Keys for XML

Peter Buneman* Susan Davidson* Wenfei Fan† Carmem Hara‡ Wang-Chiew Tan*

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 and scientific databases.

Keywords: Keys, Relative Keys.

1. INTRODUCTION

Keys are an essential part of database design [2, 14]: 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.

*University of Pennsylvania. Email: {peter,susan,wctan}@saul.cis.upenn.edu. Supported in part by Digital Libraries 2 grant DL-2 IIS 98-17444 and NSF DBI99-75206.

†Temple University. Email: fan@joda.cis.temple.edu. Supported by RIF fund.

‡Universidade Federal do Parana, Brazil. Email: carmem@inf.ufpr.br

[1]We used the "DTD Inquisitor" of Byron Choi and Arnaud Sahuguet [15, 10].

WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

Various forms of key specification for XML are to be found in the XML standard [7], XML Data [13], XML Schema [17]. Through the use of ID attributes in a DTD [7], one can uniquely identify an element within an XML document. However, it is not clear that ID attributes are intended to be used as keys rather than internal "pointers". For example, ID attributes are not scoped. In contrast to keys, they are unique within the entire document rather than among a designated set of elements. As a result, one cannot, for example, allow a student (element) and a person (element) to use the same SSN as an ID. Moreover using ID attributes as keys means that we are limiting ourselves to unary keys and, of course, to using attributes rather than elements. Finally, one can specify at most one ID attribute for an element type, while in practice one may want more than one key. XML Data introduces a notion of keys explicitly. However, its keys can only be specified in types and moreover, can only be defined for element types rather than for certain collections of elements.

XML Schema has a more elaborate proposal, which is the starting point of this paper. The proposal extends the key specification of XML Data by allowing one to specify keys in terms of XPath [11] expressions. There are a number of technical problems in connection with 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 and functions 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 in relational databases. 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. Another technical issue is value equality. XML Schema restricts equality to text, but the authors have encountered cases in which keys are not so restricted. A more detailed discussion can be found in section 7.1.

However, the main reason for writing this paper is that none of the existing key proposals address the issue of hierarchical keys, which appear to be ubiquitous in hierarchically structured databases, especially in scientific data formats. A top-level key may be used to identify components of a document, and within each component a secondary key is used to identify sub-components, and so on. Moreover, 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.

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 following 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, therefore the way we might constrain the XML document is 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.

Organization. The rest of the paper is organized as follows. Section 2 introduces the notion of node addresses and value equality. Node addresses are used in node equality testing, i.e., testing whether two nodes are the same node and value equality is used for testing whether two nodes have the same value. Section 3 introduces our path expression language which is used in the definition of keys discussed in section 4. Section 5 addresses issues in connection with reasoning about XML keys. The concept of relative or hierarchical keys together with its alternative notation is discussed in section 6. In section 7, we examine the XML-Schema proposal in some detail, discuss an alternative form of keys and various issues concerning keys.

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) have both 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 $\langle$`1#2#1`$\rangle$ and $\langle$`1#3#@num`$\rangle$. 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 *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.

Value 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. An immediate example is that when one treats `name` as a key for `person` nodes, `name` may have a complex structure consisting of `first-name` and `last-name` subelements. A more general way of describing equality is to use tree equality. The value 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 *value-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 $=_v$ for value equality.

It should be pointed out that neither equality of text nodes

⟨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⟩

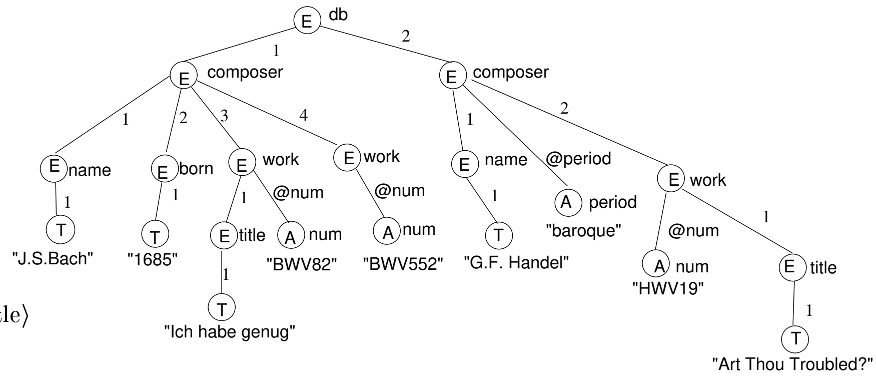

Figure 1: Some XML and its representation as a tree

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"⟩ 0 ⟨/id⟩` and `⟨id type="int"⟩ -0 ⟨/id⟩` should probably be treated as value-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 [11] expressions are used, while in semistructured data regular expressions [1] have been 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 for path expressions has the following syntax:

