Reasoning about Keys for XML

Peter Buneman[1]*, Susan Davidson[1]**, Wenfei Fan[3]***, Carmem Hara[4], and Wang-Chiew Tan[1]*

[1] University of Pennsylvania, Philadelphia, PA 19104-6389, USA
[2] Bell Laboratories, Murray Hill, NJ 07974-0636, USA
[3] Universidade Federal do Parana, Curitiba, PR 81531-990, Brazil

Abstract. We study absolute and relative keys for XML, and investigate their associated decision problems. We argue that these keys are important to many forms of hierarchically structured data including XML documents. In contrast to other proposals of keys for XML, these keys can be reasoned about efficiently. We show that the (finite) satisfiability problem for these keys is trivial, and their (finite) implication problem is finitely axiomatizable and decidable in PTIME in the size of keys.

1 Introduction

Keys are of fundamental importance in databases. They provide a means of locating a specific object within the database and of referencing an object from another object (e.g. relationships); they are also an important class of constraints on the validity of data. In particular, value-based keys (as used in relational databases) provide an invariant connection from an object in the real world to its representation in the database. This connection is crucial for modifying the database as the world that it models changes.

As XML is increasingly used to model real world data, it is natural to require a value-based method of locating an element in an XML document. Key specifications for XML have been proposed in the XML standard [22], XML Data [23], and XML Schema [26]. The authors have recently [4] proposed a key structure for XML which has the following benefits:

1. Keys are defined with path expressions and may involve attributes, subelements or more general structures. Equality is defined on tree structures instead of on simple text, referred to as *value equality*.
2. Keys, in their general form, are defined relative to a set of context nodes, referred to as *relative keys*. Such keys can be concatenated to form a hierarchical key structure, common in scientific data sets. An *absolute key* is a special case of a relative key, which has a unique context node: the root.
3. The specification of keys does not depend on any typing specification of the document (e.g. DTD or XML Schema).

* Supported by NSF IIS 99-77408 and NSF DL-2 IIS 98-17444
** Supported by NSF DBI99-75206
*** Currently on leave from Temple University. Supported in part by NSF IIS 00-93168.

```
⟨db⟩
  ⟨book  isbn=123⟩
    ⟨title⟩ HTML ⟨/title⟩
    ⟨author⟩
      ⟨name⟩
        ⟨first-name⟩ Tim ⟨/first-name⟩
        ⟨last-name⟩ Bray ⟨/last-name⟩
      ⟨/name⟩
    ⟨/author⟩
    ⟨chapter  number=1⟩ text ⟨/chapter⟩
    ⟨chapter  number=10⟩ text ⟨/chapter⟩
  ⟨/book⟩
  ⟨book  isbn=234⟩
    ⟨title⟩ XML ⟨/title⟩
    ⟨author⟩
      ⟨name⟩
        ⟨first-name⟩ Tim ⟨/first-name⟩
        ⟨last-name⟩ Bray ⟨/last-name⟩
      ⟨/name⟩
    ⟨/author⟩
    ⟨author⟩
      ⟨name⟩
        ⟨first-name⟩ Jean ⟨/first-name⟩
        ⟨last-name⟩ Paoli ⟨/last-name⟩
      ⟨/name⟩
    ⟨/author⟩
    ⟨chapter  number=1⟩ text ⟨/chapter⟩
    ⟨chapter  number=12⟩ text ⟨/chapter⟩
  ⟨/book⟩
⟨/db⟩
```

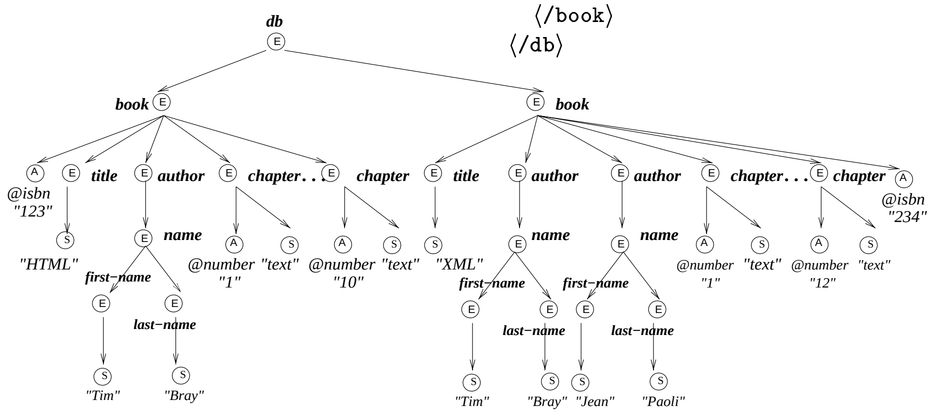

Fig. 1. Example of some XML data and its representation as a tree

In developing our notion of keys for XML, we start with a tree model of data as used in DOM [21], XSL [25, 27], XQL [19] and XML Schema [26]. An example of this representation for some XML data is shown in Fig. 1 in which nodes are annotated by their type: E for element, A for attribute, and S for string (or PCDATA). Some value-based keys for this data might include: 1) A `book` node is identified by `@isbn`; 2) An `author` node is identified by `name`, no matter where the `author` node appears; and 3) Within any subtree rooted at `book`, a `chapter` node is identified by `@number`. These keys are defined independently of any type specification. The first two are examples of absolute keys since they must hold globally throughout the tree. Observe that `name` has a complex structure. As a consequence, to test whether two authors violate this constraint involves testing value-equality on the subtrees rooted at their `name` nodes. The last one is an example of a relative key since it holds locally within each subtree rooted at a `book`. It should be noted that a chapter `@number` is not a key for the set of all `chapter` nodes in the document since two different books have chapters with `@number= 1`. It is worth remarking that proposals prior to [4] are not capable of expressing the second and third constraints.

One of the most interesting questions involving keys is that of logical implication, i.e., deciding if a new key holds given a set of existing keys. This is important for minimizing the expense of checking that a document satisfies a set of key constraints, and may also provide the basis for reasoning about how constraints can be propagated through view definitions. Thus a central task for the study of XML keys is to develop an algorithm for determining logical implication. It is also desirable to develop a sound and complete set of inference rules for generating symbolic proofs of logical implication. The existence of such inference rules, referred to as *axiomatizability*, is a stronger property than the existence of an algorithm, because the former implies the latter but not the other way around [2]. Another interesting question is whether a set of keys is "reasonable" in the sense that there exists some (finite) document that satisfies the key specification (*finite satisfiability*).

