more expressive than the keys studied in our previous paper. Secondly, we test the new implication algorithm by considering even larger sets of XML keys (of up to 180 keys). Thirdly, we include a new XML document which holds the complete Chilean electoral roll in our validation experiments. This XML document, which contains 3.2GB of data, is several orders of magnitude larger than the XML documents usually considered in previous works in this area.

Finally, we compute all non-redundant covers that are subsets of a given set of XML keys, and rank them according to the time it takes for their validation. This provides valuable information which can be used in practice to design XML documents which are more efficient to validate. For replication purposes, all the data sets used in our experiments as well as the full set of results can be downloaded from `http://emir-munoz.github.com/xml-constraints`.

1.4 Organization

We recall basic concepts and fix the formal notation in Section 2. In Section 3 we define the central notion of XML key and introduce a new and expressive fragment of XML keys (namely the Max-Keys). We establish a finite set of inference rules for deriving new Max-Keys in Section 4 and prove its completeness in Section 5. In Section 6, we presents an efficient algorithm for deciding implication of Max-Keys as well as an implementation thereof. We present the results regarding the application and performance evaluation of our decision algorithm for the finite implication problem of Max-Keys, as well as an strategy to use this algorithm in the context of XML document validation, in Section 7. We conclude the paper in Section 8 with final remarks.

2 Preliminaries

2.1 XML Data Representation

We use the common representation of XML data as ordered, node-labeled trees.

Let $\mathbf{E}$ denote a countably infinite set of element tags, $\mathbf{A}$ a countably infinite set of attribute names, and $\{S\}$ a singleton set denoting text (PCDATA). These sets are pairwise disjoint. The elements of $\mathcal{L} = \mathbf{E} \cup \mathbf{A} \cup \{S\}$ are called *labels*.

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 $V \to \mathcal{L}$ 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 $lab(v) = S$. 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, and $att(v)$ is a set of attribute nodes in V. If v is an attribute or text node, then $ele(v)$ and $att(v)$ are undefined. The partial mapping val assigns a string to each attribute and text node: for each node $v \in V$, $val(v)$ is a string if v is an attribute or text node, while $val(v)$ is undefined otherwise. Finally, r is the unique and distinguished root node.

For a node $v \in V$, each node w in $ele(v)$ or $att(v)$ is called a *child* of v, and we say that there is an edge (v, w) from v to w in T. A *path* p of T is a finite sequence of nodes $v_0, \ldots, v_m$ in V such that (v_{i-1}, v_i) is an edge of T for $i = 1, \ldots, m$. The path p determines a word $lab(v_1). \cdots .lab(v_m)$ over the alphabet $\mathcal{L}$, denoted by $lab(p)$. For a node $v \in V$, each node w reachable from v is called a *descendant* of v. Note that every XML tree has a tree structure: for each node $v \in V$, there is a unique path from the root node r to v.

2.2 Value Equality of Nodes on XML Trees

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. More formally, $u =_v v$ whenever the following conditions are satisfied:

a. $lab(u) = lab(v)$.
b. If u, v are attribute or text nodes, then $val(u) = val(v)$.
c. If u, v are element nodes, then *(i)* if $att(u) = \{a_1, \ldots, a_m\}$, then $att(v) = \{a'_1, \ldots, a'_m\}$ and there is a permutation π on $\{1, \ldots, m\}$ such that $a_i =_v a'_{\pi(i)}$ for $i = 1, \ldots, m$, and *(ii)* if $ele(u) = [u_1, \ldots, u_k]$, then $ele(v) = [v_1, \ldots, v_k]$ and $u_i =_v v_i$ for $i = 1, \ldots, k$.

Note that the notion of value equality takes the document order of the XML tree into account. We remark that $=_v$ is an equivalence relation on the node set V of the XML tree. As an example, the first and third *transaction* nodes (from left to right) in Figure 1 are value equal while all the remaining *transaction* nodes are not value equal to each other.

2.3 Node Selection Queries on XML Trees

Regular paths have been widely used to express queries for selecting nodes in XML trees. In the sequel, we use the path language $PL^{\{.,_,_^*\}}$ consisting of expressions given by the following grammar:

$$Q \to \ell \mid \varepsilon \mid Q.Q \mid _ \mid _^*$$

Herein, $\ell \in \mathcal{L}$ is any label, ε denotes the empty path expression, "." denotes the concatenation of two path expressions, "_" denotes the *single-label* wildcard, and "$_^*$" denotes the *variable-length* wildcard.

Let P, Q be words from $PL^{\{.,_,_^*\}}$. P is a *refinement* of Q, denoted by $P \lesssim Q$, if P is obtained from Q by replacing variable-length wildcards in Q by words from $PL^{\{.,_,_^*\}}$ and single-label wildcards in Q by labels from $\mathcal{L}$. For example, *bank.branch.account* is a refinement of *bank._.account*. Note that $\lesssim$ is a pre-order on $PL^{\{.,_,_^*\}}$. Let $\sim$ denote the congruence induced by the identity $_^*._^* = _^*$ on $PL^{\{.,_,_^*\}}$, and observe that $P \sim Q$ holds if and only if P and Q are refinements of each other. For example, $bank._^*.account \sim bank._^*._^*.account$.

Regular paths allow one to navigate in an XML tree. We briefly recall the semantics of expressions from $PL^{\{.,_,_^*\}}$ in the context of XML. Let Q be a word from $PL^{\{.,_,_^*\}}$. A path p in the XML tree T is called a Q-path if $lab(p)$ is a refinement of Q. For nodes $v, w \in V$, we write $T \models Q(v, w)$ if w is reachable from v following a Q-path in T.

For a node v in the XML tree T, let $v[\![Q]\!]$ denote the set of nodes in T that are reachable from v following any Q-path, that is, $v[\![Q]\!] = \{w \mid T \models Q(v, w)\}$. In particular, we use $[\![Q]\!]$ as an abbreviation for $r[\![Q]\!]$ where r is the root node.

For a subset $\mathcal{Z} \subseteq \{., _, _^*\}$, let $PL^{\mathcal{Z}}$ denote the subset of $PL^{\{.,_,_^*\}}$ with expressions restricted to the constructs in $\mathcal{Z}$. In particular, $PL^{\{.\}}$ is the set of simple path expression without wildcards.

Since attribute and text nodes in an XML tree T are always leaves, $Q \in PL^{\{.,_,_^*\}}$ is *valid* only if it has no labels $\ell \in \mathbf{A}$ or $\ell = S$ in a position other than the terminal one. Note that each prefix of a valid Q is valid, too.

Let P, Q be words from $PL^{\{.,_,_^*\}}$. P is *contained* in Q, denoted by $P \subseteq Q$, if for every XML tree T and every node v of T we have $v[\![P]\!] \subseteq v[\![Q]\!]$. It follows immediately from the definition that $P \lesssim Q$ implies $P \subseteq Q$.

We work with the quotient set $PL^{\{.,_,_^*\}}_{/\sim}$ rather than with $PL^{\{.,_,_^*\}}$ directly: A word from $PL^{\{.,_,_^*\}}$ is in *normal form* if it has no consecutive variable-length wildcards, i.e., if it has no consecutive "$_^*$" and no occurrence of "$_^*._$". Note that, each congruence class contains a unique word in normal form. Each word from $PL^{\{.,_,_^*\}}$ can be transformed into normal form in linear time, just by removing superfluous variable-length wildcards and replacing each occurrence of "$_^*._$" by "$_._^*$". The *length* $|Q|$ of a $PL^{\{.,_,_^*\}}$ expression Q is the number of labels in Q plus the number of wildcards (counting both variable-length and single-label wildcards) in the normal form of Q.

The empty path expression ε has length 0. The natural homomorphism from $PL^{\{.,_,_^*\}}$ to $PL^{\{.,_,_^*\}}_{/\sim}$ is an isomorphism when restricted to words in normal form. By abuse of notation we use the words from $PL^{\{.,_,_^*\}}$ to denote their respective congruence class.

For nodes v and v' of an XML tree T, the *value intersection* of $v[\![Q]\!]$ and $v'[\![Q]\!]$ is given by $v[\![Q]\!] \cap_v v'[\![Q]\!] = \{(w, w') \mid w \in v[\![Q]\!], w' \in v'[\![Q]\!], w =_v w'\}$. That is, $v[\![Q]\!] \cap_v v'[\![Q]\!]$ consists of all those node pairs in T that are value equal and are reachable from v and v', respectively, by following Q-paths.

3 Keys for XML Data

The expressiveness of our fragment of XML keys results from the generality of our notion of value-equality and that of the path language. For more expressive path languages the containment problem becomes at least intractable [20].

Definition 1. *An XML key φ in is an expression of the form*

$$(Q_\varphi, (Q'_\varphi, \{P_1^\varphi, \ldots, P_{k_\varphi}^\varphi\}))$$

where Q_φ, Q'_φ, P_1^φ, ..., $P_{k_\varphi}^\varphi \in PL^{\{.,_,_^*\}}$ such that $Q_\varphi.Q'_\varphi.P_1^\varphi$, ..., $Q_\varphi.Q'_\varphi.P_{k_\varphi}^\varphi$ are valid, and where k_φ is a non-negative integer.