- The empty path, "ϵ".
- A node name (a tag or attribute name).
- A wild card "_", matching any single node name.
- An arbitrary path "_$*$".
- The concatenation of paths $P.Q$, where P and Q are paths defined by these rules.

We have chosen an alternative syntax to that of XPath because the concatenation operation, which is central to our understanding of keys, does not have a uniform representation in XPath. However, the translation to XPath is straightforward: Any path meant to start from the root is prefixed with "/". In XPath, "/" itself denotes the root node. "." is used as the empty path in place of "ϵ", "*" in place of "_" and "//" in place of "_*". Also, "/" is used as the path concatenator in place of ".". In XPath, "/" is used as a separator between location steps. Therefore, we have to disallow certain concatenations. If for example we concatenate a/b with $/c/d$ we get $a/b//c/d$ with an entirely different meaning.

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 [18] description of semantics for patterns in XSL. Examples (from Figure 1):

$$\begin{array}{rcl}
\langle 2\#2\rangle[\![\mathsf{title}]\!] & = & \{\langle 2\#2\#1\rangle\} \\
[\![\mathsf{composer}._]\!] & = & \{\langle 1\#1\rangle,\ \langle 1\#2\rangle,\ \langle 1\#3\rangle,\ \langle 1\#4\rangle, \\
 & & \langle 2\#1\rangle,\ \langle 2\#2\rangle\} \\
\langle 2\#2\rangle[\![_*]\!] & = & \{\langle 2\#2\rangle,\ \langle 2\#2\#1\rangle,\ \langle 2\#2\#1\#1\rangle, \\
 & & \langle 2\#2\#\mathsf{@num}\rangle\} \\
[\![\mathsf{composer.work}]\!] & = & \{\langle 1\#3\rangle,\ \langle 1\#4\rangle,\ \langle 2\#2\rangle\} \\
[\![_*.\mathsf{num}]\!] & = & \{\langle 1\#3\#\mathsf{@num}\rangle,\ \langle 1\#4\#\mathsf{@num}\rangle, \\
 & & \langle 2\#2\#\mathsf{@num}\rangle\}
\end{array}$$

In some cases, it will be useful to restrict the path expression language so that paths are merely sequences of labels and do not contain _ or _$*$. Such paths are called *simple paths*. For example, composer.work is a simple path.

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 a set of simple 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*, and the set $\{P_1, \ldots, P_n\}$ as the *key paths.* These correspond to the absolute and relative location paths respectively in XPath terminology. Observe that for any node $n \in [\![Q]\!]$ there is a set of nodes $n[\![P_i]\!]$ found by following P_i from n. There is no restriction on the size of $n[\![P_i]\!]$; in particular it may be empty. The key paths constrain the target set as follows: Take any two nodes $(n_1, n_2) \in [\![Q]\!]$ and consider the pair of sets of nodes found by following the key path P_i from n_1 and n_2, $(n_1[\![P_i]\!], n_2[\![P_i]\!])$. If there is a non-empty intersection with respect to value equality for all such pairs of sets of nodes then the nodes n_1 and n_2 are the same node. For example, consider the following key definition:

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

The target path person.employees identifies a set of nodes in the document. This is the target set. Each of these nodes will define a subtree with an employees label at the root. Within such a subtree we will find zero or more key paths name.firstname and zero or more key paths name.lastname. Two nodes n_1, n_2 in the target set are distinct if either they do not agree on any of the nodes reachable via key path name.firstname or they do not agree on any of the nodes reachable via name.lastname.

