Keys for XML

Peter Buneman[*] Susan Davidson[†] Wenfei Fan[‡] Carmem Hara[§]
Wang-Chiew Tan[¶]

(Draft of August 9, 2000)

Abstract

We discuss the definition of keys for XML documents, paying particular attention to the concept of a *relative key*, which is commonly used in hierarchically structured documents.

1 Introduction

Keys are an essential part of database design [2, 6]: they are fundamental to data models and conceptual design; they provide the means by which one tuple in a relational database may refer to another tuple; and they are important in update, for they enable us to guarantee that an update will affect precisely one tuple. More philosophically, if we think of a tuple as representing some real-world entity, the key provides an invariant connection between the tuple and entity.

If XML documents are to do double duty as databases, then we shall need keys for them. In fact, a cursory examination[1] of existing DTDs reveals a number of cases in which some element or attribute is specified – in comments – as a "unique identifier". Moreover a number of scientific databases, which are typically stored in some special-purpose hierarchical data format which is ripe for conversion to XML, have a well-organized hierarchical key structure.

Both the XML specification [4] itself and XML-Schema [7] include some form of specification of keys. Through the use of ID attributes in a document type descriptor (DTD) one can specify an identifier for an element that is unique within a document. XML-Schema has a more elaborate proposal which is the starting point for this note. There are a number of technical issues concerning the XML-Schema proposal, but the important point is that neither XML nor XML-Schema properly address the issue of hierarchical keys, which appear to be ubiquitous in hierarchically structured databases. This is the main reason for this note. Also, the authors believe that the use of keys for citing parts of a document is sufficiently important that it is appropriate to consider key specification independently of other proposals for constraining the structure of XML documents.

[*]University of Pennsylvania. `peter@cis.upenn.edu`.
[†]University of Pennsylvania. `susan@cis.upenn.edu`.
[‡]Temple University. `fan@joda.cis.temple.edu`.
[§]Universidade Federal do Parana, Brazil. `carmem@inf.ufpr.br`.
[¶]University of Pennsylvania. `wctan@saul.cis.upenn.edu`.

[1]Motivated in part by the "DTD Inquisitor" of Byron Choi and Arnaud Sahuguet [?, ?].

How then, are we to describe keys for XML or, more generally, for semistructured data? From the start, how we identify components of XML documents is very different from the way we identify components of relational databases. Consider the two structures:

```
⟨db⟩
   ⟨student⟩  ⟨name⟩ Smith ⟨/name⟩ ⟨course⟩ Math2 ⟨/course⟩ ⟨grade⟩ B ⟨/grade⟩  ⟨/student⟩
   ⟨student⟩  ⟨name⟩ Jones ⟨/name⟩ ⟨course⟩ Math2 ⟨/course⟩ ⟨grade⟩ A+ ⟨/grade⟩  ⟨/student⟩
   ⟨student⟩  ⟨name⟩ Brown ⟨/name⟩ ⟨course⟩ Phil5 ⟨/course⟩ ⟨grade⟩ A- ⟨/grade⟩  ⟨/student⟩
⟨/db⟩
```

name	course	grade
Smith	Math2	B
Jones	Math2	A+
Brown	Phil5	A-

To identify a tuple in the relation we need to know, say, that `name` and `course` constitute a key. In the absence of a key the only way we can be sure of uniquely identifying a tuple is to give the entire tuple. For relational databases, the way we specify a key constraint is to say that if two tuples agree on their key attributes they agree everywhere. By contrast, XML documents are, first of all, documents and we can therefore use the position in the document (say a byte offset) to identify some part of it. However, when faced with the problem of updates, it is best to use value-based identifiers rather than location-based identifiers, e.g. we would like to be able to say that if two elements agree on the `name` and `course` subelements then they are the same element. Put in the contrapositive: two distinct `student` elements must differ on a `name` or `course` subelement. This raises two issues that precede any discussion of the structure of keys: that of node identification and that of equality. The latter is a thorny topic, but needs some attention.

2 Node addresses and equality

The *Document Object Model* (DOM) [3] provides some insight into a semantics for XML documents. According to the DOM, a document is a hierarchical structure of nodes. Nodes are of several types, but there are three types that are important to this discussion: element nodes, attribute nodes, and text nodes. As illustrated in Figure 1 text nodes (T) have no name but carry text, attribute nodes (A) both have a name and carry text, and element nodes (E) have a name. Element nodes may have children; attribute and text nodes are terminal. In addition the DOM specifies how to reach the children of an element node. Text and element children are held in what is essentially an array, the index in the array being determined by the order of the subelements in the document. Attribute children are held in a dictionary. The name of the attribute, which must be unique within an element, is used as the index. These indexes, an integer for an element or text child, or the name prefixed by an "@" for attributes, are shown as edge labels in Figure 1. The important point here is that the edge labels uniquely identify children.