An XML tree T satisfies φ if and only if $\forall q \in [\![Q_\varphi]\!]\ \forall q'_1, q'_2 \in q[\![Q'_\varphi]\!]$

$$\left(\bigwedge_{1 \le i \le k} q'_1[\![P_i^\varphi]\!] \cap_v q'_2[\![P_i^\varphi]\!] \neq \emptyset \right) \Rightarrow q'_1 = q'_2.$$

Herein, Q_φ is called the context path*, Q'_φ is called the* target path*, and $P_1^\varphi, \ldots, P_{k_\varphi}^\varphi$ are called the* key paths *of φ.*

By the previous definition, if the set of key paths is non empty (i.e. $k \ge 1$), a key φ is satisfied by a tree T if and only if for every node $q \in [\![Q_\varphi]\!]$ and all

nodes $q'_1, q'_2 \in q[\![Q'_\varphi]\!]$ such that there are nodes $x_i \in q'_1[\![P_i^\varphi]\!], y_i \in q'_2[\![P_i^\varphi]\!]$ with $x_i =_v y_i$ for all $i = 1, \ldots, k$, then $q'_1 = q'_2$. On the other hand, if the set of key path is empty, a key φ is satisfied by a tree T if and only if for every node $q \in [\![Q_\varphi]\!]$ there is at most one node reachable from q by following a Q'_φ-path, i.e., $q[\![Q'_\varphi]\!]$ contains at most one element. Thus, these latter keys identify nodes (within certain subtrees) by a unique path while the keys with nonempty sets of key paths identify nodes (within subtrees) by unique data values. Hence, we introduce the term *structural keys* for keys that have an empty set of key path expressions.

Example 4. We formalize the XML keys discussed in the introduction over XML trees with the layout of the tree T depicted in Figure 1.

a. $(\varepsilon, (bank, \{name\}))$ expresses "a *bank* node can be uniquely identified by the value of its *name* attribute". Over trees with this layout, the same can be expressed by $(\varepsilon, (_, \{name\})$ and also by $(\varepsilon, (_, \{_^*.name\}))$ among others.
b. $(_^*.account, (transaction, \{_\}))$ expresses "in every subtree rooted at an *account* node, a *transaction* node can be uniquely identified by the value of its child node, which can either be a *withdrawal* node or a *deposit* node". As an alternative over trees with this layout, we could use $(_._._.account, (transaction, \{_\}))$ and $(bank.branch.client.\ account, (transaction, \{_\}))$ among others.
c. $(_^*.bank, (_^*.client, \{cno, account\}))$ expresses "in every subtree rooted at a *bank* node, a *client* node can be uniquely identified by its child attribute *cno* together with its child node *account*". Over trees with this layout the same is expressed, among others, by $(_, (_.client, \{cno, account\}))$.
d. $(_^*.transaction, (_, \emptyset))$ expresses "Every *transaction* node has at most one child node".

3.1 Expressive and Tractable Fragments of XML Keys

In order to take advantage of XML keys effectively it becomes necessary to reason about them efficiently. Central to this task is the implication problem. Let $\Sigma \cup \{\varphi\}$ be a finite set of XML keys in a fragment $\mathcal{C}$. We say that Σ *(finitely) implies* φ, denoted by $\Sigma \models_{(f)} \varphi$, if and only if every (finite) XML tree T that satisfies all $\sigma \in \Sigma$ also satisfies φ. The *(finite) implication problem* for the fragment $\mathcal{C}$ is to decide, given any finite set of XML keys $\Sigma \cup \{\varphi\}$ in $\mathcal{C}$, whether $\Sigma \models_{(f)} \varphi$. If Σ is a finite set of XML keys in $\mathcal{C}$ let $\Sigma^*_{(f)}$ denote its *(finite) semantic closure*, i.e., the set of all XML keys (finitely) implied by Σ. That is, $\Sigma^*_{(f)} = \{\varphi \in \mathcal{C} \mid \Sigma \models_{(f)} \varphi\}$.

As shown in [7,8,15,16,11], efficient decision algorithm for the finite implication problem of XML keys exist for the following fragments.

$$\mathcal{K}_1 = \{(Q, (Q', \{P_1, \ldots, P_k\})) \mid k \geq 1, Q, Q' \in PL^{\{.,_^*\}} \text{ and } P_1, \ldots, P_k \in PL^{\{.\}}\}$$
$$\mathcal{K}_2 = \{(Q, (Q', \{P_1, \ldots, P_k\})) \mid k \geq 0, Q, Q' \in PL^{\{.,_^*\}} \text{ and } P_1, \ldots, P_k \in PL^{\{.\}}\}$$
$$\mathcal{K}_3 = \{(Q, (Q', \{P_1, \ldots, P_k\})) \mid k \geq 1, Q, Q' \in PL^{\{.,_,_^*\}} \text{ and } P_1, \ldots, P_k \in PL^{\{.,_\}}\}$$

Note that the fragment $\mathcal{K}_2$ includes the fragment $\mathcal{K}_1$ plus structural XML keys with context and target paths in $PL^{\{.,_^*\}}$. The fragment $\mathcal{K}_3$ is defined using a more expressive path language than $\mathcal{K}_1$ which allows the specification of single-label wildcards. Thus, it also strictly includes the fragment $\mathcal{K}_1$. Regarding the relationship between $\mathcal{K}_2$ and $\mathcal{K}_3$, it is clear that there are XML keys in $\mathcal{K}_2$ which are not in $\mathcal{K}_3$ and vice versa. For instance, the key $(\varepsilon, (_, \{name\}))$ belongs to $\mathcal{K}_3$ and does not belong to $\mathcal{K}_2$. On the contrary, the structural key $(bank.branch.client.account.transaction, (deposit, \emptyset))$ belongs to $\mathcal{K}_2$ and does not belong to $\mathcal{K}_3$.

In this work, we study a fragment of XML keys which strictly includes $\mathcal{K}_1$, $\mathcal{K}_2$ and $\mathcal{K}_3$. We define it as follows.

$$\text{Max-Keys} = \{\ (Q, (Q', \{P_1, \dots, P_k\})) \,|\, k \geq 0, Q, Q', P_1, \dots, P_k \in PL^{\{.,_,_^*\}}$$
$$\text{but such that } Q' \text{ or } P_1, \dots, P_k \in PL^{\{.,_\}}\}$$

It is easy to find examples which belong to Max-Keys and do not belong to any of the fragments $\mathcal{K}_1$, $\mathcal{K}_2$ and $\mathcal{K}_3$. Take for instance $(\varepsilon, (_, \{_^*.name\}))$ and $(_^*.transaction, (_, \emptyset))$ from Example 4.

The fragment Max-Keys of XML keys provides XML engineers with an enhanced ability to capture interesting properties of XML data. These are useful for several tasks in XML practice such as data integration, cleaning and validation among others. In the remainder of the article we will establish that despite its expressiveness, the fragment of Max-Keys can be reasoned about efficiently.

For the time being, we assume the number of key paths k to be ≥ 1 for all Max-Keys. We consider the case of Max-Keys with empty sets of key paths latter in Subsection 5.4.

The plan to verify the tractability of Max-Keys is as follows: First we will characterize the implication problem associated with Max-Keys in terms of a finite axiomatization. We can speak of *the* implication problem as the finite and unrestricted implication problems coincide for the fragment of Max-Keys. The axiomatization provides complete insight into the interaction of Max-Keys. This insight allows us to characterize the implication problem by constructive graph properties. This characterization enables us to establish a compact, low-degree polynomial-time algorithm for deciding implication.

4 Inference Rules for Max-Keys

Our goal is to establish a finite axiomatization for the implication of Max-Keys. To begin with we assemble a set of inference rules that allow us to derive new Max-Keys from given ones. Derivability with respect to a set $\mathfrak{R}$ of inference rules, denoted by the binary relation $\vdash_{\mathfrak{R}}$ between a set of Max-Keys and a single Max-Key, can be defined analogously to the notion in the relational data model [1, pp. 164-168].

We aim to find a set of inference rules which is *sound* and *complete* for the implication of Max-Keys. A set $\mathfrak{R}$ of inference rules is sound (respectively, complete) for the implication of Max-Keys if for all finite sets Σ of Max-Keys we

have $\Sigma^+_{\mathfrak{R}} \subseteq \Sigma^*$ (respectively, $\Sigma^* \subseteq \Sigma^+_{\mathfrak{R}}$). Herein, $\Sigma^+_{\mathfrak{R}} = \{\varphi \mid \Sigma \vdash_{\mathfrak{R}} \varphi\}$ denotes the *syntactic closure* of Σ under derivation by $\mathfrak{R}$.

Table 1 shows the set of inference rules for the implication of Max-Keys. Each inference rule has the form $\frac{premises}{conclusion} condition$ with premises from Max-Keys. That is, the path expressions used in the premises are always chosen such that the respective XML key lies in the fragment of Max-Keys.

Table 1. A Finite Axiomatization for Max-Keys

$\dfrac{(Q,(Q',S))}{(Q,(Q',S\cup\{P\}))}\ Q' \text{ or } P\in PL^{\{.,_\}}$ (prefix-epsilon)	$\dfrac{}{(Q,(\epsilon,S))}$ (epsilon)
$\dfrac{(Q,(Q'.Q'',S))}{(Q.Q',(Q'',S))}$ (target-to-context)	$\dfrac{(Q,(Q',S\cup\{\epsilon,P\}))}{(Q,(Q',S\cup\{\epsilon,P.P'\}))}$ (superkey)
$\dfrac{(Q,(Q'.P,\{P'\}))}{(Q,(Q',\{P.P'\}))}\ \text{at least 2 of } Q',P,P'\in PL^{\{.,_\}}$ (subnodes)	$\dfrac{(Q,(Q',S))}{(Q'',(Q',S))}\ Q''\subseteq Q$ (context-path-containment)
$\dfrac{(Q,(Q'.P,\{\epsilon,P'\}))}{(Q,(Q',\{\epsilon,P.P'\}))}\ \text{at least 2 of } Q',P,P'\in PL^{\{.,_\}}$ (subnodes-epsilon)	$\dfrac{(Q,(Q',S))}{(Q,(Q'',S))}\ Q''\subseteq Q'$ (target-path-containment)
$\dfrac{(Q,(Q',\{P.P_1,\ldots,P.P_k\})),\ (Q.Q',(P,\{P_1,\ldots,P_k\}))}{(Q,(Q'.P,\{P_1,\ldots,P_k\}))}$ (interaction)	$\dfrac{(Q,(Q',S\cup\{P\}))}{(Q,(Q',S\cup\{P'\}))}\ P'\subseteq P$ (key-path-containment)

We prove below the soundness of the *subnodes* rule since it provides valuable insight that explains out definition of Max-Keys. The soundness of the remaining rules can be shown using similar arguments. Therefore we omit those lengthy, but not very difficult proofs. For comparing, we also refer to our soundness proofs in [15,16,11] for the special cases of the smaller fragments $\mathcal{K}_1$, $\mathcal{K}_2$ and $\mathcal{K}_3$ of XML keys.