As another example, observe that the document in Figure 1 satisfies the key (composer, {name}): There are two nodes at the end of the target path composer. For each node, there is one element in the set of nodes found by following the key path name, "J.S.Bach" and "G.F.Handel". These elements are not value-equal. Less intuitively, the document also satisfies the key (composer, {born}) since the subelement <born> only appears in the first composer and is absent from the second composer.

We are now ready to give the formal definition of a key. For reasons which will emerge shortly, it is useful to define a key with respect to a given node in the document rather than assuming that the target path starts at the root.

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 $z_1 \in n_1[\![P_i]\!]$ and $z_2 \in n_2[\![P_i]\!]$ such that $z_1 =_v z_2$, then $n_1 = n_2$. That is,

$$\forall n_1, n_2 \in n[\![Q]\!] \\ \left(\left(\bigwedge_{1 \le i \le k} \exists z_1 \in n_1[\![P_i]\!]\ \exists z_2 \in n_2[\![P_i]\!]\ (z_1 =_v z_2) \right) \rightarrow n_1 = n_2 \right)$$

Note that both forms of equality are used in the definition of a key. The first deals with value-equality ($=_v$) while the second is node equality ($=$). Two nodes are node equal if they have the same node address.

When we talk about document satisfying a key specification we mean that the root of the document satisfies the key specification. The key has no impact on those nodes at which some key path is *missing*, i.e. nodes n such that $n[\![P_i]\!]$ is empty for some P_i. Observe that for any n_1, n_2 in $[\![Q]\!]$, 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 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}) expressed with respect to the root of the following 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. 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. And with our definition of keys, these two A elements are required to be the same element.

Here are some further examples of keys, expressed with respect to the root of a document.

(_∗ .person, {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 value-equality.
(person, {ϵ})	Any two person nodes immediately under the root have different values (ϵ is the empty path).
(employees, {})	An empty key. This means that the path employees, if it exists, is unique at the root. That is, there is at most one employees node immediately under the root.
(_∗, {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 value-equality. Note that an id element does not have to have an id itself. This key captures the semantics of an ID attribute in the XML standard in that id is unique within the entire document.

As with keys in relational databases, this definition of a key asserts that the values associated with key paths uniquely identify a node in the target set. However since one cannot require XML documents to be in some kind of first normal form, there are important differences between the two definitions. First, the paths that define keys need not exist [2]

[2]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.

and do not have to be unique. In contrast, in relational databases since key values cannot be null, the key must exist. Moreover, first normal form requires attribute values to be atomic values, not sets. Second, our key paths specify a set of addresses within a document, unlike the relational case in which keys specify a value.

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 definition of key satisfaction differs significantly from the relational case by allowing a (possibly empty) set of nodes at the end of each key path. We shall examine a more restrictive definition in which key satisfaction requires each of the key paths to exist and to be unique from any node in $n[\![Q]\!]$ in Section 7.

- The language of path expressions may be regarded both as too weak and too powerful. Consider the key $(Q, \{P_1, \ldots, P_k\})$: For now, we have allowed Q to be an arbitrary path expression but have restricted the P_i to be simple paths. 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., $(\mathsf{person}.(\mathsf{mother} \mid \mathsf{father})*, \{\mathsf{id}\})$. This means a person element followed by zero or more father or mother elements. Our emphasis is that the language of path expressions is provisional, and that allowing arbitrary path expression for the P_i merely complicates the definition of key but does not change much in the way of the theory.

5. KEY INFERENCE

In relational databases one can infer some keys from the presence of others. Indeed, if a set S of attributes is a key for a relation R, then any superset of S is also a key for R. This obvious fact is of great importance in query optimization. Keys are typically used as physical indexes, and this simple inference rule tells us when we have enough information to use such an index. For XML keys as we have presented them so far, the inference rules are far from obvious. These rules are fully discussed in a companion paper [8]. Here are some examples.

Fact. If $(Q,\ S)$ is a key and $S \subseteq S'$, then so is $(Q,\ S')$.

This is the counterpart of the relational inference rule. Below are two examples that have no such counterpart.

Fact. If $(Q.Q', \{P\})$ is a key then so is $(Q, \{Q'.P\})$.