In relational databases, these decision problems for keys (and more generally, functional dependencies) have been well studied (cf. [2, 18]). The finite satisfiability problem is trivial: given any finite set of keys over a relational schema, one can always find a finite instance of the schema that satisfies the keys. Implication of relational keys is also easy, and is decidable in linear time.

For XML the story is more complicated since the hierarchical structure of data is far more complex than the 1NF structure of relational data. In some proposals keys are not even finitely satisfiable. For example, consider a key of XML Schema (in a simplified syntax): $(//*,\ [id])$, where "$//*$" (in XPath [24] syntax) traverses to any descendant of the root of an XML document tree. This key asserts that any node in an XML tree must have a unique `id` subelement (of text value) and its `id` uniquely identifies the node in the entire document. However, it is clear that no finite XML tree satisfies this key because any `id` node must have an `id` itself, and this yields an infinite chain of `id` nodes. For implication of XML keys, the analysis is even more intriguing. Keys of XML Schema are defined in terms of XPath [24], which is a powerful yet complicated language. A number of technical questions in connection with XPath are still open, including the containment of XPath expressions which is important in the interpretation of XML keys. To the best of our knowledge, the implication problem for keys defined in XML Schema is still open, as is its axiomatizability.

In contrast, we show in this paper that the keys of [4] can be reasoned about efficiently. More specifically, we show that they are finitely satisfiable and their implication is decidable in PTIME. Better still, their (finite) implication is *finitely axiomatizable*, i.e., there is a finite set of inference rules that is sound and complete for implication of these keys. In developing these results, we also investigate value-equality on XML subtrees and containment of path expressions, which are not only interesting in their own right but also important in the study of decision problems for XML keys.

Despite the importance of key analyses for XML, little previous work has studied this issue. The only closely related work is [10, 11]. For a class of keys and foreign keys, the decision problems were studied in the absence [11] and presence [10] of DTDs. The keys considered there are defined in terms of XML attributes and are not as expressive as keys studied in this paper.[1] Integrity constraints defined in terms of navigation paths have been studied for semistructured data [1] and XML in [3, 7–9]. These constraints are generalizations of inclusion dependencies and are not capable of expressing keys. Generalizations of functional dependencies have also been studied [13, 16]. However these generalizations were investigated in database settings, which are quite different from the tree model for XML data. Surveys on XML constraints can be found in [6, 20].

The remainder of the paper is organized as follows. Section 2 defines value equality and (absolute and relative) keys for XML. Section 3 establishes the finite axiomatizability and complexity results: First, we give a quadratic time algorithm for determining inclusion of path expressions. The ability to determine inclusion of path expressions is then used in developing inference rules for keys, for which a PTIME algorithm is given. Finally, Sec. 4 identifies directions for further research. All the proofs are given in the full version of the paper [5].

2 Keys

As illustrated in Fig. 1, our notion of keys is based on a tree model of XML data. Although the model is quite simple, we need to do two things prior to defining keys: the first is to give a precise definition of value equality for XML keys; the second is to describe a path language that will be used to locate sets of nodes in an XML document. We therefore introduce a class of regular path expressions, and define keys in terms of this path language.

2.1 A Tree Model and Value Equality

An XML document is typically modeled as a node-labeled tree. We assume three pairwise disjoint sets of labels: **E** of element tags, **A** of attribute names, and a singleton set $\{\mathtt{S}\}$ denoting text (PCDATA).

Definition 1. *An* XML tree *is defined to be* $T = (V, lab, ele, att, val, r)$*, where (1)* V *is a set of nodes; (2)* lab *is a mapping* $V \to \mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$ *which assigns a label to each node in* V*; a node* v *in* V *is called an* element *(*E node*) if* $lab(v) \in \mathbf{E}$*, an* attribute *(*A node*) if* $lab(v) \in \mathbf{A}$*, and a text node (*S node*) if* $lab(v) = \mathtt{S}$*; (3)* ele *and* att *are partial mappings that define the edge relation of* T*: for any node* v *in* V*,*

– *if* v *is an element then* $ele(v)$ *is a* sequence *of elements and text nodes in* V *and* $att(v)$ *is a* set *of attributes in* V*; for each* v' *in* $ele(v)$ *or* $att(v)$*,* v' *is called a* child *of* v *and we say that there is a (directed) edge from* v *to* v'*;*

[1] We do not consider foreign keys and DTDs in the current paper.

– *if v is an attribute or a text node then $ele(v)$ and $att(v)$ are undefined;*

(4) val is a partial mapping that assigns a string to each attribute and text node: for any node v in V, if v is an A or S node then $val(v)$ is a string, and $val(v)$ is undefined otherwise; (5) r is the unique and distinguished root node. An XML tree has a tree structure, i.e., for each $v \in V$, there is a unique path of edges from root r to v. An XML tree is said to be finite *if V is finite.*

For example, Fig. 1 depicts an XML tree that represents an XML document.

With this, we are ready to define value equality on XML trees. Let $T = (V, lab, ele, att, val, r)$ be an XML tree, and n_1, n_2 be two nodes in V. Informally, n_1, n_2 are value equal if they have the same tag (label) and in addition, either they have the same (string) value (when they are `S` or `A` nodes) or their children are pairwise value equal (when they are `E` nodes). More formally:

Definition 2. *Two nodes n_1 and n_2 are* value equal*, denoted by $n_1 =_v n_2$, iff (1) $lab(n_1) = lab(n_2)$; (2) if n_1, n_2 are A or S nodes then $val(n_1) = val(n_2)$; and (3) if n_1, n_2 are E nodes, then (a) for any $a_1 \in att(n_1)$, there exists $a_2 \in att(n_2)$ such that $a_1 =_v a_2$, and vice versa; and (b) if $ele(n_1) = [v_1, \ldots, v_k]$, then $ele(n_2) = [v'_1, \ldots, v'_k]$ and for all $i \in [1, k]$, $v_i =_v v'_i$. That is, $n_1 =_v n_2$ iff their subtrees are isomorphic by an isomorphism that is the identity on string values.*

As an example, in Fig. 1, the `author` subelement of the first `book` and the first `author` subelement of the second `book` are value equal.

