January 2001

Indexing Keys in Hierarchical Data

Yi Chen
University of Pennsylvania

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Yifeng Zheng
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-30.

Indexing Keys in Hierarchical Data

Abstract
Building on a notion of keys for XML, we propose a novel indexing scheme for hierarchical data that is based not only on the structure but also the content of the data. The index can be used to check the validity of data with respect to a set of key specifications, as well as for efficiently evaluating queries and updates on key paths. We develop algorithms for the construction and incremental maintenance of the indexing structure, and study the complexity of these algorithms. Finally, we discuss how our indexing techniques can be used for more general queries involving key paths.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-30.

Indexing Keys in Hierarchical Data

Yi Chen, Susan B. Davidson and Yifeng Zheng
Dept. of Computer and Information Science
University of Pennsylvania
yicn@saul.cis.upenn.edu, susan@cis.upenn.edu, yifeng@seas.upenn.edu

Abstract

Building on a notion of keys for XML, we propose a novel indexing scheme for hierarchical data that is based not only on the structure but also the content of the data. The index can be used to check the validity of data with respect to a set of key specifications, as well as for efficiently evaluating queries and updates on key paths. We develop algorithms for the construction and incremental maintenance of the indexing structure, and study the complexity of these algorithms. Finally, we discuss how our indexing techniques can be used for more general queries involving key paths.

1 Introduction

Keys are an essential aspect of database design, and give the ability to identify a piece of data in an unambiguous way. They can be used to describe the correctness of data (constraints), to reference data (foreign keys), and to update data unambiguously. In relational databases, indices are typically built on the primary key to allow efficient key lookups in queries and to optimize key constraint checking with updates.

As XML is increasingly being used as a data model, database issues such as schemas, constraints, indexing and storage mechanisms have become important considerations. Although a limited form of keys (and foreign keys) has been present in XML for some time in the form of "ID" (and "IDREFS"), the notion is unsatisfactory for several reasons: First, the keys are like oids and carry no real meaning in their value. In comparison, in a relational database a key is a set of attributes, and is value-based. Second, IDs are globally unique and are therefore untyped. Third, they do not carry a notion of hierarchy, which is a distinguishing feature of XML.

In response to these and other criticisms, several key definitions for XML have recently been introduced [1, 2, 3] and aspects of these proposals are finding their way into XML Schema [4]. In particular, the proposal in [1] presents a very general definition in which keys are specified in terms of path expressions (i.e. one or more attributes, subelements or more general structures) and can be scoped within an arbitrary substructure of the document. For example, they can be scoped at the level of the root (an absolute key) or at the level of elements reached through a specified path (a relative key). Moreover, multiple keys can be specified for a node, and are independent of the type (or DTD) of the document.

As an example, consider the sample document "books.xml" in figure 1. The document describes a set of books, which may have a set of authors (book authors). Since the document also contains edited collections, chapters in some books may also have a set of authors (chapter authors). We might wish to state that the key of an author is their ID attribute, no matter where the author occurs in the document (an absolute key), and that an alternate key for the author for a given book is their firstname and lastname (a relative key). We

```
<root> <book>
           <ISBN> 0123456789 </ISBN>
           <author @ID= "984567">
                  <firstname> Bob </firstname>
                  <lastname> Smith </lastname>
           </author>
           <chapter> <author @ID= "931228"> <lastname> Smith </lastname>
                     </author>
           </chapter>
      </book> ...  </root>
```

Figure 1: Sample XML document

might also wish to state that the key of a book, a top-level element located under the root of the document, is its ISBN element (an absolute key).

Several questions remain, however, with this proposal. First, how can queries on key paths (such as those found in updates) be optimized? Second, how can updates on a single node be specified and implemented using value-based keys? Third, how can the key constraints be efficiently enforced? In this paper we propose an answer to these questions in the form of an index structure, its generation and incremental maintenance algorithm.

There have been several proposals for indexing XML documents to optimize various types of queries. The most straightforward one regards an XML document as plain text [5, 6], and applies text indexing techniques from information retrieval. However, this approach does not consider the structure of the XML document, and queries are limited to keyword searches. Another approach is to store the XML document in a relational database [7, 8, 9]. Mature indexing techniques for those databases can then be used. However, since the structure of the XML document is not captured well Path, queries are quite inefficient, also the update cost of this approach is very high.