A consequence of this model is that a path of edge labels from the root uniquely identifies a node. We shall call such paths *node addresses* and write them $\langle l_1 \ldots l_n \rangle$, for example ⟨`1.2.1`⟩ and ⟨`1.3.@num`⟩. Node addresses will be our basic means of identifying nodes. Note that an attribute name can only occur at the end of a node address. We can also talk about the address of a subnode

```
⟨db⟩
   ⟨composer⟩
      ⟨name⟩ J.S. Bach ⟨/name⟩
      ⟨born⟩ 1685 ⟨/born⟩
      ⟨work  num="BWV82"⟩
         ⟨title⟩ Ich habe genug ⟨/title⟩
      ⟨/work⟩
      ⟨work  num="BWV552"⟩
      ⟨/work⟩
   ⟨/composer⟩
   ⟨composer  period="baroque"⟩
      ⟨name⟩ G.F. Handel ⟨/name⟩
      ⟨work  num="HWV19"⟩
         ⟨title⟩ Art Thou Troubled ⟨/title⟩
      ⟨/work⟩
   ⟨/composer⟩
⟨/db⟩
```

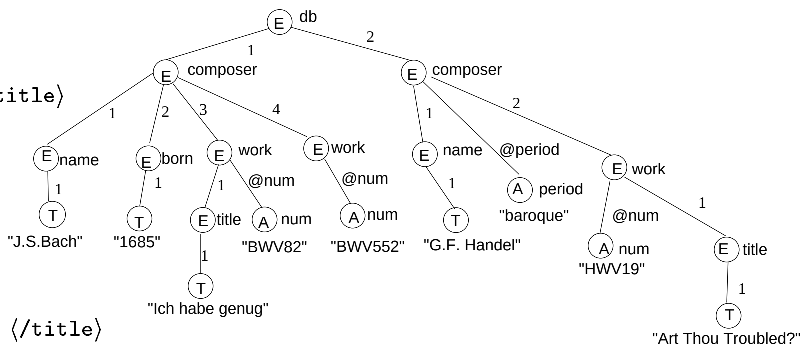

Figure 1: Some XML and its representation as a tree

relative to a node. For example any subnode of a node with address $\langle a \rangle$ will have a node address of the form $\langle a.b \rangle$ where $\langle b \rangle$ is the address of the subnode relative to $\langle a \rangle$. By a *subnode* of a node x we mean any node in the subtree rooted at x, not necessarily a child node of x.

Content equality. Equality is essential to the definition of keys, and in order to define keys we need first to define equality of the "values" associated with nodes. XML-Schema restricts equality to text nodes, but the authors have encountered cases in which keys are not so restricted. A more general way of describing equality is to use tree equality. The content of a node is specified by giving (1) a set S of relative addresses of its subnodes, (2) a partial function from S to names and (3) a partial function from S to strings. Two nodes are *content-equal* if they agree on (1), (2) and (3). With respect to the textual representation of an XML element, this definition states that the order of attributes is unimportant in defining equality. Observe that the order of subelements is specified and preserved by their indexes (integers).

Notation. We shall use $=_C$ for content equality.

It should be pointed out that neither equality of text nodes nor tree equality is entirely satisfactory in the presence of types. XML-Schema does a thorough job of defining base types, and one might want to use this to define a coarser form of equality. For example, `⟨id  type="int"⟩ 12 ⟨/id⟩` and `⟨id  type="int"⟩ 012 ⟨/id⟩` should probably be treated as content-equal. Also, there are types such as real numbers for which equality is problematic. A complete specification of keys would have to take account of these issues.

3 Path Expressions

A path expression is an expression involving node names (tags and attribute names) that describes a set of paths in the document tree.

The choice of what language we use to define path expressions is important to the expressive power of keys, and there are a number of choices. In XML-Schema, XPath [5] expressions are used, while in semistructured data regular expressions [1] are commonly used (neither subsumes the other). In the following analysis we shall assume two properties of path expressions:

- There should be a concatenation operation: $P.Q$ is the result of following first the path P and then the path Q.
- A path should move down the tree. That is if we start at a node n_1 and, by following a path described by P, we reach a node n_2 then n_2 is a subnode of n_1 (the address n_1 is a prefix of the address n_2.)