2.2 Path Languages

There are many options for a path language, ranging from very simple ones involving just labels to more expressive ones such as regular languages or even XPath. However, to develop inference rules for keys we need to be able to reason about inclusion of path expressions (the *containment* problem). It is well known that for regular languages, the containment problem is not finitely axiomatizable; and for XPath, although nothing is known at this point we strongly suspect that it is not much easier. We therefore restrict our attention to the path language PL, which is expressive enough to be interesting yet simple enough to be reasoned about efficiently. We will also use a simpler language (PL_s) in defining keys, and therefore show both languages in the table below.

Path Language	Syntax
PL_s	$\rho \ ::= \ \epsilon \mid l.\rho$
PL	$q \ ::= \ \epsilon \mid l \mid q.q \mid \_*$

In PL_s, a path is a (possibly empty) sequence of node labels. Here ϵ denotes the empty path, node label $l \in \mathbf{E} \cup \mathbf{A} \cup \{\mathtt{S}\}$, and "." is a binary operator that concatenates two path expressions. The language PL is a generalization of PL_s that allows "_*", a combination of wildcard and Kleene closure. This symbol represents any (possibly empty) finite sequence of node labels. These languages are fragments of regular expressions [14], with PL_s contained in PL.

A path in PL_s is used to describe a path in an XML tree T, and a path expression in PL describes a set of such paths. Recall that an attribute node or a text node is a leaf in T and it does not have any child. Thus a path ρ in PL_s is said to be *valid* if for any label l in ρ, if $l \in \mathbf{A}$ or $l = \mathtt{S}$, then l is the last symbol in ρ. Similarly, we define *valid* path expressions of PL. In what follows we assume that the regular language defined by a path expression of PL contains only valid paths. For example, $book.author.name$ is a valid path in PL_s and PL, while $\_*.author$ is a valid path expression in PL but it is not in PL_s.

We now give some notation that will be used throughout the rest of the paper. Let ρ be a path in PL_s, P a path expression in PL and T an XML tree.
Length. The *length* of path ρ, denoted by $|\rho|$, is the number of labels in ρ (the empty path has length 0). By treating "$\_*$" as a special label, we also define the length of PL expression P, denoted by $|P|$, to be the number of labels in P.
Membership. We use $\rho \in P$ to denote that path ρ is in the regular language defined by path expression P. For example, $book.author.name \in book.author.name$ and $book.author.name \in \_*.name$.
Reachability. Let n_1, n_2 be nodes in T. We say that n_2 is *reachable* from n_1 by following path ρ, denoted by $T \models \rho(n_1, n_2)$, iff $n_1 = n_2$ if $\rho = \epsilon$, and if $\rho = \rho'.l$, then there exists n in T such that $T \models \rho'(n_1, n)$ and n_2 is a child of n with label l. We say that node n_2 is *reachable* from n_1 by following path expression P, denoted by $T \models P(n_1, n_2)$, iff there is a path $\rho \in P$ such that $T \models \rho(n_1, n_2)$. For example, if T is the XML tree in Fig. 1, then all the `name` nodes are reachable from the root by following $book.author.name$ and also by following $\_*$.
Node set. Let n be a node in T. We use $n[\![P]\!]$ to denote the set of nodes in T that can be reached by following the path expression P from node n. That is, $n[\![P]\!] = \{n' \mid T \models P(n, n')\}$. We shall use $[\![P]\!]$ as abbreviation for $r[\![P]\!]$, when r is the root node of T. For example, referring to Fig. 1 and let n be the first `book` element, then $n[\![chapter]\!]$ is the set of all `chapter` elements of the first book and $[\![\_*.chapter]\!]$ is the set of all `chapter` elements in the entire document.

Definition 3. *The* value intersection *of node sets $n_1[\![P]\!]$ and $n_2[\![P]\!]$, denoted by $n_1[\![P]\!] \cap_v n_2[\![P]\!]$, is defined by:*

$$n_1[\![P]\!] \cap_v n_2[\![P]\!] = \{(z, z') \mid \exists\, \rho \in P,\ z \in n_1[\![\rho]\!],\ z' \in n_2[\![\rho]\!],\ z =_v z'\}$$

That is, $n_1[\![P]\!] \cap_v n_2[\![P]\!]$ consists of node pairs that are value equal and are reachable by following the same simple path in the language defined by P starting from n_1 and n_2, respectively. For example, let n_1 and n_2 be the first and second `book` elements in Fig. 1, respectively. Then $n_1[\![author]\!] \cap_v n_2[\![author]\!]$ is the set consisting of a single pair (x, y), where x is the `author` subelement of the first book and y is the first `author` subelement of the second book.

2.3 A Key Constraint Language for XML

We are now in a position to define keys for XML and what it means for an XML document to satisfy a key constraint.

Definition 4. *A* key constraint φ *for XML is an expression of the form*

$$(Q, (Q', \{P_1, \ldots, P_k\})),$$

where Q, Q' *and* P_i *are* PL *expressions such that for all* $i \in [1, k]$, $Q.Q'.P_i$ *is a valid path expression. The path* Q *is called the* context path, Q' *is called the* target path, *and* P_1, ..., P_k *are called the* key paths *of* φ.

When $Q = \epsilon$, *we call* φ *an* absolute key, *abbreviated to* $(Q', \{P_1, \ldots, P_k\})$; *otherwise* φ *is called a* relative *key. We use* $\mathcal{K}$ *to denote the language of keys, and* $\mathcal{K}_{abs}$ *to denote the set of absolute keys in* $\mathcal{K}$.

A key $\varphi = (Q, (Q', \{P_1, \ldots, P_k\}))$ specifies the following: (1) the context path Q, starting from the root of an XML tree T, identifies a set of nodes $[\![Q]\!]$; (2) for each node $n \in [\![Q]\!]$, φ defines an absolute key $(Q', \{P_1, \ldots, P_k\})$ that is to hold on the subtree rooted at n; specifically,

- the target path Q' identifies a set of nodes $n[\![Q']\!]$ in the subtree, referred to as the *target set*,
- the key paths $P_1, \ldots, P_k$ identify nodes in the target set. That is, for each $n' \in n[\![Q']\!]$ the values of the nodes reached by following the key paths from n' uniquely identify n' in the target set.