A third strategy is to index XML document directly. Since an XML document is composed of tags (structure) and text (content), such strategies fall in two categories: *value-based indexing* techniques and *path-based indexing* techniques. A "Vindex" (value-index) is proposed in [5], which is built over all the text nodes based on their values. Since this index does not involve any structure information, itself alone cannot be used for path queries, and an additional index structure must be used to locate the parent of a given node. In [5, 10, 11], several ways of indexing XML documents based on path expressions are proposed. [5] describes the "Pindex" (path index), which provides fast access to all objects reachable via a given labeled path. However, it requires a powerset construct over the underlying database, and so the index can in the worst case can be exponentially large. In [10], an index based on templates ("T-Index") is given. The technique consists in grouping database objects into equivalence classes containing objects that are indistinguishable. While the indexes based on path are very efficient for certain queries, for queries that involve constraints on the nodes along a given path in additions to those at the end of path, those indexes fail to prune the path exploration according to the constraints effectively, hence perform poorly.

Furthermore, any practical indexing scheme must be efficiently updatable as the underlying document evolves. To our knowledge, however, the only incremental update algorithm for path indicies is that in [11], which is based on bisimulation; yet even this technique only considers insertions of a single edge rather than arbitrary subtrees. Moreover, the auxiliary data structure is so complex that maintenance is very inefficient.

To solve these problems, we present a novel indexing scheme based not only on the hierarchical structure of the data but also the content of data. Capturing the hierarchical structure assists the efficient evaluation of path queries involving value-based keys. The content information helps prune unnecessary search space and guarantee the accuracy of the query result. The equivalence classes on which we base our index are defined in terms of key values. That is, we group nodes in the XML tree which have some key value in common.

In this paper, we make the following contributions:

1. An index structure, which speeds up queries related to keys;
2. An update semantics for insertions of subtrees under a given node and the deletion of a subtree rooted at a given node;
3. Bulk-load and incremental maintenance algorithms with complexity that is nearly linear in the size of the affected context, hence is near optimal;
4. A method to validate key constraints and prevent update anomalies.

The rest of the paper is organized as follows: Section 2 provides the definition of keys. Section 3 presents the index and how it is used for bulk-loading the file. A class of updates and the maintenance of the index upon updates are presented in section 4. In section 5, we analyze the time and space complexity of the generation and maintenance algorithms. We conclude with a summary and discussion of future work in section 6.

2 Keys

The definition of keys given in [1] has several salient features: First, keys are defined in terms of one or more path expressions, i.e. they may involve one or more attributes, subelements or more general structures. Equality is defined on tree structures instead of on simple text, referred to as *value equality*. Second, keys can be scoped within the context of the entire document (an *absolute* key), or within the context of particular subtrees (a *relative* key). An absolute key is a special case of a relative key. Third, the specification of keys is orthogonal to the typing specification for the document (e.g. DTD or XML Schema). The type of documents will therefore be ignored throughout this paper.

To define keys, we first give a precise definition of our model and value equality. We then discuss the path language used in keys, and present the notion of absolute and relative keys.

2.1 Tree Model and Value Equality

Our notion of keys is based on a simple tree model of XML data. Apart from the fact that our model is *unordered*, the model is similar to others commonly used for XML data (e.g. DOM [12]). An example of this representation for the XML document of figure 1 is shown in figure 2, in which each node has an *object identity* (oid) and is annotated by its type: E for element, A for attribute and S for text (or PCDATA). Element nodes carry a tag name, text nodes carry a value, whereas attributes carry both a name and a value. Note that we follow XML convention and indicate attribute labels by "@".

More formally, let $\mathcal{E}$ be a set of element tags, $\mathcal{A}$ a set of attribute names, and $\{\mathcal{S}\}$ be the singleton set denoting text (PCDATA).