This is sound because in a document with a tree-like structure, sharing of nodes is not allowed. As a result, if a node is identified in a tree then its ancestors are also determined. In other words, if a key path P uniquely identifies a node n in $[\![Q.Q']\!]$ then $Q'.P$ is a key path for the ancestor of n in $[\![Q]\!]$.

Fact. If (Q, S) is a key and Q' is contained in Q (i.e., the path language defined by Q' is included in the one defined by Q), then (Q', S) is also a key.

This fact is sound because any key of the set $[\![Q]\!]$ is also a key for any subset of $[\![Q]\!]$. Observe that $[\![Q']\!]$ is a subset of $[\![Q]\!]$ if Q' is contained in Q.

The last fact requires one to reason about the inclusion of path expressions.

Key inference is closely related to the question of key implication: suppose it is known that an XML document satisfies certain keys, does it follow that the document must necessarily satisfy some other key? We have developed algorithms for reasoning about the inclusion of certain classes of path expressions as well as for determining implication of XML keys. A detailed discussion of these algorithms as well as finite axiomatization and complexity results in connection with our key languages can be found in [8].

Another natural question to ask is whether key constraints are finitely satisfiable. 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. The same holds for XML documents under our definition of a key.

Fact. For any finite set Σ of keys, there exists an (finite) XML document satisfying Σ.

This last fact only holds because key paths may be missing. Recall the $(\_*, id)$ example: if key paths were required to exist at all nodes specified by the target path the XML document would have to be infinite to satisfy the key (see strong keys in section 7.)

Also, we note that the last fact only holds in the absence of DTDs. To illustrate this, let us consider a simple key

$$\varphi = (X, \{\ \})$$

and a simple DTD D:

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

Obviously, there exists a finite XML document that conforms to the DTD D (see, e.g., Fig. 2 (a)), and there is a finite XML document that satisfies the key φ (e.g., Fig. 2 (b)). However, there is no XML document that both conforms to D and satisfies φ. This is because D requires an XML tree to have two distinct X elements, whereas φ requires that there is at most one X node immediately under the root. This shows that DTDs interact with XML key constraints. It should be mentioned that keys defined in other proposals for XML, such as those introduced in XML Schema [17], also interact with DTDs or other type systems for XML. For a study of the interaction between constraints such as keys and DTDs see [12].

6. RELATIVE KEYS

The need for relative keys is partly motivated by scientific data formats. Many scientific databases do not use conventional database technology, and even those that do transmit their data in one of a variety of data formats. Some of these data formats are general purpose (such as ASN.1, used in GenBank [6], and ACeDB [16]) while others are domain specific (such as EMBL [4]). These data formats have easy translations to XML. XML itself is also emerging as a standard for data exchange, especially with micro-array data (see for example the DTDs GEML [20] and MAML [21]). All of these specifications have a hierarchical structure, and typically at the top level consist of a large set of entries (the order of which is usually unimportant). Molecular biology databases contain particularly rich structures of metadata. In the protein sequence database Swissprot [5] there is an accession number (a key) for each entry. Within each entry there is a sequence of citations, each of which is identified by a number 1,2,3... within the entry. Thus to identify a citation fully, we need to provide both an accession number for the entry and the number of the citation within the entry.

Another intriguing example is to be found in linguistic databases[3]. In this case the data sets (typically recordings of speech) are held in files, but the metadata is provided in part by the directory structure [19]:

```
/timit/train/dr1/fcjf0/sa1.wav
```

(TIMIT corpus, training set, dialect region 1, female speaker, speaker-ID "cjf0", sentence text "sa1", speech waveform file.) It would be quite reasonable to represent such metadata in XML, but it is immediately obvious that it requires a non-trivial hierarchical key structure.

In relational database design we also find the notion of a hierarchical key structure in *weak entities*. The key of a weak entity consists of the parent key and some additional identification of the dependent entity [14] (e.g. course Math120, section B).

To describe hierarchical key structures we introduce 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 a key (bible.book, {name}).

[3]We are grateful to Mark Liberman and Steven Bird of the Linguistic Data Consortium at the University of Pennsylvania for providing us with this example.