For example, the keys on Fig. 1 mentioned in Sec. 1 can be written as follows:
(1) `@isbn` is a key of `book` nodes: $(book, \{@isbn\})$;
(2) `name` is a key of `author` nodes no matter where they are: $(\_*.author, \{name\})$;
(3) within each subtree rooted at a `book`, `@number` is a key of `chapter` relative to `book`: $(book, (chapter, \{@number\}))$.
The first two are absolute keys of $\mathcal{K}_{abs}$ and the last one is a relative key of $\mathcal{K}$.

Definition 5. *Let* $\varphi = (Q, (Q', \{P_1, \ldots, P_k\}))$ *be a key of* $\mathcal{K}$. *An XML tree* T satisfies φ, *denoted by* $T \models \varphi$, *iff for any* n *in* $[\![Q]\!]$ *and any* n_1, n_2 *in* $n[\![Q']\!]$, *if for all* $i \in [1, k]$ *there exist a path* $\rho \in P_i$ *and nodes* $x \in n_1[\![\rho]\!]$, $y \in n_2[\![\rho]\!]$ *such that* $x =_v y$, *then* $n_1 = n_2$. *That is,*

$$\forall n \in [\![Q]\!] \;\; \forall n_1\, n_2 \in n[\![Q']\!] \;\; ((\bigwedge_{1 \le i \le k} n_1[\![P_i]\!] \cap_v n_2[\![P_i]\!] \neq \emptyset) \rightarrow \; n_1 = n_2).$$

As an example, let us consider $\mathcal{K}$ constraints on the XML tree T in Fig. 1.
(1) $T \models (book, \{@isbn\})$ because the `@isbn` attributes of the two `book` nodes in T have different string values. However, $T \not\models (book, \{author\})$ because the two books agree on the values of their first author.
(2) $T \not\models (\_*.author, \{name\})$ because the `author` of the first `book` and the first `author` of the second `book` agree on their names but they are distinct nodes.
(3) $T \models (book, (chapter, \{@number\}))$ because in the subtree rooted at each `book` node, the `@number` attribute of each `chapter` has a distinct value.

Several subtleties are worth pointing out. First, observe that each key path can specify a *set* of values. For example, consider $\psi = (book, \{@isbn, author\})$ on the XML tree T in Fig. 1, and note that the key path `author` reaches two `author` subelements from the second `book` node. In contrast, this is not allowed in most proposals for XML keys, e.g., XML Schema. The reason that we allow a key path to reach multiple nodes is to cope with the semistructured nature of XML data. Second, the key has no impact on those nodes at which some key path is *missing*. Observe that for any $n \in [\![Q]\!]$ and n_1, n_2 in $n[\![Q']\!]$, if P_i is missing at either n_1 or n_2 then $n_1[\![P_i]\!]$ and $n_2[\![P_i]\!]$ are by definition disjoint. This is similar to *unique constraints* introduced in XML Schema. In contrast to unique constraints, however, our notion of keys is capable of comparing nodes at which a key path may have multiple values. Third, it should be noted that two notions of equality are used to define keys: value equality ($=_v$) when comparing nodes reached by following key paths, and node identity ($=$) when comparing two nodes in the target set. This is a departure from keys in relational databases, in which only value equality is considered.

Our definition of a key allows key values to be "scoped" by their paths. As an example, the XML data in Fig. 2.a satisfies the key `(part, {_*.@id})`, and the XML data in Fig. 2.b satisfies the key `(book, {_*.isbn})`. Although in the first example our definition of keys captures the intended meaning, we would probably want the second example to violate the key[2]. That is, one might want `isbn` to be a key for `book` no matter where it occurs in a book. It is possible to reformulate our constraint language to be able to express both examples by modifying the definition of value intersection (Def. 3), but we do not yet know whether the proofs in this paper can be extended to a more general definition.

```
⟨part⟩
    ⟨widget id=1⟩⟨/widget⟩
⟨/part⟩
⟨part⟩
    ⟨gadget id=1⟩⟨/gadget⟩
⟨/part⟩
```

(a)

```
⟨book isbn=123⟩
⟨/book⟩
⟨book⟩
    ⟨identifier isbn=123/⟩
⟨/book⟩
```

(b)

Fig. 2. XML data and scope of key paths

2.4 Decision Problems

As mentioned in Sec. 1, the satisfiability and implication analyses of XML keys are far more intriguing than their relational databases counterpart.

We first consider satisfiability of keys of our constraint language $\mathcal{K}$. Let Σ be a finite set of keys in $\mathcal{K}$ and T be an XML tree. We use $T \models \Sigma$ to denote that T *satisfies* Σ. That is: for any $\psi \in \Sigma$, $T \models \psi$.

The *(finite) satisfiability problem* for $\mathcal{K}$ is to determine, given any finite set Σ of keys in $\mathcal{K}$, whether there exists a (finite) XML tree satisfying Σ.

As observed in Sec. 1, keys defined in some proposals (e.g., XML Schema) may not be finitely satisfiable at all. In contrast, any key constraints of $\mathcal{K}$ can always be satisfied by a finite XML tree, including the single node tree. That is,

[2] We are grateful to one of the referees for pointing this out and for providing the example.

Observation. Any finite set Σ of keys in $\mathcal{K}$ is finitely satisfiable.

Next, we consider implication of $\mathcal{K}$ constraints. Let $\Sigma \cup \{\varphi\}$ be a finite set of keys of $\mathcal{K}$. We use $\Sigma \models \varphi$ to denote Σ *implies* φ; that is, for any XML tree T, if $T \models \Sigma$, then $T \models \varphi$.

There are two implication problems associated with keys: The *implication problem* is to determine, given any finite set of keys $\Sigma \cup \{\varphi\}$, whether $\Sigma \models \varphi$. The *finite implication problem* is to determine whether Σ *finitely implies* φ, that is, whether it is the case that for any finite XML tree T, if $T \models \Sigma$, then $T \models \varphi$.

Given any finite set $\Sigma \cup \{\varphi\}$ of keys in $\mathcal{K}$, if there is an XML tree T such that $T \models \bigwedge \Sigma \wedge \neg\varphi$, then there must be a finite XML tree T' such that $T' \models \bigwedge \Sigma \wedge \neg\varphi$. Thus key implication has the finite model property (see [5] for a proof) and as a result:

Proposition 1. *The implication and finite implication problems for keys coincide.*

In light of this we can also use $\Sigma \models \varphi$ to denote that Σ finitely implies φ.