Definition 2.1: An *XML tree* is defined to be $T = (V, lab, ele, att, val, r)$, where

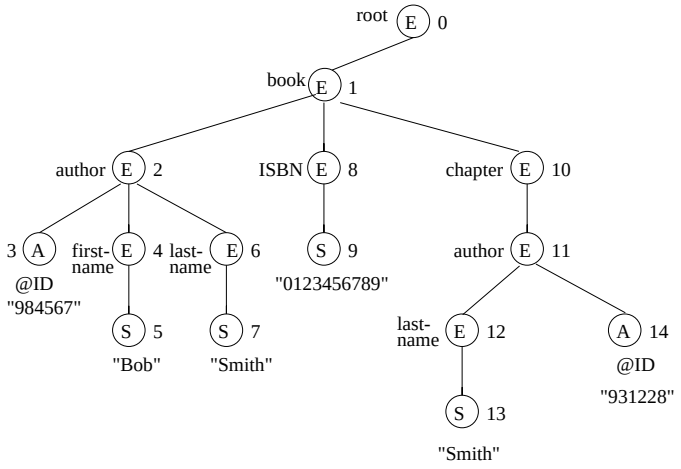

Figure 2: Tree representation of XML data

- V is a set of *vertices* (*nodes*) in T indicated by their oids;
- lab is a mapping from V to $\mathbf{E} \cup \mathbf{A} \cup \{\mathsf{S}\}$ 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 $lab(v) = S$;
- 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 $ele(v)$ ($att(v)$) is a set of elements(attributes) in V; v is said to be the *parent* of all nodes v' in $ele(v) \cup att(v)$, denoted *parent(v')*=v, and there is a directed edge from v to v';
  - if v is an attribute or text node then $ele(v)$ and $att(v)$ are undefined.
- 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;
- 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.

■

Given this representation of an XML document, we can use the standard terminology for directed trees. In particular, it will be useful when we discuss updates to talk about the descendents of a node n within a tree, denoted *desc(n)*, i.e. the set of nodes in the subtree rooted at n in T.

Two different notions of equality are used in our definition of keys: node equality and value equality. Given an XML tree, two vertices u and v are *node equal* ($u = v$) iff their oids are the same. Two nodes are value equal if they have the same type, label, and value (where defined), and their subtrees are value equal up to order. More precisely:

Definition 2.2: Two nodes u and v are *value equal* ($u =_v v$) if and only if

1. $lab(u) = lab(v)$;
2. if u, v are A or S nodes then $val(u) = val(v)$;

3. if u, v are E nodes then for every $a_u \in att(u) \cup ele(u)$ there exists an $a_v \in att(v) \cup ele(v)$ such that $a_u =_v a_v$ and vice versa.

■

2.2 Path Expressions and Keys

In defining a key we specify two things: a set on which we are defining a key (in relational databases this is a set of tuples denoted by a relation name) and the values which distinguish each element of the set (in relational databases, this is a set of attributes). When working with hierarchical data, specifying the set and the values involve path expressions.

A *path expression* is an expression that describes a set of paths in the tree. There are many options for path languages, ranging from simple ones involving just labels to more expressive ones such as regular languages or XPath [13]. Following [1], we adopt a simple language in which a path is a (possibly empty) sequence of node labels and wildcards. The language, which is a subset of both XPath and regular expressions, has the following syntax:

$$q \quad ::= \quad \epsilon \mid l \mid q.q \mid \_ \mid \_*$$

Here ϵ denotes the empty path, node label $l \in \mathbf{E} \cup \mathbf{A} \cup \{\mathsf{S}\}$, "." concatenates two path expressions, "_" matches a single label, and "_*" matches zero or more labels. Paths which are merely sequences of labels and do not contain _ or _* are called *simple paths*.

Note that for the purposes of this paper, the choice of a path language is not crucial; a more complex language will only make the algorithms of section 3 and 4 more complex. The benefit of this particular language is that it is quite expressive, paths always move down the tree, and equivalence of path expressions is decidable. We have also chosen this syntax instead 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 as shown in [1].

In what follows, we will use the notation $n[\![P]\!]$ to denote the set of nodes in T that can be reached by following the path expression P from node n in T. We also use $[\![P]\!]$ as an abbreviation for $r[\![P]\!]$, where r is the root node of T.