The second property is not enjoyed by XPath. We shall discuss the choice of a language of path expressions later, but in the meantime adopt for illustrative purposes a simple language that is certainly a subset of both XPath and regular expressions. Our language has the following constructs.

- The empty path, ϕ.
- A node name (a tag or attribute name).
- An arbitrary path $\_*$ (a combination of a "wild card" and Kleene star, though for the time being, neither of these alone is in the language).
- The concatenation of paths, $P.Q$, where P and Q are paths defined by these rules.

We shall use the notation $n[\![P]\!]$ to denote the set of nodes (node addresses) reached by starting at node n and following a path that conforms to (is in the language of) P. We shall sometimes use $[\![P]\!]$ as an abbreviation for $root[\![P]\!]$. The syntax is borrowed from Wadler's [8] description of semantics for patterns in XSL. Examples (from Figure 1):

$$\begin{array}{rcl}\langle 2.2\rangle[\![\,\mathsf{title}]\!] & = & \{\langle 2.2.1\rangle\}\\ \langle 2.2\rangle[\![\_*]\!] & = & \{\langle 2.2\rangle,\ \langle 2.2.1\rangle,\ \langle 2.2.1.1\rangle,\ \langle 2.2.\mathsf{@num}\rangle\ \}\\ [\![\mathsf{composer.\ work}]\!] & = & \{\langle 1.3\rangle,\ \langle 1.4\rangle,\ \langle 2.2\rangle\}\\ [\![\_*.\mathsf{num}]\!] & = & \{\ \langle 1.3.\mathsf{@num}\rangle,\ \langle 1.4.\mathsf{@num}\rangle,\ \langle 2.2.\mathsf{@num}\rangle\}\end{array}$$

4 Definition of Keys

In defining a key we specify two things: a set on which we are defining the key (in relational databases this is a relation – the set of tuples identified by a relation name) and the "attributes" (relational terminology for a set of column names) which together uniquely identify elements in the set. This is the motivation for our central definition of a *key specification*, which is a pair $(Q, \{P_1, \ldots, P_n\})$ where Q is a path expression and $\{P_1, \ldots, P_n\}$ is set of path expressions. The idea is that the path expression Q identifies a set of nodes, which we refer to as the *target set*, on which the key constraint is to hold. Let us refer to Q as the *target path*. The set $\{P_1, \ldots, P_n\}$, which we shall call the *key paths*, constrains the target set as follows: Take any two nodes $(n_1, n_2) \in [\![Q]\!]$ and consider the pairs of nodes found by following a key path P_i from n_1 and n_2. If all such pairs of nodes are content-equal, then the nodes n_1 and n_2 are the same node. In order for this to make sense we need to assume that following a key path gives us a single node, e.g., $n_1[\![P_i]\!]$ contains one node.

Before writing down the formal definition, consider an example:

$$(\mathsf{person.employees}, \{\mathsf{name.firstname},\ \mathsf{name.lastname}\})$$

The target path `person.employees` identifies a set of nodes in the document (there may be several `employees` and `person` nodes). This is the target set. Each of these nodes will define a subtree with an `employees` label at the root. Within a subtree we need, first of all, the key paths `name.firstname` and `name.lastname` to be unique. Second, we require that two distinct nodes in the target set differ on at least one of the paths `name.firstname` and `name.lastname`. Here we mean that the nodes at the ends of these paths are not content-equal (obviously they are distinct nodes).

Definition. A path expression P *is unique at* n if $|n[\![P]\!]| = 1$.

For example (referring to Figure 1), name is unique at $\langle 1 \rangle$, but work and num are not unique at this node.

We now give the formal definition of a key. In our example we assumed that the target path started at the root, however for reasons shortly to emerge, it is useful to define a key with respect to a given node in the document.

Definition. A node n *satisfies* a key specification $(Q, \{P_1, \dots, P_k\})$ if

- For all n' in $n[\![Q]\!]$ and for all $P_i (1 \leq i \leq k)$, P_i is unique at n'.
- For any n_1, n_2 in $n[\![Q]\!]$, if $n_1[\![P_i]\!] =_C n_2[\![P_i]\!] (1 \leq i \leq k)$ then $n_1 = n_2$.

Note that both forms of equality are used in the definition of a key. Here are some further examples. In these, we assume that the key acts at the root, i.e., the root node satisfies the key.