3 Key Implication

We now study the finite implication problem for keys. Our main result is:

Theorem 1. *The finite implication problem for $\mathcal{K}$ is finitely axiomatizable and decidable in PTIME in the size of keys.*

We provide a finite axiomatization and a PTIME algorithm for determining finite implication of $\mathcal{K}$ constraints. In contrast to their relational database counterparts, the axiomatization and algorithm are nontrivial. A road map for the proof of the theorem is as follows. We first study containment of path expressions in the language PL defined in the last section, since the axioms rely on path inclusion. We then provide a finite set of inference rules and show that it is sound and complete for finite implication of $\mathcal{K}$ constraints. Finally, taking advantage of the inference rules, we develop a PTIME algorithm for determining finite implication. We shall also present complexity results in connection with finite implication of absolute keys in $\mathcal{K}_{abs}$.

3.1 Inclusion of *PL* Expressions

A PL expression P is said to be *included* (or *contained*) in PL expression Q, denoted by $P \subseteq Q$, if for any XML tree T and any node n in T, $n[\![P]\!] \subseteq n[\![Q]\!]$. That is, the nodes reached from n by following P are contained in the set of the nodes reached by following Q from n. We write $P = Q$ if $P \subseteq Q$ and $Q \subseteq P$.

In the absence of DTDs, $P \subseteq Q$ is equivalent to the containment of the regular language defined by P in the regular language defined by Q. Indeed, if there exists a path $\rho \in P$ but $\rho \notin Q$, then one can construct an XML tree T with a path ρ from the root. It is obvious that in T, $[\![P]\!] \not\subseteq [\![Q]\!]$. The other direction is immediate. Therefore, $P \subseteq Q$ iff for any path $\rho \in P$, $\rho \in Q$.

$$\frac{P \in PL}{\epsilon.P \subseteq P \quad P \subseteq \epsilon.P \quad P.\epsilon \subseteq P \quad P \subseteq P.\epsilon} \text{ (empty-path)}$$

$$\frac{P \in PL}{P \subseteq P} \text{ (reflexivity)} \qquad \frac{P \in PL}{P \subseteq \_*} \text{ (star)}$$

$$\frac{P \subseteq P' \quad Q \subseteq Q'}{P.Q \subseteq P'.Q'} \text{ (composition)} \qquad \frac{P \subseteq Q \quad Q \subseteq R}{P \subseteq R} \text{ (transitivity)}$$

Table 1. $\mathcal{I}^p$**: rules for** PL **expression inclusion**

We investigate inclusion (containment) of path expressions in PL: given any PL expressions P and Q, is it the case that $P \subseteq Q$? As will become clear shortly, this is important to the proof of Theorem 1. It is decidable with a low complexity:

Theorem 2. *There are a sound and complete finite set of inference rules and a quadratic time algorithm for determining inclusion of* PL *expressions.*

It is worth mentioning that PL is a star-free regular language (cf. [28] for a definition). The inclusion problem for general star-free languages is co-NP complete [15]. For inclusion of PL expression, we are able to provide a set of inference rules in Table 1, denoted by $\mathcal{I}^p$, and to develop a quadratic time algorithm.

Proof sketch: The soundness of $\mathcal{I}^p$ can be verified by induction on the lengths of $\mathcal{I}^p$-proofs. The proof of completeness is based on a simulation relation defined on the nondeterministic finite automata (NFA [14]) that characterize PL expressions. More specifically, let the NFA for PL expressions P and Q be $M(P) = (N_1, C \cup \{\_\}, \delta_1, S_1, F_1)$ and $M(Q) = (N_2, C \cup \{\_\}, \delta_2, S_2, F_2)$ respectively. Observe that the alphabets of the NFA have been extended with the special character "_" which can match any letter in C. We define a *simulation relation*, $\lhd$, on $N_1 \times N_2$. For any $n_1 \in N_1$ and $n_2 \in N_2$, $n_1 \lhd n_2$ iff the following conditions are satisfied: (1) If $n_1 = F_1$ then $n_2 = F_2$. (2) If $\delta_1(n_1, \_) = n_1$ then $\delta_2(n_2, \_) = n_2$. (3) For any $l \in C$, if $\delta_1(n_1, l) = n_1'$ for some $n_1' \in N_1$, then either (a) there exists a state $n_2' \in N_2$ such that $\delta_2(n_2, l) = n_2'$ and $n_1' \lhd n_2'$, or (b) $\delta_2(n_2, \_) = n_2$ and $n_1' \lhd n_2$. The simulation is defined in such a way that $P \subseteq Q$ is equivalent to $S_1 \lhd S_2$. Intuitively, this means that starting with the start states of $M(P)$ and $M(Q)$ and given an input string, every step taken by $M(P)$ in accepting this string has a corresponding step in $M(Q)$ according to the simulation relation. In light of $\mathcal{I}^p$ and the claims, we provide in Algorithm 1 a recursive function $Incl(n_1, n_2)$ for testing inclusion of PL expressions. We use $visited(n_1, n_2)$ to keep track of whether $Incl(n_1, n_2)$ has been evaluated before, which ensures that each pair (n_1, n_2) is checked at most once. The function $Incl(n_1, n_2)$ returns true iff $n_1 \lhd n_2$. Since $P \subseteq Q$ iff $S_1 \lhd S_2$, $P \subseteq Q$ iff $Incl(S_1, S_2)$. Its complexity is kept low by the use of the boolean $visited$. See [5] for details. ∎

Algorithm 1. $Incl(n_1,\, n_2)$

1. if $visited(n_1, n_2)$ then return false else mark $visited(n_1, n_2)$ as true;
2. process $n_1,\, n_2$ as follows:
   Case 1: if $n_1 = F_1$ then
         if $n_2 = F_2$ and $(\delta_1(F_1, \_) = \emptyset$ or $\delta_2(F_2, \_) = F_2)$
         then return true;
         else return false;
   Case 2: if $\delta_1(n_1, a) = n_1'$ and $\delta_2(n_2, a) = n_2'$ for letter a
         and $\delta_1(n_1, \_) = \emptyset$ and $\delta_2(n_2, \_) = \emptyset$
      then return $Incl(n_1',\, n_2')$;
   Case 3: if $\delta_1(n_1, a) = n_1'$ and $\delta_2(n_2, \_) = n_2$ and $\delta_2(n_2, a) = n_2'$ for letter a
      then return $(Incl(n_1',\, n_2)$ or $Incl(n_1',\, n_2'))$
      else if $\delta_1(n_1, a) = n_1'$ and $\delta_2(n_2, \_) = n_2$ and $\delta_2(n_2, a) = \emptyset$
         then return $Incl(n_1',\, n_2)$;
3. return false

3.2 Axiomatization for Absolute Key Implication

Recall that an absolute key (Q', S) is a special case of a $\mathcal{K}$ constraint $(Q, (Q', S))$, i.e., when $Q = \epsilon$. As opposed to relative keys, absolute keys are defined on the entire XML tree T rather than on certain subtrees of T. The problem of determining implication of absolute keys is simpler than that for relative keys. Since most of the rules for relative key implication are an obvious generalization of those for absolute keys, we start by giving a discussion on the rules for absolute key implication. The set of rules, denoted by $\mathcal{I}_{abs}$, is shown in Table 2.