Lemma 1. *The* subnodes *rule is sound for the implication of XML keys in the fragment of Max-Keys.*

Proof. Suppose an XML tree T violates $(Q,(Q',\{P.P'\}))$. Then there is some node $q \in [\![Q]\!]$ and there are some nodes $q'_1, q'_2 \in q[\![Q']\!]$ such that $q'_1 \neq q'_2$ and such that there exist $p'_1 \in q'_1[\![P.P']\!]$ and $p'_2 \in q'_2[\![P.P']\!]$ where $p'_1 =_v p'_2$ holds. By definition of concatenation, there exist $p_1 \in q'_1[\![P]\!]$ and $p_2 \in q'_2[\![P]\!]$ such that $p'_1 \in p_1[\![P']\!]$ and $p'_2 \in p_2[\![P']\!]$ hold. Since T is a tree and due to the condition of the *subnodes* rule Q' or $P.P'$ is a $PL^{\{.,_\}}$ expression (i.e. has no variable-length wildcards), it holds that q'_1 is neither an ancestor nor a descendant of q'_2.

Consequently, $p_1 \neq p_2$ (since otherwise $q'_1 = q'_2$). This, however, means that T also violates $(Q, (Q'.P, \{P\}))$. □

In the proof of the previous lemma we have made an interesting observation: If we allow the use of variable-length wildcards in the target and key paths at the same time, then we could have a pair of distinct target nodes q'_1 and q'_2 so that one is an ancestor of the other one. This would allow us to build a counter-example tree to demonstrate that the *subnodes* rule is not sound for such an extended fragment of Max-Keys.

5 Axiomatic Characterization of Max-Keys Implication

Our goal is to demonstrate that the set $\mathfrak{R}$ of inference rules is also complete for the implication of Max-Keys. Completeness means we need to show that for an arbitrary finite set $\Sigma \cup \{\varphi\}$ of Max-Keys with $\varphi \notin \Sigma^+_{\mathfrak{R}}$ there is some XML tree T that satisfies all members of Σ but violates φ. That is, T is a counter-example tree for the implication of φ by Σ.

The general proof strategy is as follows: For T to be a counter-example we i) require a context node q_φ with (at least) two target nodes q'_φ that have value equal key paths $q^\varphi_1, \ldots, q^\varphi_{k_\varphi}$, and ii) must for every context node q_σ not have target nodes q'_σ with value equal key nodes $q^\sigma_1, \ldots, q^\sigma_{k_\sigma}$, for each $\sigma \in \Sigma$. Such a counter-example tree exists if and only if these two conditions can be satisfied simultaneously.

In a fist step, we represent φ as a finite node-labeled tree $T_{\Sigma,\varphi}$, which we call the *mini-tree*. Then, we reverse the edges of the mini-tree and add to the resulting tree downward edges for certain members of Σ. This final digraph $G_{\Sigma,\varphi}$ is called the *witness network*. A downward edge resulting from σ tells us that under each source node there can be at most one target node. Now, if we can reach the target node of φ from its context node along a dipath then there is no counter-example tree T. In other words, if we satisfy condition ii) above, then we cannot satisfy condition i). Otherwise, we can construct a counter-example tree T.

5.1 Witness Networks

Let $\Sigma \cup \{\varphi\}$ be a finite set of Max-Keys. Let $\mathcal{L}_{\Sigma,\varphi}$ denote the set of all labels $\ell \in \mathcal{L}$ that occur in path expressions of members in $\Sigma \cup \{\varphi\}$, and fix a label $\ell_0 \in \mathbf{E} - \mathcal{L}_{\Sigma,\varphi}$. First we transform the path expressions occurring in φ into simple path expressions in $PL^{\{.\}}$. For that purpose we replace each single-label wildcard "_" by ℓ_0 and each variable-length wildcard "_*" by a sequence of $l+1$ labels ℓ_0, where l is the maximum number of consecutive single-label wildcards that occurs in any Max-Key in $\Sigma \cup \{\varphi\}$. This transformation turns Q_φ into O_φ, Q'_φ into O'_φ, and each P^φ_i into O^φ_i for $i = 1, \ldots, k_\varphi$. The path expressions after the transformation do not contain any more wildcards (neither single-label nor variable-length ones).

The proper choice of the integer l is essential for the later construction. In particular, if there are no occurrences of single-label wildcards in the Max-Keys under consideration, then $l = 0$ and we just replace each variable-length wildcard "$_^*$" by one ℓ_0.

To continue with our construction, 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 $i = 1, \ldots, k_\varphi$, let p_i be a O_i^φ-path from a node r_i^φ to a node x_i^φ, 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_i^φ 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 $Q_i^\varphi \neq \varepsilon$, then $\mathcal{M}$ consists of the leaves of $T_{\Sigma,\varphi}$, and otherwise $\mathcal{M}$ consists of all descendant nodes of q'_φ in $T_{\Sigma,\varphi}$.

We use mini-trees to calculate the impact of Max-Keys in Σ on a possible counter-example tree for the implication of φ by Σ. To distinguish Max-Keys that have an impact from those that do not, we introduce the notion of *applicability*. Intuitively, when a Max-Key is not applicable, then we do not need to satisfy it in a counter-example tree as it does not require all its key paths. Let $T_{\Sigma,\varphi}$ be the mini-tree of the Max-Key φ with respect to Σ, and let $\mathcal{M}$ be its marking. A Max-Key σ is said to be *applicable* to φ if there are nodes $w_\sigma \in [\![Q_\sigma]\!]$ and $w'_\sigma \in w_\sigma[\![Q'_\sigma]\!]$ in $T_{\Sigma,\varphi}$ such that $w'_\sigma[\![P_i^\sigma]\!] \cap \mathcal{M} \neq \emptyset$ 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 network* $G_{\Sigma,\varphi}$ of φ and Σ as the node-labeled digraph obtained from $T_{\Sigma,\varphi}$ as follows: the nodes and node-labels of $G_{\Sigma,\varphi}$ are exactly the nodes and node-labels of $T_{\Sigma,\varphi}$, respectively. The edges of $G_{\Sigma,\varphi}$ consist of the reversed edges from $T_{\Sigma,\varphi}$. Furthermore, for each Max-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 we add a directed edge (w_σ, w'_σ) to $G_{\Sigma,\varphi}$. We refer to these additional edges as *witness edges* while the reversed edges from $T_{\Sigma,\varphi}$ are referred to as *upward edges* of $G_{\Sigma,\varphi}$. This is the case since for every witness w_σ and w'_σ the node w'_σ is a descendant of the node w_σ in $T_{\Sigma,\varphi}$, and thus the witness edge (w_σ, w'_σ) is a downward edge or loop in $G_{\Sigma,\varphi}$.

Example 5. Let us consider the following Max-Keys which were specified by the database designer for XML trees with the layout of the tree in Figure 1.

$$\begin{aligned}
\varphi &= (\varepsilon, (bank._^*.client.account, \{no, kind\})) \\
\sigma_1 &= (\varepsilon, (bank._, \{_^*.account.no\})) \\
\sigma_2 &= (bank, (_^*.client, \{_.no\})) \\
\sigma_3 &= (_^*.client, (account, \{kind\}))
\end{aligned}$$

Let $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$. The left picture of Figure 3 shows the mini-tree $T_{\Sigma,\varphi}$ and its marking (note that leaves are marked by $\times$). The right picture in the same figure shows its corresponding witness network $G_{\Sigma,\varphi}$. Note that all the Max-Keys in Σ are applicable to φ. On the other hand, the Max-Key $\sigma_4 = (\varepsilon, (bank.branch.client.account, \{no, kind\}))$ is not applicable to φ. In fact, σ_4

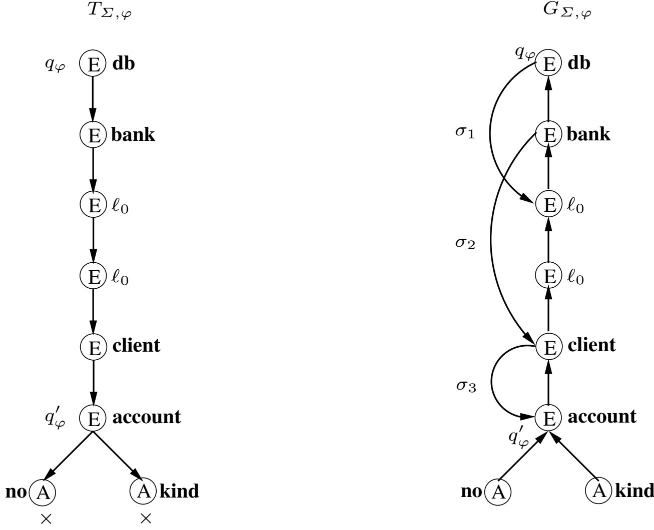


Fig. 3. A mini-tree and a witness network

is implied by φ since it can be derived by an application of the *target-path-containment* rule.

In the following section we will prove the following crucial fact. If q'_φ is reachable from q_φ in $G_{\Sigma,\varphi}$, then $\varphi \in \Sigma^+_{\mathfrak{R}}$. In other words, if φ is not derivable from Σ, then there is *no* dipath from q_φ to q'_φ in $G_{\Sigma,\varphi}$.

5.2 Reachability Implies Derivability

The following important lemma holds the key to prove the completeness of $\mathfrak{R}$ and also for our low-degree polynomial time decision algorithm for the implication problem of Max-Keys.