$(\_*.\mathsf{person}, \{\mathsf{id}\})$	Any two person elements, no matter where they occur, have unique id subelements and differ on those elements.
$(\mathsf{person}, \{\phi\})$	Any two person nodes immediately under the root have different values (ϕ is the empty path).
$(\mathsf{employees}, \{\})$	An empty key. This means that the path employees, if it exists, is unique at the root. I.e., there is at most one employees node immediately under the root.
$(\_*, \{k\})$	At first sight this appears to require that every element have a key k. However, any element whose name is k must also have a key, and this calls for some form of "infinite" document. One might get round this with a more complex (grep-like) path expression such as $([\hat{\ }k]*, \{k\})$. In Section 7 we describe a weaker definition of keys in which $(\_*, \{k\})$ is satisfiable by a finite document.

Comparing this definition with the relational definition of keys [2, 6], the first part of this constrains the paths that define keys to exist and to be unique. In relational databases key values cannot be null (the key must exist) and first normal form requires attribute values to be individuals, not sets. The second part states that a key specifies a unique *address* within a document (unlike the relational case, it does not specify a unique value). When we talk about document satisfying a key specification we mean that the root of the document satisfies the key specification. There are, of course, other ways of defining keys, both more and less restrictive than what we have described. Some justification of the choices is in order.

- We have used a *set* of key paths to define a key. In order to talk about a set (as opposed to a tuple or list) of path expressions we need to be able to talk about equality of path expressions. The equivalence of two path expressions in our language of path expressions is decidable, as it is for the more general class of regular expressions.

- Given that we have defined equality on trees, do we need to have more than one key path in a key specification? We could always design our documents so that all the key "attributes" are represented as subnodes of some node. The problem here is that we would have to constrain the node to contain only these subnodes for tree equality to have the desired effect. This seems to be too restrictive and constitutes unnecessary interference between key specifications and data models.

- The first part of the definition of key satisfaction requires each of the paths to exist and to be unique from any node in $n[\![Q]\!]$. We shall examine the possibility of dropping this condition in Section 7.

- The language of path expressions may be regarded both as too weak and too powerful. Consider the key $(Q, \{P_1, \ldots, P_k\})$, would one ever want an arbitrary path $(\_*)$ in one of the P_i? Also, it is not hard to come up with examples in which one would like something more powerful to express Q, e.g., $(\mathsf{person}.(\mathsf{mother} \mid \mathsf{father})*, \{\mathsf{id}\})$. We stress that the language of path expressions is provisional.

As in relational databases we can infer some keys from the presence of others. There is nothing that directly corresponds to the notion of a super key [6]. For example, if a node n satisfies the key (Q, S) and $S \subseteq S'$, we *cannot* conclude that it satisfies the key (Q, S'). The reason is that the uniqueness condition is not guaranteed to hold. However the following inferences are sound:

Fact. If $(Q, \{P_1, \ldots, P_i, \ldots, P_k\})$ and $(Q.P_i, \{P_{i,1}, \ldots, P_{i,j_i}\})$ are keys, so is

$$(Q, \{P_1, \ldots, P_i.P_{i,1}, \ldots, P_i.P_{i,j_i}, \ldots, P_k\}).$$

Fact. If (Q, S_1) and (Q, S_2) are keys, so is $(Q, S_1 \cup S_2)$.

5 Relative Keys

There are many situations in which a key provides only a relative specification. For example, a verse number is unique only within a chapter. In relational database design, the key of a weak entity is made up of the key of the "parent" entity and some additional identification [6], e.g. course Math120, section B. To describe this we need the notion of a *relative* key, which consists of a pair (Q, K) where Q is a path expression and K is a key.

Definition. A document satisfies a relative key specification (Q, (Q',S)) iff for all nodes n in $[\![Q]\!]$, n satisfies the key (Q', S).

In other words (Q, K) is a relative key if K is a key for every "sub-document" rooted at a node in $[\![Q]\!]$. Examples:

(bible.book.chapter, (verse, {number}))	A verse number uniquely identifies a verse within a chapter.
(bible.book, (chapter, {number}))	Chapter numbers uniquely identify a chapter within a book.
(bible, (book, {name}))	If there is only one bible node immediately under the root, this is the same as specifying an absolute key (bible.book, {name}).

Observe that in a relative key $(Q, (Q', S))$, Q is from the root whereas Q' starts at a node in $[\![Q]\!]$. It is for this reason that we defined key satisfaction at arbitrary nodes.