$\dfrac{(Q,\, S) \quad P \in PL}{(Q,\, S \cup \{P\})}$	(superkey)	$\dfrac{(Q,\, S \cup \{P_i, P_j\}) \quad P_i \subseteq P_j}{(Q,\, S \cup \{P_i\})}$	(containment-reduce)
$\dfrac{(Q.Q',\, \{P\})}{(Q,\, \{Q'.P\})}$	(subnodes)	$\dfrac{(Q,\, S) \quad Q' \subseteq Q}{(Q',\, S)}$	(target-path-containment)
$\dfrac{(Q,\, S \cup \{\epsilon, P\}) \quad P' \in PL}{(Q,\, S \cup \{\epsilon, P.P'\})}$	(prefix-epsilon)	$\dfrac{S \text{ is a set of } PL \text{ expressions}}{(\epsilon,\, S)}$	(epsilon)

Table 2. $\mathcal{I}_{abs}$**: Rules for absolute key implication**

- *superkey.* If S is a key for the nodes in $[\![Q]\!]$ then so is any superset of S. This is the only rule of $\mathcal{I}_{abs}$ that has a counterpart in relational key inference.
- *subnodes.* Since we have a tree model, any node $v \in [\![Q.Q']\!]$ must be in the subtree rooted at a unique node v' in $[\![Q]\!]$. Therefore, if a key path P identifies a node in $[\![Q.Q']\!]$ then $Q'.P$ uniquely identifies nodes in $[\![Q]\!]$.

- *prefix-epsilon*. Note that $n_1 =_v n_2$ if $n_1[\![\epsilon]\!] \cap_v n_2[\![\epsilon]\!] \neq \emptyset$. In addition, for any $n_1, n_2 \in [\![Q]\!]$, if $n_1[\![P.P']\!] \cap_v n_2[\![P.P']\!] \neq \emptyset$ and $n_1 =_v n_2$, then $n_1[\![P]\!] \cap_v n_2[\![P]\!] \neq \emptyset$. Thus by the definition of keys, $S \cup \{\epsilon, P.P'\}$ is also a key for $[\![Q]\!]$.
- *containment-reduce*. For any nodes n_1, n_2 in $[\![Q]\!]$, if $n_1[\![P_i]\!] \cap_v n_2[\![P_i]\!] \neq \emptyset$, then we must have $n_1[\![P_j]\!] \cap_v n_2[\![P_j]\!] \neq \emptyset$ given $P_i \subseteq P_j$. Thus by the definition of keys $S \cup \{P_i\}$ is also a key for $[\![Q]\!]$.
- *target-path-containment*. A key for the set $[\![Q]\!]$ is also a key for any subset of $[\![Q]\!]$. Observe that $[\![Q']\!] \subseteq [\![Q]\!]$ if $Q' \subseteq Q$.
- *epsilon*. There is only one root, and thus any set of PL expressions forms a key for the root.

We omit the proof of the following theorem. Details can be found in [5].

Theorem 3. *The set $\mathcal{I}_{abs}$ is sound and complete for (finite) implication of absolute keys of $\mathcal{K}_{abs}$. In addition, the problem can be determined in $O(n^4)$ time.*

3.3 Axiomatization for Key Implication

We now turn to the finite implication problem for $\mathcal{K}$, and start by giving in Table 3 a set of inference rules, denoted by $\mathcal{I}$. Most rules are simply generalizations of rules shown in Table 2. The only exceptions are rules that deal with the context path in relative keys. We briefly illustrate these rules below.

- *context-path-containment*. If (Q', S) holds on all subtrees rooted at nodes in $[\![Q]\!]$, then it also holds on subtrees rooted at nodes in subset $[\![Q_1]\!]$ of $[\![Q]\!]$.
- *context-target*. If in a tree T rooted at a node n in $[\![Q]\!]$, S is a key for $n[\![Q_1.Q_2]\!]$, then in any subtree of T rooted at n' in $n[\![Q_1]\!]$, S is a key for $n'[\![Q_2]\!]$. In particular, when $Q = \epsilon$ this rules says that if the (absolute) key holds on the entire document, then it must also hold on any sub-document.
- *interaction*. By the first key in the precondition, in each subtree rooted at a node n in $[\![Q_1]\!]$, $Q'.P_1, \ldots, Q'.P_k$ uniquely identify a node in $n[\![Q_2]\!]$. The second key in the precondition prevents the existence of more than one Q' node under Q_2 that coincide in their $P_1, \ldots, P_k$ nodes. Therefore, $P_1, \ldots, P_k$ uniquely identify a node in $n[\![Q_2.Q']\!]$ in each subtree rooted at n in $[\![Q_1]\!]$.

Note that key inference in the XML setting relies heavily on path inclusion. That is why we need to develop inference rules for PL expression inclusion.

Given a finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, we use $\Sigma \vdash_{\mathcal{I}} \varphi$ to denote that φ is provable from Σ using $\mathcal{I}$ (and $\mathcal{I}_p$ for path inclusion).

We next show that $\mathcal{I}$ is indeed an axiomatization for $\mathcal{K}$ constraint implication.