Lemma 2. *Let $\Sigma \cup \{\varphi\}$ be a finite set of Max-Keys. If q'_φ is reachable from q_φ in the witness network $G_{\Sigma,\varphi}$, then $\varphi \in \Sigma^+_{\mathfrak{R}}$.*

Proof (Sketch). Let D denote the simple path in $G_{\Sigma,\varphi}$ from q_φ to q'_φ. According to the definition of the witness network we can assume without loss of generality that D consists of a sequence $\pi_1, \ldots, \pi_{n+1}$, $n \geq 1$, where for each $i = 1, \ldots, n$, π_i starts with a possibly empty sequence of upward edges followed by a single witness edge $(w_{\sigma_i}, w'_{\sigma_i})$ where w_{σ_i} and w'_{σ_i} witness the applicability of σ_i to φ, and π_{n+1} is a possibly empty sequence of upward edges. Moreover, we can assume that $q_\varphi, w'_{\sigma_1}, \ldots, w'_{\sigma_n}$ form a proper descendant chain, q'_φ is a proper descendant of $w'_{\sigma_{n-1}}$ and w'_{σ_n} is a descendant node of q'_φ in $T_{\Sigma,\varphi}$. This situation is illustrated by the witness network $G_{\Sigma,\varphi}$ in Figure 3.

We now describe a series of assumptions which we use to show that the existence of the witness edges in D implies the existence of a witness edge (q_φ, q'_φ) which results from a Max-Key σ in Σ^+.

1. The final witness edge in D can be replaced by a witness edge that ends in q'_φ. That is, we can assume without loss of generality that π_{n+1} is indeed an empty sequence and $w'_{\sigma_n} = q'_\varphi$ where the set of key paths of σ_n is $\{P_1^\varphi, \ldots, P_{k_\varphi}^\varphi\}$.
2. If there is a witness edge (w_σ, w'_σ) in the witness network $G_{\Sigma,\varphi}$ that corresponds to the applicability of some $\sigma \in \Sigma^+_{\mathfrak{R}}$ to φ, then for each node w between w_σ and w'_σ in $T_{\Sigma,\varphi}$ there is also a witness edge (w, w'_σ) in $G_{\Sigma,\varphi}$ which corresponds to the applicability of some $\sigma' \in \Sigma^+_{\mathfrak{R}}$ to φ. In our example in Figure 3, this indicates the existence of a witness edge from the first ℓ_0 node (from top to bottom) to the *client* node.
3. If there is a witness edge $(w_{\sigma_1}, w'_{\sigma_1})$ and another witness edge $(w'_{\sigma_1}, q'_\varphi)$, then there is also a witness edge $(w_{\sigma_1}, q'_\varphi)$. In our example in Figure 3, this indicates the existence of a witness edge from the *db* node to the *account* node.

The strategy to prove the previous assumptions consists on showing that for each new witness edge D' obtained by virtue of the assumptions, there is a corresponding inference by $\mathfrak{R}$ of a Max-Key σ' that corresponds to the witness edge D'. For the sake of presentation, we omit this lengthy but not very difficult part of the proof.

From Assumption 1 and 2, we can conclude that there is a simple path D' in $G_{\Sigma,\varphi}$ from q_φ to q'_φ. In fact, D' consists of the sequence $\pi'_1, \ldots, \pi'_n$ where each π'_i with $1 \leq i \leq n$ consists of a single witness edge $(w_{\sigma_i}, w'_{\sigma_i})$ where $w'_{\sigma_i} = w_{\sigma_{i+1}}$ for $i = 1, \ldots, n-1$ and where $w_{\sigma_1} = q_\varphi$ and $w'_{\sigma_n} = q'_\varphi$. Again, $q_\varphi, w'_{\sigma_1}, \ldots, w'_{\sigma_n}$ form a proper descendant chain.

At this stage we can use Assumption 3 repeatedly to conclude that there is a single witness edge $D_0 = (q_\varphi, q'_\varphi)$ in $G_{\Sigma,\varphi}$ resulting from the Max-Key $\sigma = (Q_\sigma, (Q'_\sigma, \{P_1^\varphi, \ldots, P_{k_\varphi}^\varphi\}))$ in $\Sigma^+_{\mathfrak{R}}$ that is applicable to φ. Due to the applicability of σ to φ we conclude that $Q_\varphi \subseteq Q_\sigma$ and $Q'_\varphi \subseteq Q'_\sigma$. We can now apply the *context-path-containment* and *target-path-containment* rule to obtain $(Q_\varphi, (Q'_\varphi, \{P_1^\varphi, \ldots, P_{k_\varphi}^\varphi\})) \in \Sigma^+_{\mathfrak{R}}$, which proves the lemma. □

The next example illustrates how the edges of the witness network encode an inference by $\mathfrak{R}$.

Example 6. Let φ and Σ be as in Example 5. Recall that the right picture of Figure 3 shows the corresponding witness network. Thus by Lemma 2, $\Sigma \vdash_{\mathfrak{R}} \varphi$. Next, we show the actual derivation by $\mathfrak{R}$. We first apply the *superkey* rule to σ_3 to derive $\sigma'_3 = (_^*.client, (account, \{no, kind\}))$ (cf. with Assumption 1 in the proof of Lemma 2). We also apply the *superkey* rule to σ_1 and σ_2 to derive $\sigma'_1 = (\varepsilon, (bank._, \{_^*.account.no, _^*.account.kind\}))$ and $\sigma'_2 = (bank, (_^*.client, \{_.no, _.kind\}))$, respectively. Then we apply the *target-to-context* rule to σ'_2 to derive $\sigma''_2 = (bank._^*, (client, \{_.no, _.kind\}))$ (cf. with Assumption 2 in the proof

of Lemma 2). It is easy to see that by successively applying the *context-path-containment*, *target-path-containment* and *key-path-containment* rules, we can derive from σ_1', σ_2'' and σ_3' the following Max-Keys, respectively.

$$\alpha_1 = (\varepsilon, (bank._^*, \{client.account.no, client.account.kind\}))$$
$$\alpha_2 = (bank._^*, (client, \{account.no, account.kind\}))$$
$$\alpha_3 = (bank._^*.client, (account, \{no, kind\}))$$

Finally, by application of the *interaction* rule to α_2 and α_3 we obtain $\alpha_4 = (bank._^*, (client.account, \{no, kind\}))$, and again by application of the *interaction* rule to α_1 and α_4 we derive φ (cf. with Assumption 3 in the proof of Lemma 2).

5.3 Completeness

We have now the tools to prove the completeness of our set of inference rules.

Theorem 1. *The inference rules in Table 1 are complete for the implication of Max-Keys.*

Proof (Sketch). Let $\Sigma \cup \{\varphi\}$ be a finite set of Max-Keys such that $\varphi \notin \Sigma^+_{\mathfrak{R}}$. We will show that $\varphi \notin \Sigma^*$ by constructing a finite XML tree T which satisfies all Max-Keys in Σ but does not satisfy φ. Since $\varphi \notin \Sigma^+_{\mathfrak{R}}$ we know by Lemma 2 that there is no path from q_φ to q'_φ in the witness network $G_{\Sigma,\varphi}$. Let u denote the bottom-most descendant node of q_φ in $T_{\Sigma,\varphi}$ that is reachable from q_φ in $G_{\Sigma,\varphi}$. Note that u is a proper ancestor of q'_φ in $T_{\Sigma,\varphi}$ since otherwise u and thus q'_φ were reachable from q_φ in $G_{\Sigma,\varphi}$.

Let T_0 denote a copy of the path from r to u in $T_{\Sigma,\varphi}$, and T_1, T_2 denote two node-disjoint copies of the subtree of $T_{\Sigma,\varphi}$ rooted at u. We want that a node of T_1 and a node of T_2 become value equal precisely when they are copies of the same marked node in $T_{\Sigma,\varphi}$. For attribute and text nodes this is achieved by choosing string values accordingly, while for element nodes we can adjoin a new child node with a label from $\mathcal{L} - (\mathcal{L}_{\Sigma,\varphi} \cup \{\ell_0\})$ to achieve this. The counter-example tree T is obtained from T_0, T_1, T_2 by identifying the leaf node u of T_0 with the root nodes of T_1 and T_2. We conclude that T violates φ, and our construction guarantees that T satisfies all $\sigma \in \Sigma$. □

The construction of the counter example tree T in the proof of Theorem 1 is illustrated by the following example.

Example 7. Let σ_1, σ_2 and φ be as in Example 5. The corresponding mini-tree $T_{\Sigma,\varphi}$, witness network $G_{\Sigma,\varphi}$ and counter-example tree T for the implication of φ by $\Sigma = \{\sigma_1, \sigma_2\}$ are shown in Figure 4. Note that T satisfies σ_1 and σ_2, and violates φ.

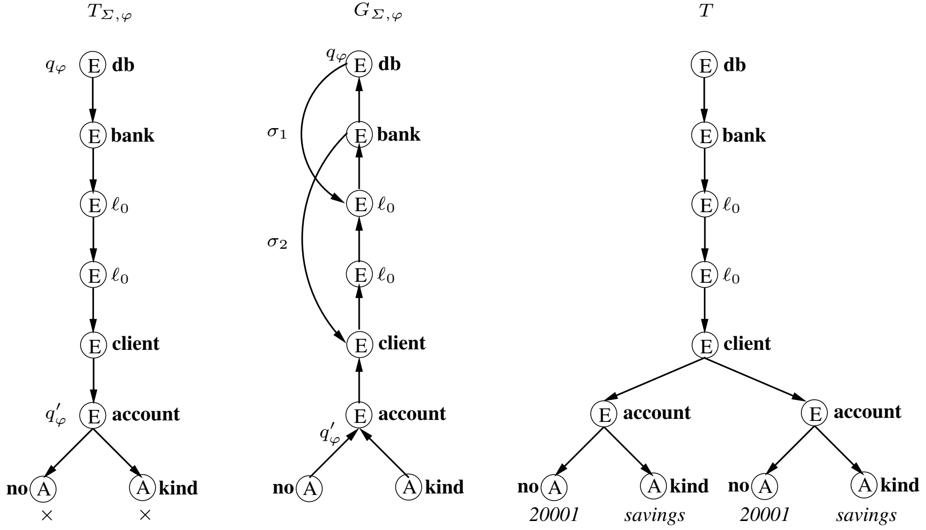


Fig. 4. Mini-tree, witness network and counter-example tree

5.4 Dealing with Structural Max-Keys

Max-Keys with empty sets of key paths behave quite differently than Max-Keys with non empty key paths. The former ones identify nodes (within certain subtrees) by a unique context path. The latter ones identify nodes (within subtrees) by unique data values.