We are now in a position to define keys. A *key specification* is a pair $(Q, (Q', \{P_1, \ldots, P_p\}))$, where Q, Q' are path expressions and $\{P_1, \ldots, P_p\}$ is a set of simple path expressions. Q is called the *context path*, Q' the *target path*, and $P_1, ..., P_p$ the *key paths*. The idea is that the context path Q identifies a set of nodes $[\![Q]\!]$, each of which we refer to as a *context node*; for each context node $n \in [\![Q]\!]$, the key constraint must hold on the *target set* $n[\![Q']\!]$. A node $m \in n[\![Q']\!]$ is called a *target node*. A node in the set $m[\![P_i]\!]$, $i = 1, ..., p$, is called a *key node*, and the subtree rooted at a key node is called a *key tree*. When $Q = \epsilon$, we call the key an *absolute key*, otherwise it is a *relative* key.

Definition 2.3: Let $KS = (Q, (Q', \{P_1, \ldots, P_p\}))$ be a key specification. An XML tree T *satisfies* φ iff for each context node n in $[\![Q]\!]$ and for any target nodes m_1, m_2 in $n[\![Q']\!]$, if for all $i \in [1, \ldots, p]$ there exist $z_1 \in m_1[\![P_i]\!]$ and $z_2 \in m_2[\![P_i]\!]$ such that $z_1 =_v z_2$, then $m_1 = m_2$. That is,

$$\forall\, n \in [\![Q]\!] \quad \forall\, m_1\, m_2 \in n[\![Q']\!]$$
$$\Big(\bigwedge_{1 \le i \le p} \exists z_1 \in m_1[\![P_i]\!], \exists z_2 \in m_2[\![P_i]\!](z_1 =_v z_2) \Big) \rightarrow (m_1 = m_2).$$

■

Two examples of key specifications for the XML tree in figure 2 follow:

KS_1= $(\epsilon, (book, \{ISBN\}))$: a book is uniquely identified by its ISBN within the whole repository.

KS_2= $(book, (\_ * .author, \{firstname, lastname\}))$: the authors for a given book can be distinguished by their firstnames and lastnames.

KS_1 is an absolute key since its only context node is the root, while KS_2 is a relative key.

As mentioned in the introduction, the index will group nodes according to their key values, hence we will be storing key values in the index and testing equality over them. Since key values are themselves XML trees, we must be able to efficiently check value equality. Rather than naively implementing the definition of value equality, we will serialize key values and store them as strings so that two key values are value equal if and only if the two serialized key values are equal as strings.

In the syntax that follows, we use the delimiters "[" and "]" to represent levels of nesting.[1]

Definition 2.4: Let *SortElem(B)* be a function which takes a bag B, orders its elements lexicographically, and eliminates duplicates. Then the *serialized* form of a node v, *serialize(v)*, is defined as follows:

1. If v is text, then *serialize(v)*= $[val(v)]$.
2. If v is an attribute, then *serialize(v)*= $@lab(v)[val(v)]$.
3. If v is an element, then *serialize(v)* is the concatenation of $lab(v)$ with [*SortElem(*serialize(u), *for all* $u \in att(v) \cup ele(v)$ *)*]

■

To illustrate, the serialized form of some nodes in figure 2 is given below:

- *serialize(3)*= $@ID[984567]$
- *serialize(2)*= $author[firstname[[Bob]]lastname[[Smith]]@ID[984567]]$

It is easy to show that $u =_v v$ iff *serialize(u) = serialize(v)* (proof is deferred to the appendix).

A final notion that must be discussed before moving on is that of *transitivity* of a set of key specifications [1]. It is possible that a set of key specifications may not enable us to identify every node in an XML document using a value-based key. Some of the reasons for this have to do with the particular instance we are dealing with: First, a node in the instance may not match any of the key specifications. Second, two nodes in the instance may not be distinguishable by their key values. If two nodes u and v both have an empty value for some key path P_i and have some common value for all the other key paths, then u and v are not distinguishable by their key value. Note that they do not violate the key specification in this case. Such difficulties must be detected as the index is being constructed for a particular instance.

Howerver, there are potential problems that can be detected statically from the key specification. Consider the set of key specifications $\{KS_1, KS_2\}$ of the previous section. Since KS_2 is a relative key, by itself it

[1] Recall that our model of XML trees is *unordered*.

does not uniquely identify a particular book author in the whole XML tree. However, if we give the ISBNs of a book (KS_1) , the set of first names and the set of lastnames, we can uniquely specify an author as illustrated above. We formalize this as follows:

Definition 2.5: $(Q_1, (Q_1', P_1))$ *immediately precedes* $(Q_2, (Q_2', P_2))$ iff $Q_2 = Q_1.Q_1'$. The *precedes* relation is the transitive closure of the immediately precedes relation. ■

Note that in our example, KS_1 immediately precedes KS_2, and that by definition any absolute key immediately precedes itself.