Transitivity of relative keys. The purpose of keys is to specify uniquely certain components of a document. Obviously, a relative key such as (bible.book.chapter, (verse, {number})) alone does not uniquely identify a particular verse in the bible, however we believe that if we give a book name, a chapter number, and a verse number, we have specified a verse. It is this intuition that we need to formalize.

First observe that the relative key $(\phi, (Q', S))$ is equivalent to the key (Q', S). Now consider two relative keys. We say that $(Q_1, (Q'_1, S_1))$ *immediately precedes* $(Q_2, (Q'_2, S_2))$ if $Q_2 = Q_1.Q'_1$. Any relative key immediately precedes itself. Define the *precedes* relation as the transitive closure of the immediately precedes relation.

Definition. A set Σ of relative keys is *transitive* if for any relative key $(Q_1, (Q'_1, S_1)) \in \Sigma$ there is a key $(\phi, (Q'_2, S_2)) \in \Sigma$ which precedes $(Q_1, (Q'_1, S_1))$.

As an example, this set of keys is transitive:

$(\phi$, (bible.book, {name}))
(bible.book, (chapter, {number}))

This set is not:

$(\phi$, (bible.book, {name}))
(bible.book.chapter, (verse, {number}))

Observe that keys are special cases of relative keys since they are expressible in the form of $(\phi, (Q, \{P_1, \ldots, P_k\}))$. Any transitive set of relative keys must contain some key.

Completeness of relative keys. Consider the following (transitive) key specification:

$(\phi$, (university, {name}))
(university, (dept, {dept-name}))
(university, (dept.employee, {emp-id}))

To identify an `employee` node we need only to specify a university name and an `emp-id` within that university. Even though the employees of interest are in departments and departments are specified by `dept-name`, we do not need a `dept-name` to identify an employee. There is something anomalous here if one considers the addition of a new employee to the database. In order to do this one needs to specify the department in which the employee is to live, even though one does not need this knowledge in order to identify that employee. This “smells” of the anomalies that occur in relational databases that are not in second normal form. There is something wrong with the design of the document in that employees should not be children of department nodes, but only of university nodes. The linkage between employees and departments should be expressed through a foreign key.

This motivates our final definition of *completeness* as shown below. One can always insert an element in an XML document, that satisfies a complete key specification, by specifying the keys at every level.

Definition. A set Σ of relative keys is *complete* if it is transitive and whenever $(Q1, (Q_2.n, S_1)) \in \Sigma$ there is a relative key $(Q1, (Q2, S_2)) \in \Sigma$. Here n is a node name.

6 A notation for relative keys

If a system of relative keys is transitive, it forms a hierarchical structure. We can therefore create a compressed syntax for such systems. Again, we stress that the syntax is “abstract”; a concrete syntax may be based on that of regular expressions or XPath.

The basic syntactic form is

$$Q_1\{P^1_1, \ldots, P^1_{k_1}\}.Q_2\{P^2_1, \ldots, P^2_{k_2}\}. \ \ldots \ .Q_n\{P^n_1, \ldots, P^n_{k_n}\}$$

This describes a complete system of relative keys. For each i, $1 \leq i \leq n$, it defines the relative key $(Q_1. \ \ldots \ .Q_{i-1}, (Q_i, \{P^i_1, \ldots, P^i_{k_i}\}))$. It should be noted that the first of these is of the form $(\phi, (Q_1, \{P^1_1, \ldots, P^1_{k_1}\}))$ and is a key.

For example bible{}.book{name}.chapter{number}.verse{number} specifies the complete system of keys:

$(\phi$, (bible, {}))
(bible, (book, {name}))
(bible.book, (chapter, {number}))
(bible.book.chapter, (verse, {number}))

So far the key hierarchies we have specified are linear. Consider the following two specifications: company{name}.employee{id} and company{name}.department{name}. It is helpful to fold these into a single specification:

company{name}[.employee{id}, .department{name}]

This is simply a syntactic shorthand: $R[R_1, \ldots, R_n]$ for RR_1, …, RR_n.