Let us consider Max-Keys of the form $\alpha = (Q, (Q', \emptyset))$ whose key path Q' may contain single-label wildcards, but no variable-length wildcards. It is not hard to see that the inference rules in Table 1 still hold for this case. Using the *superkey* rule we can derive $\alpha' = (Q, (Q', \{_^*\}))$ from α.

Thus when constructing the witness network $G_{\Sigma,\varphi}$ for some Σ and φ in Max-Keys such that α belongs to Σ then we will replace α by α'. As we have demonstrated above the counter-example tree T constructed for φ and the resultant Σ' would satisfy Σ' but violate φ. However, it is easy to validate that by our construction T satisfies not only α' but even α. So T would actually satisfy Σ but violate φ, thus showing that Σ does not imply φ.

If φ has an empty set of key paths, then we will construct the witness network $G_{\Sigma,\varphi'}$ where $\varphi' = (Q_\varphi, (Q'_\varphi, \{_^*\})))$. As we have demonstrated above the counter-example tree T constructed for φ' and Σ would satisfy Σ but violate φ'. Consequently, it would also violate φ (as φ implies φ'), thus showing that Σ does not imply φ.

Finally, note that $\varphi' = (Q_\varphi, (Q'_\varphi, \{_^*\})))$ does not generally imply φ. A counter-example tree T can be easily constructed from merging two copies of $T_{\Sigma,\varphi'}$ on all nodes other than $q'_{\varphi'}$ and then assigning two different string values to the two copies of $q'_{\varphi'}$. Then T would satisfy φ' but violate φ.

6 A Decision Algorithm for Max-Keys Implication

We will now design a low-degree polynomial time decision algorithm for the implication problem of Max-Keys. It is based on the following characterization of the implication problem in terms of the reachability problem of nodes in the witness network.

Theorem 2. *Let $\Sigma \cup \{\varphi\}$ be a finite set of Max-Keys. We have $\Sigma \models \varphi$ if and only if q'_φ is reachable from q_φ in the witness network $G_{\Sigma,\varphi}$.*

Proof. Assume that q'_φ is not reachable from q_φ in $G_{\Sigma,\varphi}$. Then we can generate a counter-example tree for the implication of φ by Σ as described in the proof of Theorem 1. It follows that Σ does not imply φ.

On the contrary, assume now that q'_φ is indeed reachable from q_φ in $G_{\Sigma,\varphi}$. We conclude by Lemma 2 that φ is implied by Σ. □

Theorem 2 tells us that we can decide implication by constructing the witness network and testing reachability in $G_{\Sigma,\varphi}$ by applying well known search techniques. This establishes a surprisingly compact algorithm.

Algorithm 1. Decision Algorithm for the Implication Problem of Max-Keys

Input: finite set of Max-Keys $\Sigma \cup \{\varphi\}$
Output: yes, if $\Sigma \models \varphi$; no, otherwise
1: Construct $G_{\Sigma,\varphi}$ for Σ and φ;
2: **if** q'_φ is reachable from q_φ in $G_{\Sigma,\varphi}$ **then return** yes;
else return no; **end if**

The presentation of Algorithm 1 to decide the implication of Max-Keys remains the same as Algorithm 4.2 in [15] to decide the implication of the strictly less expressive fragment $\mathcal{K}_1$ of XML keys. However, the construction of the witness network $G_{\Sigma,\varphi}$, which is central to both algorithms, requires considerably more effort for the more expressive fragment of Max-Keys. This effort results in an slight increase in the worst-case time complexity of the algorithm. Nevertheless, the simplicity of Algorithm 1 enables us to conclude that the implication of Max-Keys can be decided in low-degree polynomial time in the worst case.

6.1 Implementation and Complexity Analysis of Algorithm 1

In this subsection we discuss our implementation of Algorithm 1 and analyze its theoretical complexity.

Data Structures. We need data structures suitable to represent mini-trees and witness networks. The obvious candidates are adjacency matrices and adjacency lists [2]. 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 networks are represented likewise.

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. Reverse all edges of $T_{\Sigma,\varphi}$;
iv. 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 $\mathcal{O}(|\varphi| \times l)$, where $|\varphi|$ is the sum of the lengths of all path expressions in φ and l is the maximum number of consecutive single-label wildcards that occurs in Σ. Note that the mini-tree $T_{\Sigma,\varphi}$ has at most $(|\varphi| \times l) + 1$ 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 $\mathcal{O}(|\varphi| \times l)$ time.

Assuming that the adjacency list of a vertex v_i lists the vertices v_j such that there is an edge from v_i to v_j. Substep (iii) involves the generation of a new adjacency list which corresponds to $T_{\Sigma,\sigma}$ with all its directed edges reverted. Again, this takes time $\mathcal{O}(|\varphi| \times l)$.

Substep (iv) 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 $w_\sigma \in [\![Q_\sigma]\!]$. Note that a query of the form $v[\![Q]\!]$ is a Core XPath query and can be evaluated on a node-labeled three T in order $\mathcal{O}(|T| \times |Q|)$ time [13]. Hence, we can evaluate $w'_\sigma[\![P_i^\sigma]\!]$ for all $i = 1, \ldots, k_\sigma$ in time $\mathcal{O}(|\varphi| \times l \times |\sigma|)$. Since $[\![Q_\sigma]\!]$ and $w_\sigma[\![Q'_\sigma]\!]$ contain at most $|\varphi| \times l$ nodes each,

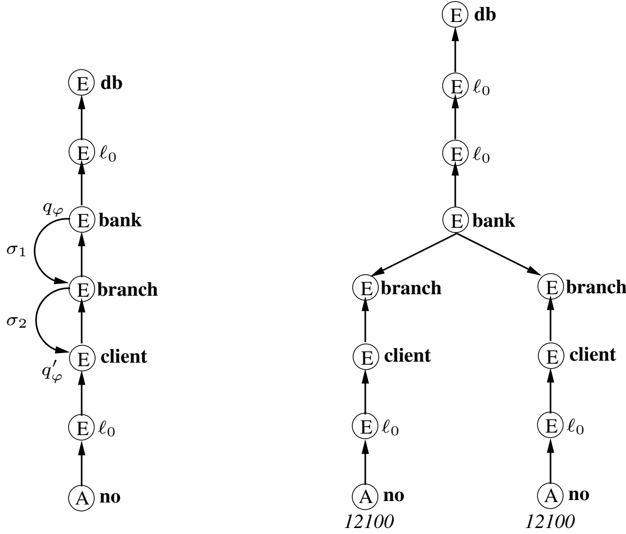


Fig. 5. Incorrectly built witness network and counter-example

this step can be executed in $\mathcal{O}((|\varphi| \times l)^3 \times |\sigma|)$ time for each σ. Hence, we need $\mathcal{O}(||\Sigma|| \times (|\varphi| \times l)^3)$ time to generate $G_{\Sigma,\varphi}$, where $||\Sigma||$ denotes the sum of all sizes $|\sigma|$ for $\sigma \in \Sigma$.

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 $G_{\Sigma,\varphi}$ [17]. Thus reachability can be decided in $\mathcal{O}(|\varphi + l|^2)$.

Complexity. The following result is a consequence of the previous observations.

Theorem 3. *If $\Sigma \cup \{\varphi\}$ is a finite set of Max-Keys, then the implication problem $\Sigma \models \varphi$ can be decided in $\mathcal{O}(||\Sigma|| \times (|\varphi| \times l)^3)$ time, where $|\varphi|$ is the sum of the lengths of all path expressions in φ, $||\Sigma||$ denotes the sum of all sizes $|\sigma|$ for $\sigma \in \Sigma$, l is the maximum number of consecutive single-label wildcards that occur in Σ.*

It is important to note the blow-up in the size of the counter-example with respect to φ. This is due to the occurrence of consecutive single-label wildcards. If the number l is fixed in advance, then Algorithm 1 establishes a worst-case time complexity that is quadratic in the input. In particular, if the input consists of XML keys in $\mathcal{K}_1$, as studied in [15,12], then the worst-case time complexity of Algorithm 1 is that of the algorithm dedicated to XML keys in $\mathcal{K}_1$ only [15].

Remark 1. If we simply replace each variable-length wildcard "_*" by the single-label ℓ_0 and not by a sequence of $l + 1$ labels ℓ_0, then Theorem 2 does not hold. To see this, consider

$$\varphi = (_^*.bank, (branch.client, \{_no\}))$$
$$\sigma_1 = (_.bank, (branch, \{client._no\}))$$
$$\sigma_2 = (_^*.bank.branch, (client, \{_no\}))$$

Let $\Sigma = \{\sigma_1, \sigma_2\}$. A simple replacement of "$_^*$" by ℓ_0 results in the witness network shown on the first (from the left) picture in Figure 5. But then by Lemma 2, Σ would imply φ, which is clearly incorrect as shown by the counter-example tree on the second picture in Figure 5.