Lemma 1. *The set $\mathcal{I}$ is sound and complete for finite implication of $\mathcal{K}$ constraints. That is, for any finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, $\Sigma \models \varphi$ iff $\Sigma \vdash_{\mathcal{I}} \varphi$.*

Proof sketch: Soundness of $\mathcal{I}$ can be verified by induction on the lengths of $\mathcal{I}$-proofs. For the proof of completeness, we show that if $\Sigma \not\vdash_{\mathcal{I}} \varphi$, then there exists a finite XML tree G such that $G \models \Sigma$ and $G \models \neg\varphi$, i.e., $\Sigma \not\models \varphi$. In other words, if $\Sigma \models \varphi$ then $\Sigma \vdash_{\mathcal{I}} \varphi$. See [5] for the details of the proof. ∎

Finally, we show that $\mathcal{K}$ constraint implication is decidable in PTIME.

$\dfrac{(Q,\ (Q',\ S)) \quad P \in PL}{(Q,\ (Q',\ S \cup \{P\}))}$	(superkey)
$\dfrac{(Q,\ (Q'.Q'',\ \{P\}))}{(Q,\ (Q',\ \{Q''.P\}))}$	(subnodes)
$\dfrac{(Q,\ (Q',\ S \cup \{P_i, P_j\})) \quad P_i \subseteq P_j}{(Q,\ (Q',\ S \cup \{P_i\}))}$	(containment-reduce)
$\dfrac{(Q,\ (Q',\ S)) \quad Q_1 \subseteq Q}{(Q_1,\ (Q',\ S))}$	(context-path-containment)
$\dfrac{(Q,\ (Q',\ S)) \quad Q_2 \subseteq Q'}{(Q,\ (Q_2,\ S))}$	(target-path-containment)
$\dfrac{(Q,\ (Q_1.Q_2,\ S))}{(Q.Q_1,\ (Q_2,\ S))}$	(context-target)
$\dfrac{(Q,\ (Q',\ S \cup \{\epsilon, P\})) \quad P' \in PL}{(Q,\ (Q',\ S \cup \{\epsilon, P.P'\}))}$	(prefix-epsilon)
$\dfrac{\begin{array}{c}(Q_1,\ (Q_2,\ \{Q'.P_1, \ldots, Q'.P_k\})) \\ (Q_1.Q_2,\ (Q',\ \{P_1, \ldots, P_k\}))\end{array}}{(Q_1,\ (Q_2.Q',\ \{P_1, \ldots, P_k\}))}$	(interaction)
$\dfrac{Q \in PL,\ S \text{ is a set of } PL \text{ expressions}}{(Q,\ (\epsilon,\ S))}$	(epsilon)

Table 3. $\mathcal{I}$**: Inference rules for key implication**

Lemma 2. *There is an algorithm that, given any finite set* $\Sigma \cup \{\varphi\}$ *of* $\mathcal{K}$ *constraints, determines whether* $\Sigma \models \varphi$ *in PTIME.*

Proof sketch: In Algorithm 2 we provide a function for determining finite implication of $\mathcal{K}$ constraints. The correctness of the algorithm follows from Lemma 1 and its proof. It applies $\mathcal{I}$ rules to derive φ if $\Sigma \models \varphi$. The overall cost of the algorithm is $O(n^8)$, where n is the size of keys involved, and therefore we have a PTIME algorithm. The details of the proof can be found in [5]. ∎

Theorem 1 follows from Lemmas 1 and 2.

4 Discussion

We have investigated the (finite) satisfiability and (finite) implication problems associated with the XML key constraint language introduced in [4]. These keys are capable of expressing many important properties of XML data; moreover, in contrast to other proposals, this language can be reasoned about efficiently. More specifically, keys defined in this language are always finitely satisfiable, and their

Algorithm 2. Finite implication of $\mathcal{K}$ constraints

Input: a finite set $\Sigma \cup \{\varphi\}$ of $\mathcal{K}$ constraints, where $\varphi = (Q, (Q', \{P_1, ..., P_k\}))$
Output: true iff $\Sigma \models \varphi$

// Epsilon rule.
1. if $Q' = \epsilon$ then output true and terminate

// Containment-reduce rule.
2. for each $(Q_i, (Q'_i, S_i)) \in \Sigma \cup \{\varphi\}$ do
    repeat until no further change
        if $S_i = S \cup \{P', P''\}$ such that $P' \subseteq P''$ then $S_i := S_i \setminus \{P''\}$
3. $X := \emptyset$;

// Use the containment rules, context-target, superkey, subnodes, prefix-epsilon, and interaction.
4. repeat until no keys in Σ can be applied in cases (a)-(d).
  for each $\phi = (Q_\phi,\ (Q'_\phi,\ \{P'_1, ..., P'_m\})) \in \Sigma$ do
   // Prove φ when Q_ϕ contains a prefix of Q.
   (a) if there is Q_t, R_p in PL such that $Q \subseteq Q_\phi.Q_t$, $Q_t.Q'.R_p \subseteq Q'_\phi$, $R_p = \epsilon$ if $m > 1$ and
     for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
     (i) $P_s \subseteq R_p.P'_j$ or
     (ii) there exists $l \in [1, k]$ and R_j in PL such that $P_l = \epsilon$ and $P_s \subseteq R_p.P'_j.R_j$
    then output true and terminate

   // Prove φ when Q is contained in a prefix of Q_ϕ.
   (b) if there are Q_c, Q_t, R_p in PL such that
     $Q.Q_c \subseteq Q_\phi$, $Q'.R_p \subseteq Q_c.Q'_\phi$, $R_p = \epsilon$ if $m > 1$, $Q' = Q_c.Q_t$ and
     for all $j \in [1, m]$ there is there is $s \in [1, k]$ such that either
     (i) $P_s \subseteq R_p.P'_j$ or
     (ii) there exists $l \in [1, k]$ and R_j in PL such that $P_l = \epsilon$ and $P_s \subseteq R_p.P'_j.R_j$;
     and moreover, there is $(Q,\ (Q_c,\ \{Q_t.P_1, ..., Q_t.P_k\}))$ in X
    then output true and terminate

   // Produce intermediate results in X when Q_ϕ contains a prefix of Q.
   (c) if there are Q_c, Q_t, R_p in PL such that $Q \subseteq Q_\phi.Q_c$, $Q_c.Q' \subseteq Q'_\phi.R_p$, $Q' = Q_t.R_p$ and
     for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
     (i) $R_p.P_s \subseteq P'_j$ or
     (ii) there exists $l \in [1, k]$ and R_j in PL such that $P_l = \epsilon$ and $R_p.P_s \subseteq P'_j.R_j$
    then
    (1) if $m = 1$ then $X := X \cup \{(Q, (Q_1,\ \{Q_2.R_p.P_1, \ldots, Q_2.R_p.P_k\}))\}$
      where $Q_t = Q_1.Q_2$ for some $Q_1, Q_2 \in PL$;
    (2) if $m > 1$ then $X := X \cup \{(Q, (Q_t,\ \{R_p.P_1, \ldots, R_p.P_k\}))\}$;
    (3) $\Sigma := \Sigma \setminus \{\phi\}$;