Definition 2.6: A set Σ of keys is *transitive* iff for any relative key $(Q_1, (Q_1', P_1)))$ there is a key $(\epsilon, (Q_2', P_2))$ which precedes it. ■

Returning to our example, $\{KS_1, KS_2\}$ is transitive.

Checking that a set of key specifications is transitive can be checked in quadratic time in the number of keys [14]. Throughout the rest of this paper, we assume that the key set is transitive.

3 Efficiently Indexing Keyed Data

In order to efficiently query nodes according to their keys, we build an index. Unlike other approaches that use only either the content or the structure of the document, our index incorporates knowledge of both content and structure. Specifically, the hierarchical structure of the index reflects the hierarchical structure of the key specification, which assists efficient evaluation of key look-ups and certain types of path queries. At the bottom level, oids of all the nodes are grouped into equivalence classes according to their key values based on value equivalence, which helps to prune unnecessary search space both along and at the end of paths. In this section, we present the structure of the index as well as the algorithm to construct the index. In the next section we will discuss how to dynamically maintain the index in the face of updates to the XML document.

3.1 The index

The index is a hierarchical hash table structure, and can be thought of in levels. The top level is the *key specification level*, which partitions the nodes in the XML tree according to their key specifications. Since a node may match more than one key specification, it may appear in more than one partition. The second level is the *context level*, which groups target nodes by their context. The third level is the *key path level*, which groups nodes based on key paths. The fourth level is the *key value level*, which groups target nodes by equivalence classes. The equivalence classes are defined such that the nodes in a class have some key nodes which are value-equivalent, following the same key path under the same context in a particular key.

Definition 3.1: The equivalence relation defined over the target nodes of a key specification $KS_i = (Q, (Q', \{P_1, ..., P_p\}))$ is defined as follows: $u \sim_{ij} v$ iff there exists a context node $n \in [\![Q]\!]$ such that $u, v \in n[\![Q']\!]$ are target nodes and there exists one key path $P_j \in \{P_1, ..., P_p\}$ such that $u_{c1} =_v v_{c2}$ for some $u_{c1} \in u[\![P_j]\!]$, $v_{c2} \in v[\![P_j]\!]$.

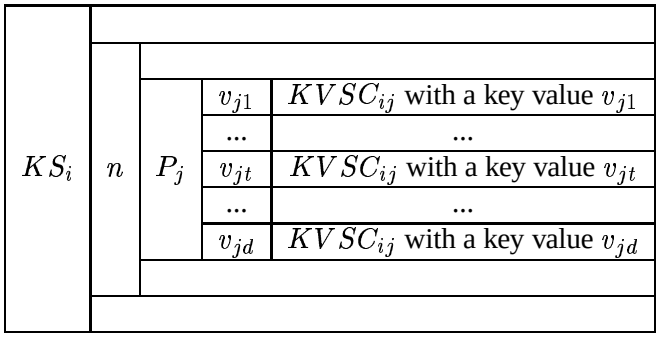

Figure 3: Index entry for key KS_i

KS_1	0	ISBN	0123456789	{1}
KS_2	1	first name	Bob	{2}
		lastname	Smith	{2, 11}
KS_3	0	@ID	984567	{2}
			931228	{11}

Figure 4: Key index for example 3.1

The equivalence classes induced by $\sim_{ij}$ over target nodes under the context node n of key specification KS_i and key path P_j are called *key value sharing classes* ($KVSC_{ij}$). ■

Figure 3 illustrates part of the index structure for a key specification $KS_i = (Q, (Q', \{P_1, ..., P_p\}))$. For each context node $n \in [\![Q]\!]$, target nodes $m \in n[\![Q']\!]$ are grouped into KVSCs for each key path $P_j \in \{P_1, ..., P_p\}$.

Example 3.1: Consider the repository of books "books.xml" in figure 1. Assume there are three key specifications:

$KS_1 = (\epsilon, (book, \{ISBN\}))$: a book is uniquely identified by its ISBN in the whole repository

$KS_2 = (book, (\_*.author, \{firstname, lastname\}))$: the authors for a given book can be distinguished by their first names and lastnames.