7 Applications and Performance Evaluation

In this section we present the results regarding the application and performance evaluation of our decision algorithm for the finite implication problem of Max-Keys. We also compare these results against the results obtained in [12] for a strictly less expressive fragment of XML keys. We start by describing in Subsection 7.1 the construction of the sets of XML keys used in our experiments. These set of keys correspond to actual XML documents which are also described in this section, and latter used in the experiments regarding the application of XML key reasoning in the context of XML document validation. These latter experiments are reported in Subsection 7.3. The experiments regarding the performance of the decision algorithm for the implication of Max-Keys are presented in Subsection 7.2.

The running times reported in Subsection 7.2 were obtained in a fairly standard Intel Core i7 2.8 GHz machine, with 4 GB of RAM, running a Linux kernel 2.6.32. We compiled our C++ implementation of the algorithms using the standard g++ compiler from the GNU Compiler Collection 4.6.3. The experiments in Subsection 7.3 were run on a cluster of 160 nodes of type Xeon E5-2680(i7) with 128 GB and 16 cores per node (2.68 GHZ) running Linux RHEL 6.1. The reason for using this facility was to save time on the experiments concerning the validation of XML documents, in particular this was required for the validation of the big XML document of 3.2GB corresponding to the Chilean electoral role. The use of these nodes allowed us to load in RAM the whole DOM representation of each XML tree tested in the experiments (including the one corresponding to the Chilean electoral role), thus simplifying the task of writing a validation algorithm to test our ideas. It should be noted that this task is accessory to this paper as it was only used to showcase the benefits of using XML key reasoning in this context. In this work, we are not concerned with parallelization challenges. In fact, we ran these jobs with a standard sequential algorithm, having one instance of an XML document and corresponding set of keys to be validated per node. We simply run several experiments (over different documents) at once by submitting different jobs (i.e., by using a separated node for each validation instance).

We remind the reader that, as stated in the introduction, all the data sets used in our experiments as well as the full set of results and the binary codes of the implemented algorithms are publicly available at `http://emir-munoz.github.com/xml-constraints`. To the best of our knowledge, this is the first

time that large sets of non-trivial XML keys for real XML documents are made publicly available. Our hope is that this contribution facilitates further research in the area.

7.1 Defining the Data Sets: XML Keys and XML Documents

We use a collection of XML documents from [22] plus a large XML document (*padron.xml*) that holds the publicly available Chilean electoral roll[1]. A characterization of these XML documents is shown in Table 2. From [22] we use the documents *321gone.xml* and *yahoo.xml* (auction data), *dblp.xml* (bibliographic information on CS), *nasa.xml* (astronomical data), *SigmodRecord.xml* (articles from SIGMOD Record), and *mondial-3.0.xml* (world geographic database).

Table 2. XML Documents

Doc ID	Document	No. of Elements	No. of Attributes	Size	Max. Depth	Average Depth
doc1	padron.xml	119,235,504	39,745,398	3.2 GB	5	4.06667
doc2	321gone.xml	311	0	24 KB	5	3.76527
doc3	yahoo.xml	342	0	25 KB	5	3.76608
doc4	dblp.xml	29,494	3,247	133.9 MB	6	2.90228
doc5	nasa.xml	476,646	56,317	25 MB	8	5.58314
doc6	SigmodRecord.xml	11,526	3,737	478 KB	6	5.14107
doc7	mondial-3.0.xml	22,423	47,423	1.9 MB	5	3.59274

In order to generate realistic XML keys for the experiments regarding the performance of the decision algorithm in Subsection 7.2 as well as to use the same sets of XML keys for the experiments involving documents validation in Subsection 7.3, we follow a strategy that is grounded in the XML documents described above. We explain this strategy using the XML document with the Chilean electoral roll as a model. The layout of this document is illustrated in Figure 6. For clarity of presentation we have translated the labels of the nodes from Spanish to English and simplified the structure by collapsing the first name, father's surname and mother's surname descendant nodes of the *name* element node into just one text node. We start by defining a series of keys which are appropriate in the context of this XML document. We list some of them together with an informal interpretation for reference.

a. $(\varepsilon, (commune, \{person\}))$
   "A person cannot be enrolled in more than one commune".
b. $(\varepsilon, (commune, \{person.id\}))$
   "Similar to (a), but not equivalent".

[1] The Chilean electoral roll can be downloaded in PDF format from the official site `http://www.servel.cl` of the Electoral Commission of Chile. The XML version used in this paper was extracted from the PDF document by Cristian Bravo-Lillo and can be obtained from `http://manzanamecanica.org/2012/11/padron_electoral_en_xml.html`

c. $(commune, (person, \{id\}))$
"A *person* node can be identified by its *id* attribute respectively to a *commune* node".
d. $(commune.person, (polling, \{circumscription, district\}))$
"A *polling* node can be identified by its child nodes *circumscription* and *district* node respectively to a *person* node".
e. $(commune.person, (polling, \emptyset))$
"A *person* node can only have one child node with label *polling*".
f. $(commune.person.polling, (circumscription, \emptyset))$
"A *polling* node can only have one child node with label *circumscription*".
g. $(_._._, (circumscription, \emptyset))$
"Idem (f) over trees with the layout of the tree in Figure 6".
h. $(\varepsilon, (_, \{_^*.id\}))$
"Idem (b) over trees with the layout of the tree in Figure 6".
i. $(_._, (_^*.polling, \{_.S\}))$
"A *polling* node can be identified by the value of its descendant text nodes relatively to the nodes at level two in the XML tree".
j. $(_._, (_, \{_^*.S\}))$ "Every node at level three can be identified by its descendant text nodes respectively to a node at level two. In trees with the layout of the tree in Figure 6, this means that for instance the name of a person cannot coincide with the address which in turn cannot coincide with the circumscription nor with the district".

Note that the XML keys in (a)–(d) are in the strictly less expressive fragment $\mathcal{K}_1$ of XML keys studied in [12] while the remaining keys are not. All the XML keys in the previous list are however in the fragment of Max-Keys covered in this work. Also note that the XML keys in (e)–(g) are structural keys and that, over trees with the layout of the tree in Figure 6, the XML keys in (i) and (j) cannot be expressed without allowing wildcards in the key paths.

We then define new (implied) XML keys, by successively applying the inference rules for Max-Keys from Table 1 to the previously defined set of XML keys.

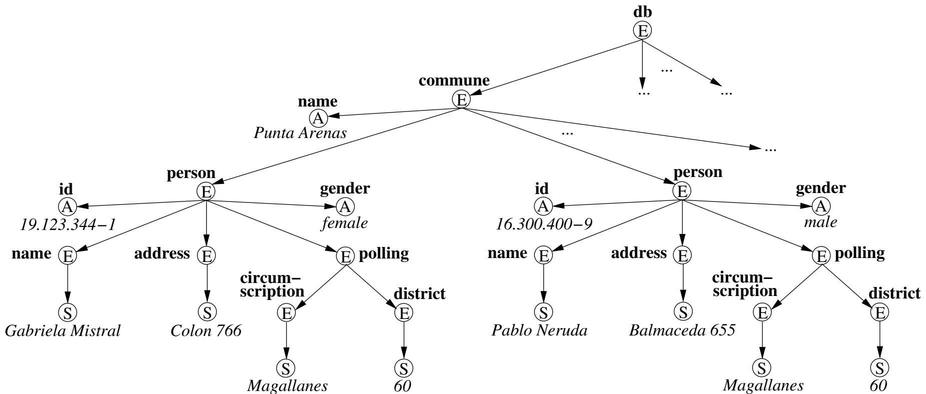


Fig. 6. Chilean electoral roll

For instance, by applying the *superkey* rule to the XML key in (h) we can obtain the new XML keys $(\varepsilon, (_, \{_^*.id, _^*.gender\}))$, $(e, (_, \{_^*.id, _^*.name\}))$ and $(e, (_, \{_^*.id, _^*.gender, _^*.name\}))$ among others.

Finally, we define sets $\Sigma \cup \{\varphi\}$ of Max-Keys such that the keys in Σ are applicable to φ and $\Sigma \not\models \varphi$. The idea is to test the algorithm also against sets of Max-Keys $\Sigma \cup \{\varphi\}$ for which it is not only the case that φ is not implied by Σ, but also it is non trivial for the algorithm to determine this fact. The following steps allow us to obtain sets of Max-Keys with these characteristics: a) define a Max-Key φ, b) build its corresponding witness network $G_{\emptyset,\varphi}$, c) add several witness edges to $G_{\emptyset,\varphi}$ taking care of keeping q'_φ not reachable from q_φ, and d) define Max-Keys corresponding to those witness edges. As an example, let us take $\varphi = (\varepsilon, (commune.person.polling, \{district\}))$. The corresponding witness network $G_{\emptyset,\varphi}$ is shown in the first picture of Figure 7. From the witness edges A1, A2 and A3 shown in the second picture of Figure 7, we can derive the following set Σ of Max-Keys among others:

$$\begin{aligned}
\sigma_1 &= (commune, (person, \{polling.district\})) \\
\sigma_2 &= (_, (_, \{_^*.district\})) \\
\sigma_3 &= (_^*.person, (polling, \{district\})) \\
\sigma_4 &= (_^*.person, (_, \{_^*.district\})) \\
\sigma_5 &= (_, (_^*.polling, \{_\})) \\
\sigma_6 &= (_, (_._, \{district\}))
\end{aligned}$$

where the keys σ_1 and σ_2 correspond to the witness edge A1, the keys σ_3 and σ_4 correspond to the witness edge A2, and the keys σ_5 and σ_6 correspond to the witness edge A3.