Observe that in a relative key $(Q, (Q', S))$, Q starts 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 uniquely specify 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 $(\epsilon, (Q', S))$ is equivalent to the key (Q', S). Thus keys defined in section 4 are a special case of relative keys. To distinguish these two notions we refer to the former as *absolute keys* or simply *keys*. Now consider two relative keys. We say that $(Q_1, (Q'_1, S_1))$ *immediately precedes* $(Q_2, (Q'_2, S_2))$ if $Q_2 = Q_1.Q'_1$. Also, any absolute 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 $(\epsilon, (Q'_2, S_2)) \in \Sigma$ which precedes $(Q_1, (Q'_1, S_1))$.

As an example, this set of keys is transitive:

(ϵ, (bible.book, {name}))
(bible.book, (chapter, {number}))

This set is not:

(ϵ, (bible.book, {name}))
(bible.book.chapter, (verse, {number}))

Any transitive set of relative keys must contain some absolute key.

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

(ϵ, (university, {name}))
(university, (dept.employee, {emp-id}))

To *identify* an `employee` node in this database, we need only to specify a university name and an `emp-id` within that university. However, to *add* a new employee to the database, we clearly need to specify a department for the employee. However, although this key specification is transitive, there is no way to identify a department and hence there could be many ways to add an employee. This motivates our final definition of *updatability* as shown below: With updatability, one can always insert an element in the "keyed" part of the document unambiguously by specifying where to insert the element using keys.

Definition. A set Σ of relative keys is *updatable* if it is transitive and whenever $(Q_1, (Q_2.n, S_1)) \in \Sigma$ there is a relative

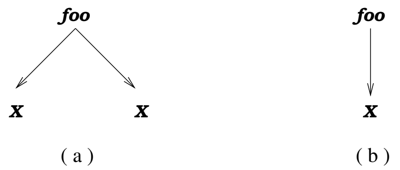

Figure 2: An XML tree conforming to D, and an XML tree satisfying φ

key $(Q_1, (Q_2, S_2)) \in \Sigma$ where $|Q_2| > 0$. Here n is a node name.

Informally, this definition gives us the property that every element with a prefix along the path $Q_1.Q_2$ can be identified through some keys. Therefore, it is easy to see that the addition of the following key will make the previous example updatable. In particular, to insert an employee, we now can specify which department they are in (in addition to the university).

$$(\mathsf{university}, (\mathsf{dept}, \{\mathsf{dept\text{-}name}\}))$$

Even though we can now add new employees, there is still something anomalous: Although employees are nested under departments, nothing about the department is necessary to identify them. This is reminiscent of the anomalies that occur in non-second normal form of relational databases. There is something wrong with the design of this 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. Formalizing the concept of a well-designed document with respect to its key specification is beyond the scope of this paper.

6.1 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. 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 system of relative keys: a relative key $(Q_1. \;\ldots\; .Q_{i-1}, (Q_i, \{P^i_1, \ldots, P^i_{k_i}\}))$ is defined for each i, $1 \leq i \leq n$. It should be noted that the first of these is of the form $(\epsilon, (Q_1, \{P^1_1, \ldots, P^1_{k_1}\}))$ and is a key.

For example

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

specifies the updatable system of keys:

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

So far the key hierarchies we have specified are linear. Consider the following two specifications:

$\mathsf{company}\{\mathsf{name}\}.\mathsf{employee}\{\mathsf{id}\}$
$\mathsf{company}\{\mathsf{name}\}.\mathsf{department}\{\mathsf{name}\}$.

It is helpful to fold these into a single specification:

$$\mathsf{company}\{\mathsf{name}\}[.\mathsf{employee}\{\mathsf{id}\}, .\mathsf{department}\{\mathsf{name}\}]$$

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

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

This is another example of a transitive set of relative keys. It is worthwhile to remark again that for identifying student nodes, one does not need to be aware of which school the student belongs to. However, to insert a new student into the document, one needs specify under which school (in addition to which university) to insert the student element so as to avoid ambiguity.