KS_3: $(\epsilon, (\_*.author, \{@ID\}))$: the authors can also be distinguished by their ID attribute in the whole document.

The index structure for these key specifications is shown in figure 4. Note that nodes 2 and 11 are each keyed by KS_2 and KS_3. ■

3.2 Index construction

The algorithm for index construction is shown in figure 5. The main process `KeyIndexBL` initiates a DFA for Q_i in every key specification and a SAX parser. SAX parses the XML document and drives the DFAs

into different states according to the tags encountered. When SAX recognizes a node, it will signal all DFAs with the node id and tag. Meanwhile each DFA is waiting to receive a signal from the SAX parser, and changes its state according the signal received. When a DFA reaches its final state, it will fork another DFA to deal with the next level. Specifically, the main process `KeyIndexBL` builds a process `DFA_QPath` for each key specification to find path Q. When process `DFA_QPath` reaches its final state (which means it has found a context node), it forks a DFA `DFA_Q'Path` to look for path Q' under this context node. Similarly, when `DFA_Q'Path` reaches its final state (which means that a target node is recognized), it will fork a DFA `DFA_Keypath` for each key path in parallel. When any of these DFAs reaches their final state, a key node is recognized and stored as a key value. Note that since Q and Q' may contain regular expressions, several context nodes for one key and several target nodes for one context node can be activated at the same time. Reflecting this, the DFAs do not block at their final state, but continue to seek the next matching.

`KeyIndexBL` also invokes another process `KeyCheck` to check satisfaction of the key specification. If the key constraint is satisfied, the target node is inserted into the corresponding entries(KVSCs) in the index. `KeyCheck` checks if there are two target nodes that share the some key value for every key path in some key specification $(Q, (Q', \{P_1, ..., P_p\}))$. For each key path P_i ($1 \leq i \leq p$), it unions all the KVSCs that a target node m belongs to, and produces the set of nodes S_i that share some key value with m. For all the key path $P_1, ..., P_p$ it then computes the intersection of $S_1, ..., S_p$ to get a set S, which is the set of nodes that share some key value for all the key paths. If there is more than one node in S, those nodes violate the key specification.

3.3 Answering Key-based Queries

Since the index partitions nodes in the document according to their KVSCs and stores the key values under their context, it can also be used for query optimization. Rather than going into detail we will give the intuition via some examples. The examples use XQuery [15] for syntax, however the ideas can be used for any XML query language.

Example 3.2: Retrieve the ID of the author of the book with $ISBN = \text{“}0123456789''$ whose first name is “Bob” and last name is “Smith”:

FOR $\$x_1$ IN (document(“books.xml”)/book)
    $\$x_2$ IN $\$x_1$//author
WHERE $\$x_1$/ISBN = “0123456789”
    AND $\$x_2$/firstname = “Bob” AND $\$x_2$/lastname = “Smith”
RETURN $\$x_2$/@ID

From KS_1 we know that $ISBN$ is a key of book and that the context is the root (node 0). The KVSC of book nodes with the key value “0123456789” following key path $ISBN$ is $\{1\}$, hence x_1 is bound to node 1. Since *firstname* and *lastname* are the key paths of an author under the context of a book, we can get the KVSC of author nodes with the key value *“Bob”* following the key path *firstname* under the context node 1. This class contains node 2. We can also get the KVSC of author nodes with the key value *“Smith”* following key path *lastname* under the context node 1, yielding $\{2, 11\}$. Since KS_2 has two key paths, we take the intersection of these equivalence classes and as a result bind x_2 to node 2. Assuming XML native storage, we can easily get the value of the id of node 2, “984567”. ■