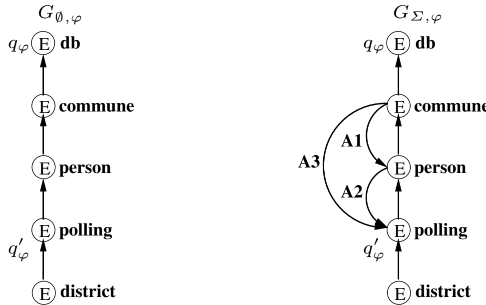


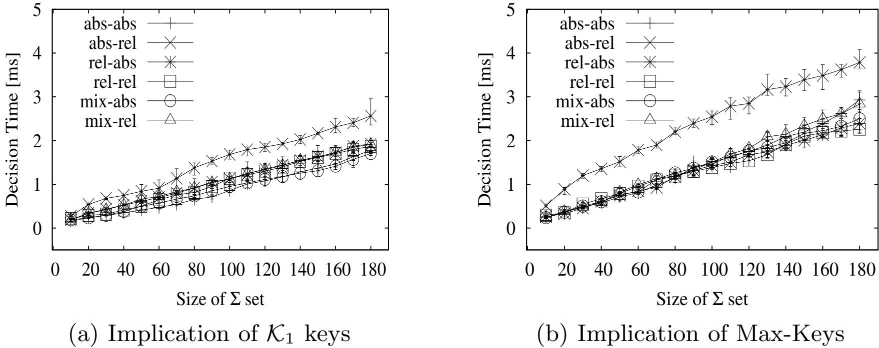
Fig. 7. Witness networks for a non-implied key

The processes described in this section were used to produce a robust collection of Max-Keys as well as a collection of $\mathcal{K}_1$ keys to thoroughly test the performance of the decision algorithm for the implication problem of Max-Key and for the implication problem of $\mathcal{K}_1$ keys, respectively. The results are reported in the next section.

7.2 Performance of the Decision Algorithm for Max-Keys

In order to have a baseline to determine how much the increase in expressibility of the considered class of Max-Keys affects the performance of the decision algorithm, we also include measures of the performance of the decision algorithm in [12] which is optimized for deciding implication of the strictly less expressive fragment $\mathcal{K}_1$ of XML keys from [15].

The results regarding running times for deciding the implication of $\mathcal{K}_1$ keys and Max-Keys are shown in Figures 8(a) and 8(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 reported running time corresponds to $\left(\sum_{\varphi_i \in \Phi} time(\Sigma, \varphi_i)\right)/|\Phi|$. In our experiments the sets Φ were composed of 10 fixed XML keys. We tested the scalability of the algorithms by adding, in each iteration, 10 new XML keys to the corresponding Σ sets. Each of the experiments was executed 5 times. The resulting error bars are included in the graphs. They are consistent with time variations commonly produced by the scheduling of the operating system and the use of the $time()$ function to measure the experiments [21].



(a) Implication of $\mathcal{K}_1$ keys (b) Implication of Max-Keys

Fig. 8. Performance of the Decision Algorithms for the Implication of XML Keys

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 performance shown by the "abs-rel" curves 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 networks are defined.

Finally, we present in Figure 9 the aggregate results. For a small set Σ with about 10 XML keys, the execution takes 0.2ms in average, whereas for a large set of about 180 XML keys, the execution takes around 4ms in the worst case. Thus, we can conclude that both decision algorithms are efficient in practice and both scale well. In particular, the price to pay for the added expressibility provided by the Max-Keys is in the order of just 3ms for a considerable big set of 180 Max-Keys.

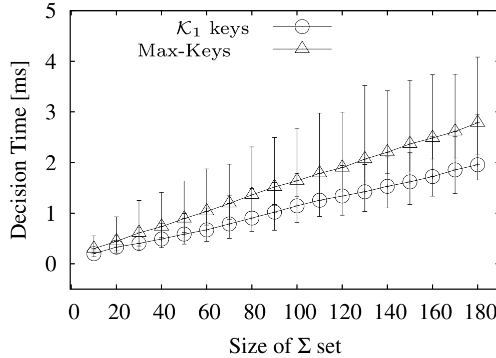


Fig. 9. Aggregate results for the Performance of the Decision Algorithms

7.3 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 [5]. In this section we propose to use our decision algorithm to compute non-redundant cover sets of Max-Keys. As shown in the experiments this has the potential to significantly reduce the time needed to validate XML documents against sets of Max-Keys. In our experiments we validate XML documents as a whole, starting from scratch. This is clearly useful for instance for tasks such as data cleaning, and it is enough to showcase a concrete example in which our algorithm for the implication of XML keys can be sucessfully used in practice. Nevertheless we note that the central case of incremental validation, which is left out of the scope of this paper, can also benefit from this idea.

Cover Sets for XML Keys. We define the concept of cover set of XML keys following the notion given in [19] for functional dependencies in the relational model. Thus, two sets Σ_1 and Σ_2 of XML keys are a *cover* of one another if they imply exactly the same set of XML keys, i.e., if $\Sigma_1^* = \Sigma_2^*$. A set Σ_c of XML keys is a *non-redundant cover* if none of its proper subsets is a cover for it.

This is the case if there is no key φ in Σ_c such that $\Sigma_c - \{\varphi\} \models \varphi$. Note that a non-redundant cover set Σ_c of a given set Σ of XML keys has potentially fewer keys than Σ and at the same time, every tree T that satisfies all the keys in Σ_c, also satisfies all the keys in the original set Σ.

Thus, given a set Σ of Max-Keys, we can use our decision algorithm to compute a non-redundant cover set Σ_c for it and then, provided it has fewer keys than the original set Σ, validate the target XML document against Σ_c instead of Σ. As shown in our experiments, this can potentially result in enormous time savings.

It is important to note that a set Σ can contain more than one non-redundant cover set and there can also exist non-redundant cover sets that are not included in Σ. In this work we only consider cover sets that are included in Σ.

Validation against Non-redundant Covers. The aim is to determine the viability of computing non-redundant cover sets to reduce the overall time required to validate XML documents against sets of Max-Keys.

Validating an XML document against a set of XML keys involves checking, for every XML key in the set, whether the document satisfies such key. For this task, we use a semi-naïve 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 not use sophisticated validation algorithms such as [9,18] because they are not suitable in its current form to validate Max-Keys. An adaptation of such kind of algorithms for Max-Keys is a complex task which is out of the scope of this work and non essential to prove our point, that is to prove that reasoning about XML keys brings important benefits in this context. Furthermore, the proposed optimization based on non-redundant cover sets is independent of the particular algorithm used for XML key validation.

Tests Results. Given a set Σ of Max-Keys, we compute a non-redundant cover set Σ_c by simply checking for every key $\varphi_i \in \Sigma$, whether $\Sigma - \{\varphi_i\} \models \varphi_i$ (this step is done by our decision algorithm for the implication problem of Max-Keys). If we find a key φ_i that is implied by $\Sigma - \{\varphi_i\}$, we eliminate it from Σ and continue checking the remaining keys against the resulting set $\Sigma - \{\varphi_i\}$. For comparison purpose, we also compute the set $\mathcal{C}$ of all non redundant covers that are strictly contained in Σ. To obtain $\mathcal{C}$, we first check every singleton subset of Σ, then we check every subset of Σ of cardinality two, and so on till we have checked every strict subset of Σ. Note that once we find a non-redundant cover set $\Sigma_c \subset \Sigma$, we can automatically discard every $S_i \subset \Sigma$ such that $\Sigma_c \subseteq S_i$.

The results obtained from the computation of non-redundant cover sets are summarized in Table 3, were $time(\Sigma_c)$ denotes the time in milliseconds required to compute a non-redundant cover $\Sigma_c \subset \Sigma$, $time(\mathcal{C})$ denotes the time in milliseconds needed to compute the set $\mathcal{C}$ of all non-redundant covers contained in Σ, and $min(\Sigma_c)$ and $max(\Sigma_c)$ denote the cardinality of the smallest non-redundant

Table 3. Computation of Non-redundant Covers

XML document	$\|\Sigma\|$	$\|\Sigma_c\|$	$time(\Sigma_c)$	$\|\mathcal{C}\|$	$time(\mathcal{C})$	$min(\Sigma_c)$	$max(\Sigma_c)$
padron							
doc1	10	5	4.041	8	2896.630	5	5
321gone & yahoo							
doc2 & doc3	13	8	5.388	1	22792.900	8	8
DBLP							
doc4	12	7	5.879	1	9424.210	7	7
nasa							
doc5	12	7	5.469	2	10417.200	7	7
SigmodRecord							
doc6	12	6	4.783	2	10633.400	6	6
mondial							
doc7	12	6	3.078	12	9460.690	6	7

cover and the cardinality of the biggest non-redundant cover contained in Σ, respectively.

Figure 10 shows the optimization achieved by pre-calculating non-redundant cover sets during the validation process of the XML documents in Table 2. In both plots the x-axis represent the documents and the y-axis represent the running time in milliseconds. We use a separate plot for doc1 due to the difference in scale, and consequently in validation time, with respect to the others XML documents. Also due to differences in scale, we had to truncate the bars representing the validation time for doc4, doc5 and doc7 against the full set of XML keys. The actual time required for those validations is indicated on top of the corresponding bars.

This results clearly 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. It also confirms that the proposed optimization of the validation process, based on the pre-calculation of non-redundant covers, can significantly reduce the time required for this task.

Significant time savings are achieved in the validation of doc1, doc4, doc5 and doc7. This coincides with the largest tested documents. For instance, while it takes 38 minutes to validate the XML document *padron.xml* (doc1) against 10 Max-Keys, the validation against the best cover in $\mathcal{C}$ only requires 20 minutes.