   // Produce intermediate results in X when Q is contained in a prefix of Q_ϕ.
   (d) if there are Q_c, Q_t, R_p in PL such that $Q.Q_c \subseteq Q_\phi$, $Q' \subseteq Q_c.Q'_\phi.R_p$, $Q' = Q_c.Q_t.R_p$ and
     for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
     (i) $R_p.P_s \subseteq P'_j$ or
     (ii) there exists $l \in [1, k]$ and R_j in PL such that $P_l = \epsilon$ and $R_p.P_s \subseteq P'_j.R_j$;
     and moreover, there is $(Q,\ (Q_c,\ \{Q_t.R_p.P_1, ..., Q_t.R_p.P_k\}))$ in X
    then
    (1) if $m = 1$ then $X := X \cup \{(Q, (Q_1,\ \{Q_2.R_p.P_1, \ldots, Q_2.R_p.P_k\}))\}$
      where $Q_c.Q_t = Q_1.Q_2$ for some $Q_1, Q_2 \in PL$;
    (2) if $m > 1$ then $X := X \cup \{(Q, (Q_c.Q_t,\ \{R_p.P_1, \ldots, R_p.P_k\}))\}$;
    (3) $\Sigma := \Sigma \setminus \{\phi\}$;
5. output false

(finite) implication is finitely axiomatizable and decidable in PTIME in the size of keys. We believe that these key constraints are simple yet expressive enough to be adopted by XML designers and users.

For further research, a number of issues deserve investigation. First, our results are established in the absence of DTDs. Despite their simple syntax, there is an interaction between DTDs and our key constraints. To illustrate this, let us consider a simple DTD D:

```
<!ELEMENT  foo (X, X)>
<!ELEMENT  X   (empty)>
```

and a simple (absolute) key $\varphi = (X, \emptyset)$. Obviously, there exists a finite XML tree that conforms to the DTD D and there exists another finite XML tree that satisfies the key φ. However, there is no XML tree that both conforms to D and satisfies φ, because D requires an XML tree to have two distinct X elements, whereas φ requires that the path X, if it exists, must be unique at the root. This shows that in the presence of DTDs, the analysis of key satisfiability and implication can be wildly different. It should be mentioned that keys defined in other proposals for XML, such as XML Schema [26], also interact with DTDs or other type systems for XML. This issue was recently investigated in [10].

Second, one might be interested in using a different path language to express keys. The containment problem for the full regular language is PSPACE-complete [12], and it is not finitely axiomatizable. Another alternative is the language of [17], which simply adds a single wildcard to PL. Despite the seemingly trivial addition, containment of expressions in their language is only known to be in PTIME. It is possible to develop an inclusion checking algorithm with a complexity comparable to the related result in this paper. For XPath [24] expressions, questions in connection with their containment and equivalence, as well as (finite) satisfiability and (finite) implication of keys defined in terms of these complex path expressions are, to the best of our knowledge, still open.

Third, along the same lines as our XML key language, a language of foreign keys needs to be developed for XML.

A final question is about key constraint checking. An efficient incremental checking algorithm for our keys is currently under development.

Acknowledgments. The authors thank Michael Benedikt, Chris Brew, Dave Maier, Keishi Tajima and Henry Thompson for helpful discussions. They would also like to thank one of the referees for pointing out the possible need for a more general definition of a key constraint (Sec. 2.)

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufman, 2000.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

3. S. Abiteboul and V. Vianu. Regular path queries with constraints. *Journal of Computer and System Sciences (JCSS)*, 58(4):428–452, 1999.
4. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW'10*, 2001.
5. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about absolute and relative keys for XML. Technical Report TUCIS-TR-2001-002, Temple University, 2001.
6. P. Buneman, W. Fan, J. Siméon, and S. Weinstein. Constraints for semistructured data and XML. *SIGMOD Record*, 30(1), 2001.
7. P. Buneman, W. Fan, and S. Weinstein. Path constraints on semistructured and structured data. In *PODS*, 1998.
8. P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. In *PODS*, 1999.
9. P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured databases. *Journal of Computer and System Sciences (JCSS)*, 61(2):146–193, 2000.
10. W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. In *PODS*, 2001.
11. W. Fan and J. Siméon. Integrity constraints for XML. In *PODS*, 2000.
12. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
13. C. S. Hara and S. B. Davidson. Reasoning about nested functional dependencies. In *PODS*, 1999.
14. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addision Wesley, 1979.
15. H. Hunt, D. Resenkrantz, and T. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *Journal of Computer and System Sciences (JCSS)*, 12:222–268, 1976.
16. M. Ito and G. E. Weddell. Implication problems for functional constraints on databases supporting complex objects. *Journal of Computer and System Sciences (JCSS)*, 50(1):165–187, 1995.
17. T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
18. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.
19. J. Robie, J. Lapp, and D. Schach. *XML Query Language (XQL)*. Workshop on XML Query Languages, Dec. 1998.
20. V. Vianu. A Web odyssey: From Codd to XML. In *PODS*, 2001.
21. W3C. *Document Object Model (DOM) Level 1 Specification*. Recommendation, Oct. 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.
22. W3C. *Extensible Markup Language (XML) 1.0*, Feb 1998. `http://www.w3.org/TR/REC-xml`.
23. W3C. XML-Data. Note, Jan. 1998. `http://www.w3.org/TR/1998/NOTE-XML-data`.
24. W3C. *XML Path Language (XPath)*. Working Draft, Nov. 1999. `http://www.w3.org/TR/xpath`.
25. W3C. *XSL Transformations (XSLT)*. Recommendation, Nov. 1999. `http://www.w3.org/TR/xslt`.
26. W3C. *XML Schema*. Working Draft, May 2001. `http://www.w3.org/XML/Schema`.
27. P. Wadler. A Formal Semantics for Patterns in XSL. Technical report, Computing Sciences Research Center, Bell Labs, Lucent Technologies, 2000.
28. S. Yu. Regular languages. In G. Rosenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 41–110. Springer, 1996.