As a further example, consider

university{name}.school[{name}, .department[{name}, .student{id}]]

This is an example of a transitive but incomplete set of relative keys. Here, students are uniquely specified by an id within a university (not just within a department or school)

Specifications such as these are reasonably compact and understandable. Their importance is not only to ensure the internal consistency of a document, but also to tell others how to cite a component of our document. This is especially important if the document is subject to change. Even though we have constructed a minimal system for describing hierarchical key structures, it turns out that this takes us some way towards describing a data model. Contrast relational database specification student(snum, name, major) and enroll(snum,cnum,grade) with a key specification

[student{snum}[.name{}, .major{}], enroll{snum,cnum}.grade{}]

They describe closely related structures. The specification [name{}, major{}] ensures that under a student node there is at most one name and at most one major node. However the key specification allows other unspecified nodes to occur under a student node and, of course, it does not require any kind of first normal form. Nevertheless, we can specify that our documents have a structured "core" somewhat akin to the complex object or nested relational structures that have been studied in databases [2]. Not surprisingly there is close interaction between key constraints and data models which requires much further study.

7 Discussion

Our main reason for writing this document was to clarify the notion of a relative key and to understand the hierarchical key structure that appears to occur naturally in a variety of data formats. What we have described here is a proposal for a key definition, and there are a number of variations on this definition which should be considered. This section contains a brief review of those alternatives, starting with the proposals in XML-Schema.

7.1 XML-Schema

XML-Schema includes a syntax for specifying keys which is related to our definition, but there are some substantive differences, even if we ignore the issue of relative keys. Possibly the most important of these is that the language for path expressions is XPath. XPath is a relatively complex language in which one can not only move down the document tree, but also sideways or upwards, not to mention that predicates can be embedded as well. The main problem with XPath is that questions about equivalence or inclusion of XPath expressions are, as far as the authors are aware, unresolved; and these issues are important if we want to reason about keys as we do – for quite practical purposes – in relational databases. Here is a brief summary of the other salient differences between our definitions and the XML-Schema proposal.

Equality. We have used a more general form of equality than that in XML-Schema. However, as pointed out in Section 2 a full treatment of equality might involve types or even some form of user-defined equality.

Definition of the target set. In XML-schema the path expression that defines the target set is taken to start at arbitrary nodes. Thus if Q defined the target set in XML-schema, this is equivalent to $\_*.Q$ in our notation, starting from the root. Incidentally, if one wants to "root" the path expression in XML-Schema, this can be done because XPath contains an operation that means "move to the root".

Definition of key paths. XML-Schema talks about a *list* (not a set) of key paths. While this avoids issues of equivalence of XPath expressions, one can construct keys that are, presumably, equivalent, but have different or anomalous presentations. For example (temporarily using [...] for lists): (person,[firstname, lastname]), (person,[lastname, firstname]), (person,[lastname, lastname, firstname]) impose the same constraint. Yet until we know how to determine the equivalence of XPath expressions, there is no general method of saying whether two such specifications are equivalent.

Relative keys. While there is no direct notion of a relative key in XML-Schema, in certain circumstances one can achieve a related effect. Consider for example:

$$\mathsf{book}\{\mathsf{name}\}.\mathsf{chapter}\{\mathsf{name}\}.\mathsf{verse}\{\mathsf{number}\}.$$

In XML-Schema one can specify a key for verse as

$$(\mathsf{book.chapter.verse},\ [\mathsf{number},\ \mathit{up}.\mathsf{name},\ \mathit{up}.\mathit{up}.\mathsf{name}]).$$

Here *up* (this is not XPath syntax) is the XPath instruction to move up one node. Thus part of the key is outside of the contents of a verse node. One of the inferences one could make for such a specification is that (book.chapter, [name, *up*.name]) is a key *provided* the nodes in the target set all contain at least one verse child node. Again, it is not clear how to reason generally about such specifications.

7.2 Choice of a path expression language

We have used a language for path expressions that contains just enough to illustrate most of the issues that occur in connection with keys for XML. In order to reason about keys, it is essential that equivalence and inclusion of path expressions are decidable, but this is the case for the more expressive language of regular expressions, and we could equally well have used this language; none of the results would be affected. However the examples we found that used the added expressive power were somewhat contrived, and it is not clear whether this larger language is of practical use.

An interesting issue is whether, in defining a key $(Q, \{P_1, \ldots, P_n\})$, the language used to describe the target path Q needs to be the same as the language used to define the key paths $P_1, \ldots, P_n$. For example, there seems to be little use for the arbitrary path $\_*$ in key paths. One could choose a *simpler* language for key paths that is a sublanguage of the language for target paths. In fact, we only require that the composition $Q.P_i$ of a target path and a key path should be in the language of target paths. While the current proposal allows us to express some mildly nonsensical key paths, we could see no benefit to simplifying the language of key paths.