For reference, we list next the set Σ of Max-Keys used for the validation experiments with the document *padron.xml*. Note that the nodes *given*, *father* and *mother*, which are not included in the simplified tree in Figure 6, refer to the child nodes of the *name* node in the actual structure of the XML document padron.xml. The labels *father* and *mother* refer to the father's surname and the mother's surname, respectively. The keys used in the validation experiments with the others XML documents tested in this work, can be found online as

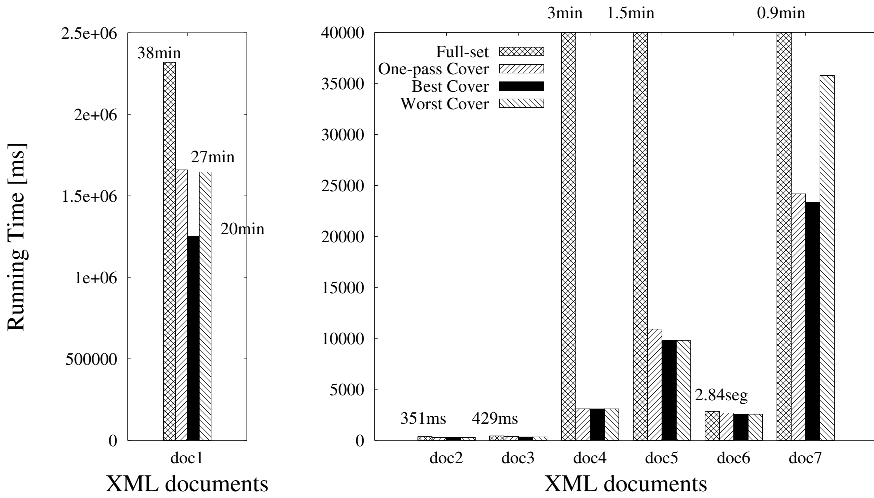


Fig. 10. Validation of XML documents

indicated at the beginning of this section. For the sake of presentation, we omit them here.

1. $(\varepsilon, (commune, \{name.given, name.father, name.mother\}))$
2. $(\varepsilon, (commune, \{_^*, person\}))$
3. $(\varepsilon, (commune, \{name.given, name.father, name.mother, person\}))$
4. $(\varepsilon, (_, \{person\}))$
5. $(\varepsilon, (commune, \{person.id\}))$
6. $(\varepsilon, (commune, \{person.id, person.gender\}))$
7. $(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
8. $(\varepsilon, (commune, \{person.id, person.gender, person.name.given, person.name.father, person.name.mother\}))$
9. $(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$
10. $(\varepsilon, (commune, \{person.id, person.gender, person.name.given, person.name.father, person.name.mother, person.address\}))$

The result of computing all non-redundant covers included in Σ is a collection $\mathcal{C}$ with 8 sets, shown in Table 4. The last set contains simpler keys (i.e., with lower length and fewer single-label and variable-length wildcards), which results in less complex XPath queries and the minimum time required for validation. Specifically, the more complex Max-Keys like $(\varepsilon, (comuna, \{_^*, person\}))$ involve 345 *commune* nodes, that in total compress 13, 248, 351 *person* (complex) nodes with various element and attribute children. Moreover, if we discard the absolute Max-Key with a variable-length wildcard in its target path (which involves 3, 239, 791 nodes), we can reduce the validation time to six minutes. Indeed, when more nodes are selected by the target path and more complex nodes by the key paths, then it becomes more time consuming to compute the corresponding value intersection among all pairs of elements.

Table 4. Non-redundant covers used in the validation experiments with *padron.xml*

<table>
<tr><td>$(\varepsilon, (commune, \{name.given, name.father, name.mother, person\}))$
$(\varepsilon, (_, \{person\}))$
$(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
$(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$
$(\varepsilon, (commune, \{person.id, person.gender, person.name.given, person.name.father, person.name.mother, person.address\}))$</td></tr>
<tr><td>$(\varepsilon, (commune, \{name.given, name.father, name.mother, person\}))$
$(\varepsilon, (_, \{person\}))$
$(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
$(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$
$(\varepsilon, (commune, \{person.id, person.gender, person.name.given, person.name.father, person.name.mother\}))$</td></tr>
<tr><td>$(\varepsilon, (commune, \{name.given, name.father, name.mother, person\}))$
$(\varepsilon, (_, \{person\}))$
$(\varepsilon, (commune, \{person.id, person.gender\}))$
$(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
$(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$</td></tr>
<tr><td>$(\varepsilon, (commune, \{name.given, name.father, name.mother, person\}))$
$(\varepsilon, (_, \{person\}))$
$(\varepsilon, (commune, \{person.id\}))$
$(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
$(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$</td></tr>
<tr><td>$(\varepsilon, (commune, \{name.given, name.father, name.mother\}))$
$(\varepsilon, (_, \{person\}))$
$(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
$(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$
$(\varepsilon, (commune, \{person.id, person.gender, person.name.given, person.name.father, person.name.mother, person.address\}))$</td></tr>
<tr><td>$(\varepsilon, (commune, \{name.given, name.father, name.mother\}))$
$(\varepsilon, (_, \{person\}))$
$(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
$(\varepsilon, (commune, \{person.id, person.gender, person.name.given, person.name.father, person.name.mother\}))$
$(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$</td></tr>
<tr><td>$(\varepsilon, (commune, \{name.given, name.father, name.mother\}))$
$(\varepsilon, (_, \{person\}))$
$(\varepsilon, (commune, \{person.id, person.gender\}))$
$(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
$(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$</td></tr>
<tr><td>$(\varepsilon, (commune, \{name.given, name.father, name.mother\}))$
$(\varepsilon, (_, \{person\}))$
$(\varepsilon, (commune, \{person.id\}))$
$(\varepsilon, (commune, \{_.id, _.gender, _.name\}))$
$(\varepsilon, (_^*, \{person.id, person.gender, person.name, person.address\}))$</td></tr>
</table>

8 Conclusion

Our contribution is two-fold. Firstly, we introduced an expressive fragment of XML keys (Max-Keys) that is sufficiently flexible to advance XML data processing in important areas of XML application such as consistency management, data integration, query optimization and view maintenance. The flexibility results from the right balance between expressiveness and efficiency of maintenance. Secondly, we have shown through extensive experimentation that reasoning about this expressive fragment of XML keys can be done efficiently in practice, and scales well. Our results promote the use of XML keys to real-world XML practice, where a little more semantics makes applications a lot more effective.

Additionally, we have shown that our contribution to 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 have studied 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). 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.

The experiments with large sets of non-trivial XML keys and large XML documents provide a good platform (and also pinpoint the need) for further research in the area. This can go into various directions. XML practice might well warrant the study of other classes of XML keys that require different paradigms to select and compare nodes, or specify restrictions. It would be interesting to investigate the interaction of XML keys with schema specification languages and other classes of database constraints, including functional, multivalued and inclusion dependencies. This is likely to be a challenging task as already observed and illustrated by examples in previous work [8]: keys can non-trivially interact with content models and thus behave differently under such specifications.

It would also be interesting to explore other practical applications of the decision algorithm for the implication problem in areas such as optimization of XPath queries, XML constraint mining, and XML design. The broad area in which XML keys can be applied, as indicated in several parts of this article, warrant further studies.

Acknowledgement. This paper was funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2). The authors would like to thank the anonymous reviewers for their comments and suggestions, which have contributed to improve the paper. Finally, the authors wish to acknowledge the

contribution of the NeSI high-performance computing facilities and the staff at the Centre for eResearch at the University of Auckland. New Zealand's national facilities are provided by the New Zealand eScience Infrastructure (NeSI) and funded jointly by NeSI's collaborator institutions and through the Ministry of Business, Innovation and Employment's Infrastructure programme (http://www.nesi.org.nz).

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Aho, A., Ullman, J., Hopcroft, J.: Data structures and algorithms. Addison-Wesley (1983)
3. Apparao, V., et al.: Document object model (DOM) level 1 specification, W3C recommendation (1998), http://www.w3.org/TR/REC-DOM-Level-1/
4. 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)
5. Arenas, M., Libkin, L.: XML data exchange: Consistency and query answering. J. ACM 55, 7:1–7:72 (2008)
6. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (XML) 1.0, 4th edn., W3C recommendation (2006), http://www.w3.org/TR/xml
7. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.: Keys for XML. Computer Networks 39(5), 473–487 (2002)
8. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.: Reasoning about keys for XML. Inf. Syst. 28(8), 1037–1063 (2003)
9. 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)
10. Clark, J., DeRose, S.: XML path language (XPath) version 1.0, W3C recommendation (1999), http://www.w3.org/TR/xpath
11. Ferrarotti, F., Hartmann, S., Link, S., Wang, J.: Promoting the semantic capability of XML keys. In: Lee, M.L., Yu, J.X., Bellahsène, Z., Unland, R. (eds.) XSym 2010. LNCS, vol. 6309, pp. 144–153. Springer, Heidelberg (2010)
12. Ferrarotti, F., Hartmann, S., Link, S., Marin, M., Muñoz, E.: Performance analysis of algorithms to reason about XML keys. In: Liddle, S.W., Schewe, K.-D., Tjoa, A.M., Zhou, X. (eds.) DEXA 2012, Part I. LNCS, vol. 7446, pp. 101–115. Springer, Heidelberg (2012)
13. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. Trans. Database Syst. 30(2), 444–491 (2005)
14. Hartmann, S., Köhler, H., Link, S., Trinh, T., Wang, J.: On the notion of an XML key. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 103–112. Springer, Heidelberg (2008)
15. Hartmann, S., Link, S.: Efficient reasoning about a robust XML key fragment. ACM Trans. Database Syst. 34(2) (2009)
16. Hartmann, S., Link, S.: Expressive, yet tractable XML keys. In: EDBT 2009: 12th International Conference on Extending Database Technology. ACM International Conference Proceeding Series, vol. 360, pp. 357–367. ACM (2009)

17. Jungnickel, D.: Graphs, Networks and Algorithms. Springer (1999)
18. 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)
19. Maier, D.: Minimum Covers in the Relational Database Model. J. ACM 27, 664–674 (1980)
20. Miklau, G., Suciu, D.: Containment and equivalence for a fragment of XPath. J. ACM 51(1), 2–45 (2004)
21. Stewart, D.B., Khosla, P.K.: Mechanisms for Detecting and Handling Timing Errors. Commun. ACM 40(1), 87–93 (1997)
22. Suciu, D.: XML Data Repository, University of Washington (2002), `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`
23. Thompson, H., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1: Structures, 2nd edn., W3C Recommendation (2004), `http://www.w3.org/TR/xmlschema-1/`