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 $\mathsf{student}(\underline{\mathsf{snum}}, \mathsf{name}, \mathsf{major})$ and $\mathsf{enroll}(\underline{\mathsf{snum}}, \underline{\mathsf{cnum}}, \mathsf{grade})$ with a key specification

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

They describe closely related structures. The specification $[.\mathsf{name}\{\}, .\mathsf{major}\{\}]$ ensures that under a $\mathsf{student}$ 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 $\mathsf{student}$ 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. As mentioned before, XPath is a language used for accessing parts of XML documents. XPath supports a variety of axes that allows one not only to move down an XML document tree from a node, but also to move to its ancestors and siblings. Moreover, one can embed predicates or even functions in XPath. For example /A/B[last()]/C/D/E/ancestor::* selects all ancestor nodes along the path A.B.C.D.E starting from the root. Observe that a predicate (qualifier) is specified in the expression: B must be the last B child of A. With such complex functionality, questions about the equivalence or inclusion of XPath expressions remains open. As demonstrated by examples in Section 5, 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. Recall that in a key $(Q,(Q',S))$ of our notation, the target path Q always starts from the root (also recall that an absolute key (Q',S) is equivalent to $(\epsilon,(Q',S))$). But it is straightforward to let Q start from arbitrary node: one needs simply to substitute $\_^*.Q$ for Q in our notation. More specifically, we write $(\_^*.Q,(Q',S))$ (observe that $\_^*.Q$ starts from the root). It is, of course, possible to "root" a path expression XML-Schema.

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. Since the issue of equivalence of XPath expressions is unresolved, there is no general method of checking 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:

book{name}.chapter{name}.verse{number}

In XML-Schema one can specify a key for verse (this is not XML-Schema syntax) as

(book.chapter.verse, [number, *up*.name, *up*.*up*.name])

Here "*up*" is the XPath instruction to move up one node. Thus part of the key is outside of the value 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 Some stronger definitions of keys

The definition of keys we have adopted in this paper is quite weak, which we believe is in keeping with the semi-structured nature of XML. This certainly does not mirror the requirements imposed by a key in relational databases, i.e. the *uniqueness* of a key and *equality* of key values. We now explore a definition which captures both these requirements.

Strong Keys. In a strong key definition, we require that the keys paths exist and are *unique*, i.e. $n[\![P_i]\!]$ contains exactly one node for $1 \le i \le n$. The key paths constrain 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 value-equal, then the nodes n_1 and n_2 are the same node.

As an example of what it means for a path expression to be unique, consider Figure 1: name is unique at $\langle 1\rangle$, but work and num are not unique at this node. The definition of satisfaction for strong keys now becomes the following.

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

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

To distinguish the two definitions of keys let us refer to keys defined above as *strong keys* and the keys defined in Section 4 as *weak keys*. Given this strong notion of keys, let us re-examine some examples given before.

$(\_*$ .person, {id})	Any two person elements, no matter where they occur, have unique id subelements and differ on those elements.
(person, $\{\epsilon\}$)	The interpretation of this key remains unchanged under a strong key semantics.
(employees, {})	Again, the semantics of this key is the same with respect to the strong and weak key specifications.
$(\_*$, {k})	This requires that every element has a key k, including any element whose name is k.

The last example illustrates that under a strong key semantics, finite satisfiability (the finite model property) does not hold for all keys: The key $(\_*$, {k}) imposes an infinite chain of k nodes and therefore, there is no finite document satisfying it. The problem arises because 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.

Due to the existence requirement on key paths in the definition of strong keys, a strong key imposes certain structural (typing) constraints which are typically found in schema specifications in a traditional database system. For example, the following document does not satisfy the strong key (A, {B}) since the key requires that B elements must exist under every A element (and be unique). In other words, it does not allow keys paths to have a "null" value. In contrast, the same document satisfies the weak key (A, {B}) as a weak key permits "null" value. Observe, however, the weak key clearly does not allow one to distinguish between these A elements.

```
<ROOT>
    <A> 1 </A>
    <A> 2 </A>
</ROOT>
```

It should be mentioned that the distinction between (traditional) structural constraints (types) and (traditional) integrity constraints is not always well-defined. It is dictated largely by what conventional programming languages treat as types. See [9] for detailed discussion on this topic.

The concept of relative keys can be naturally adapted for strong keys as well. We say a document satisfies a strong relative key specification $(Q, (Q', S))$ iff for all nodes n in $[\![Q]\!]$, n satisfies the strong key (Q', S).