7.3 A weaker definition of keys

Given a key constraint, it is natural to ask whether it is finitely satisfiable, i.e., whether there exists a finite XML document satisfying the key. In relational databases, all keys are finitely satisfiable: given any schema S and any finite set Σ of keys, one can always construct a finite database instance of S that satisfies Σ. In XML, however, some keys are not finitely satisfiable. Indeed, as mentioned earlier, $(\_*, \{id\})$ imposes an infinite chain of id nodes and therefore, there is no finite document satisfying the key. This is because in our key specification, we require that key paths must exist. It should be mentioned that the corresponding key in XML-Schema, $(//*, [id])$, is not meaningful either, because an id node cannot have a base type if it is to have an id subelement itself.

To ensure finite satisfiability of keys, we give a weaker notion of key specification. As before we specify a key in terms of a pair $(Q, \{P_1, \ldots, P_k\})$, where Q is the target path and $P_1, \ldots, P_k$ are the key paths. But we do not require the key paths to exist. More specifically, for any node n in $[\![Q]\!]$, there is no restriction on $n[\![P_i]\!]$: it can be the empty set, a singleton set, or a set having multiple elements. The key specification only requires that if two nodes in $[\![Q]\!]$ are distinct, then the two sets of nodes reached on some P_i must be disjoint up to content-equality. More specifically, for any distinct nodes n_1, n_2 in $[\![Q]\!]$, there must exist some P_i, $1 \le i \le k$, such that for all x in $n_1[\![P_i]\!]$ and y in $n_2[\![P_i]\!]$, $x \ne_C y$. Formally, we state the semantics of this weaker definition of keys as follows:

Definition. A node n *satisfies* a key specification $(Q, \{P_1, \ldots, P_k\})$ iff for any n_1, n_2 in $n[\![Q]\!]$, if for all i, $1 \le i \le k$, there exist a path $p \in P_i$ (p is in the language defined by P_i) and nodes $x \in n_1[\![P_i]\!]$ and $y \in n_2[\![P_i]\!]$ such that $x =_C y$, then $n_1 = n_2$. That is,

$$\forall n_1\, n_2 \in n[\![Q]\!] \;\; ((\bigwedge_{1 \le i \le k} \exists p \in P_i \;\exists\, x \in n_1[\![p]\!] \;\exists\, y \in n_2[\![p]\!] \;(x =_C y)) \;\rightarrow\; n_1 = n_2).$$

Along the same lines of keys in relational databases, this weak definition of keys asserts that the values associated with key paths uniquely identify a node in the target set. It generalizes relational keys to accommodate key paths that lead to multiple nodes, since one cannot require XML documents to be in some kind of first normal form.

As before, we assume that the target path Q starts at the root and by saying that a document satisfies a key, we mean that the root of the document satisfies the key. For a node n in $[\![Q]\!]$, we say that P_i is *missing at* n if $n[\![P_i]\!]$ is empty. Observe that for any n_1, n_2 in $[\![Q]\!]$, if P_i is missing at either n_1 or n_2, i.e., if either $n_1[\![P_i]\!]$ or $n_2[\![P_i]\!]$ is empty, then $n_1[\![P_i]\!]$ and $n_2[\![P_i]\!]$ are disjoint. Thus the key has no impact on those nodes at which some key paths are missing. This is similar to *unique constraints* introduced in XML-Schema. In contrast to unique constraints, this notion of key specification is capable of comparing nodes at which a key path may lead to multiple nodes. As an example, consider a key $(A, \{B\})$ on a document:

```
⟨db⟩
      ⟨A⟩  ⟨B⟩ 1 ⟨/B⟩  ⟨/A⟩
      ⟨A⟩  ⟨B⟩ 1 ⟨/B⟩  ⟨B⟩ 2 ⟨/B⟩  ⟨/A⟩
⟨/db⟩
```

This key asserts that an A element is uniquely identified by the values of its B subelements. In other words, if two A elements agree on the values of some of their B subelements, then the two nodes must be identical. The document does not satisfy the key because the B subelement in the first A element and the first B subelement of the second A element have the same value. Thus with our weak notion of keys we can distinguish these two A elements. In contrast, key and unique constraints of XML-Schema cannot compare these A elements because the second A element has more than one B subelement.

Recall the strong notion of key specification introduced earlier. The strong notion and weak notion impose different restrictions on key paths. At one end of the spectrum, the strong notion requires that all key paths must exist and be unique. At the other end, the weak notion imposes no structural constraints on key paths.

Given this weaker notion of keys, let us re-examine some examples given above.