The strong notion and weak notion of keys impose different restrictions on key paths. At one end of the spectrum, all key paths must exist and be unique (strong keys). At the other end, no structural constraints are imposed on key paths (weak keys). There are also possibilities in between; for example, adopting a slightly stronger notion of weak keys which substitutes *equality* for *value intersection* of the node sets reachable by a simple key path.

7.3 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. 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 $P_1, \ldots, P_n$. 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.

To simplify the discussion, so far we have required key paths to be simple paths. However, we could see no other benefit to simplifying the language of key paths. Below we extend the current proposal by allowing key paths to include $\_$ and $\_*$, i.e., to be expressed in the same language that defines target paths. To do so, we first define a notion of *value intersection*. Observe that the regular language defined by a path expression is a set of simple paths. Let us use ρ to range over simple paths. Given a path expression P, we use $\rho \in P$ to denote the simple path ρ in the language defined by P.

Value intersection. Let n_1 and n_2 be two nodes in an XML tree T and P be a path expression in the language defined in Section 3. The *value intersection* of $n_1[\![P]\!]$ and $n_2[\![P]\!]$, denoted by $n_1[\![P]\!] \cap_v n_2[\![P]\!]$, is defined as follows:

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

Intuitively, $n_1[\![P]\!] \cap_v n_2[\![P]\!]$ consists of pairs of nodes 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.

Using this notation, we extend our key specification as follows.

Key specification. A key is a pair $(Q, \{P_1, \ldots, P_n\})$, where Q and P_i's are path expressions in the language defined in Section 3. A node n *satisfies* the key iff for any n_1, n_2 in $n[\![Q]\!]$, if for all i, $1 \leq i \leq k$, the value intersection of $n_1[\![P_i]\!]$ and $n_2[\![P_i]\!]$ is not empty, then $n_1 = n_2$. That is,

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

It should be mentioned that the complexity results of [8] were developed for this general definition of keys.

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

```
<db>
   <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>
</db>
```

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. The 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 values); 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 adding a "virtual" subelement, `node-name` to each named node, whose value consists of the node name. With this extension, 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, Keishi Tajima, Chris Brew and Henry Thompson for a number of useful comments and discussions.

8. 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] Baker, Wendy and van den Broek, Alexandra and Camon, Evelyn and Hingamp, Pascal and Sterk, Peter and Stoesser, Guenter and Tuli, Mary Ann. The EMBL Nucleotide Sequence Database. In Nucleic Acids Research, 28(1):19-23, 2000.

[5] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. In Nucleic Acids Research, 28:45-48, 2000.

[6] D. Benson and I. Karsch-Mizrachi and D. Lipman and J. Ostell and B.A. Rapp and D. Wheeler. GenBank. In Nucleic Acids Research, 28(1):15-18, 2000.

[7] 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`.

[8] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Reasoning about keys for XML. University of Pennsylvania Technical Report MS-CIS-00-26, 2000. `http://db.cis.upenn.edu/DL/absolute-full.ps`

[9] Peter Buneman, Wenfei Fan, Jerome Simeon and Scott Weinstein. Constraints for Semistructured Data and XML. SIGMOD Record 30(1), March 2001.

[10] Byron Choi and Arnaud Sahuguet. DTD Inquisitor Demonstration. `http://xml.cis.upenn.edu/DTDi/`.

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

[12] Wenfei Fan and Leonid Libkin. On XML Integrity Constraints in the Presence of DTDs. In *PODS 2001*, 2001.

[13] Andrew Layman, Edward Jung, Eve Maler, and Henry S. Thompson. *XML-Data.* W3C Note, January 1998. `http://www.w3.org/TR/1998/NOTE-XML-data`.

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

[15] Arnaud Sahuguet. Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask. In *WebDB*, 2000.

[16] J. Sulston and Z. Du and K. Thomas and R. Wilson and L. Hillier and R. Staden and N. Halloran and P. Green and J. Thierry-Mieg and L. Qiu. The C. elegans genome sequencing project: A beginning. In Nature, 356(6364):37-41, 1992.

[17] 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/`.

[18] 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/who/wadler/xml.html`.

[19] TIMIT. CDROM TIMIT Directory and File Structure. `http://www.ldc.upenn.edu/readme_files/timit.readme.html`.

[20] GEML. `http://www.geml.org/`

[21] MAML. `http://www.oasis-open.org/cover/maml.html`