$(\_*.\mathsf{person}, \{\mathsf{id}\})$	Any person element, if it has id subelements, is uniquely identified by the values of the id's. In other words, any two person elements are disjoint on their id fields up to content-equality.
$(\mathsf{person}, \{\phi\})$	The interpretation of this key remains unchanged given the weaker notion of key specification.
$(\mathsf{employees}, \{\})$	Again, the semantics of this key is the same with respect to the strong and weak key specifications.
$(\_*, \{id\})$	Any element that has id subelements is uniquely identified by the values of the id's. That is, any two nodes are disjoint on their id fields up to content-equality. In contrast to the strong notion of key specification, here an id element does not have to have an id itself, and as a result, there are finite documents satisfying the key. In fact, this key constraint captures the semantics of an ID attribute in the XML standard.

The weaker notion of key specification has the following property that is enjoyed by keys in relational databases.

Fact. For any finite set Σ of keys, there exists a finite XML document satisfying Σ.

In addition, as in relational databases, the inference rule for super keys is also sound here.

Fact. If $(Q,\, S)$ and $S \subseteq S'$, then $(Q,\, S')$.

However, in contrast to relational databases, observe that given $(Q, \{P_1, \ldots, P_k\})$, not all nodes in $[\![Q]\!]$ can be identified by $P_1, \ldots, P_k$ since some of the key paths may be missing. This might be taken as allowing *null-valued* keys, but whether we should equate missing key paths with null values is arguable and depends on the semantics of the languages we use to query XML documents.

7.4 Node names as key values

The choice of an appropriate definition for keys for XML will ultimately be determined by practice. The aim of setting out a key specification is to cover the practical cases without using definitions that are too complex to allow any kind of reasoning about keys. Have the proposals in this paper covered the practical cases?

There is one issue that may arise in "unconstrained" XML. Consider the database

```
⟨parts⟩
   ⟨widget⟩
      ⟨id⟩ 123 ⟨/id⟩⟨weight⟩ 1.5 ⟨/weight⟩
   ⟨/widget⟩
   ⟨widget⟩
      ⟨id⟩ 234 ⟨/id⟩⟨weight⟩ 2.5 ⟨/weight⟩
   ⟨/widget⟩
   ⟨gadget⟩
      ⟨id⟩ 123 ⟨/id⟩⟨weight⟩ 3.2 ⟨/weight⟩
   ⟨/gadget⟩
⟨/parts⟩
```

The type of a part – `widget` or `gadget` – is expressed in the tag. In alternative XML representations it might be expressed as an attribute or subelement of a `part` element. They key for a part is to be taken as its type together with its `id`. With our current machinery, the key constraint can be expressed as `parts{}[.widget{id}, .gadget{id}]`. However, if we introduce a new part type, a `thingy`, the key specification will have to be changed to include a key path involving `thingy`. No change would be needed in the alternative representations. The problem arises because we are interchanging structure (the names) with data (their contents); but the ability to do this is supposed to be one of the strong points of semistructured data and XML.

Our definition of a key (weak or strong) can be extended to express this by two simple changes:

- The addition of a wild-card _, which matches *single* edges, to the language of path expressions. This change is only needed to the language of path expressions for target paths.
- the addition of a "virtual" subelement, `node-name` to each named node whose contents consist of the node name.

With these extensions, the key for our example can be expressed as `parts{}._{node-name, id}`.

This does not alter any of the properties we expect to hold for keys and appears to account for any practical use of tag names in keys.

Acknowledgements. We are grateful to Byron Choi, Hartmut Liefke, Arnaud Sahuguet and Keishi Tajima for a number of useful discussions.

References

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufman, 2000.

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

[3] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, and Lauren Wood. *Document Object Model (DOM) Level 1 Specification.* W3C Recommendation, October 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[4] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0.* World Wide Web Consortium (W3C), Feb 1998. `http://www.w3.org/TR/REC-xml`.

[5] James Clark and Steve DeRose. *XML Path Language (XPath).* W3C Working Draft, November 1999. `http://www.w3.org/TR/xpath`.

[6] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill Higher Education, 2000.

[7] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures.* W3C Working Draft, April 2000. `http://www.w3.org/TR/xmlschema-1/`.

[8] Philip Wadler. A Formal Semantics for Patterns in XSL. Technical report, Computing Sciences Research Center, Bell Labs, Lucent Technologies, 2000. `http://www.cs.bell-labs.com/~wadler/topics/xml#xsl-semantics`.