Example 3.3: For book with ISBN “0123456789” or “2345678901”, retrieve their chapters whose author has last name “Smith”.

```
process KeyIndexBL (XMLdocument, keyspecification)
begin
    fork a process DFA_QPath for each keyspecification.
    initiate the SAX parser.
    if error
        then terminate all processes, ABORT
        else COMMIT
end
```

```
process DFA_QPath_KS_i
begin
    do
        receive(SAX, tag, n)
        drive the DFA into its next state according to tag
        if a context node is reached then fork a process DFA_Q'Path_KS_i (n)
    while not (end_of_document ∨ ERROR)
end
```

```
process DFA_Q'Path_KS_i (m)
begin
    context_node ← m
    construct context level in index.
    build key path level entries P_j, 1 ≤ j ≤ p in index.
    do
        receive(SAX, tag, n)
        drive the DFA into its next state according to tag
        if target node is reached
        then fork processes DFA_KeyPath_P_j,KS_i(n, context_node) 1 ≤ j ≤ p
    while (n is not the end tag of context_node )
    wait all DFA_KeyPath_P_j,KS_i processes return True
        then submit context level info to temporary index fi le on disk.
end
```

```
procedure DFA_KeyPath_P_j,KS_i (m, l)
begin
    target_node ← m  context_node ← l
    do
        receive(SAX, tag, n)
        drive the DFA into its next state according to tag
        if reach a key node then
            key_node ← n
            do
                receive(SAX, tag, n)
                build key_tree using tag and n
            while (n is not the end tag of the key_node )
            v ← serialize(key_tree)
            if v is not in key value level for P_j,KS_i
                create entry with empty KVSC.
    while ( n is not the end tag of target_node )
    if all other DFA_KeyPath_P_r,KS_i(target_node, context_node) r ≠ j processes has terminated then
        if KeyCheck (target_node, context_node) is invalid then
            output violated nodes and KS_i
            signal(ERROR)
        else
            lock(context_levelofindex)
            add target node to KVSC.
            unlock(context_levelofindex)
            return True
end
```

```
procedure KeyCheck(target_node, context_node)
begin
    lock(context_levelofindex)
        for j = 1 to p do
            for key path P_j, for any KVSC C_{j,t} node m belongs to,
                S_j ← ⋃_t C_{j,t}
            end
        end
        S ← ⋂_j S_j
        if |S| ≤ 1
            then return True
            else return False
    unlock(context_levelofindex)
end
```

Figure 5: Key index construction algorithm

FOR $\$x_1$ IN (document("books.xml")/book)
    $\$x_2$ IN $\$x_1$/chapter
WHERE $\$x_1$/ISBN = "0123456789" or "9876543210"
    AND $\$x_2$/author/lastname = "Smith"
RETURN $\$x_2$

The keys $KS_1 : (\epsilon, (book, \{ISBN\}))$ and $KS_2 : (book, (\_ * .author, \{firstname, lastname\})$ are determined to be related to this query, since path $book.ISBN = book.ISBN$ and $book.chapter.author.lastname \subseteq book.\_ * .author.lastname$. Similar to the above example, we retrieve node set $\{1\}$ first according to the first condition in the query. Under the context of node 1, we get the equivalence class of author nodes with the key value *"Smith"* following key path *lastname*. This class contains $\{2, 11\}$. Since the node we want to retrieve is "book.chapter", we look for whether nodes 2 and/or 11 have such a parent. Node 10 is returned as result.

Note that we needed an implementation of parent for this query. ■

From these examples, we can get an intuition of how the key index can be used to speed up queries that match the set of key specifications. In fact, for key-related queries the key index outperforms a V-index [5] which is based only on the content of the document, and path indexes which are based on structure summary (e.g. P-index [5], 1-index,2-index [10]). To see this, consider the fact that a V-index indexes over the values of all text nodes. Although we can easily get the text node(s) which have label "lastname" and value "Smith", many of these nodes may fail to have a path from the root which matches `book.chapter.author.name`. So the search space may be much bigger than necessary. On the other hand, a path index will search all nodes at the end of the path `book.chapter.author.name.lastname`, but does not check value constraints until the end. For queries in which we want to check constraints on the nodes along a given path in addition to those at the end of the path, the path index do not cut search space efficiently, as we can in this example.

4 Incremental maintenance of key index

In general, there are two different kinds of updates that may occur: updates to the key specification file (i.e. the insertion of a new key specification or the deletion of an existing one), and updates to the XML document. Updates to the key specification file are fairly straightforward, and involve modifications to the key specification level of the index. If a new key specification is inserted, we must feed the whole XML document and the new key specification to the $keyIndexBL$ algorithm. The resulting index generated is then added to the original index structure. If a key specification is deleted, the corresponding entry at the key specification level will be removed.

Handling updates to the XML document itself is more common and also more complicated since lower levels of the key index must be changed. In this section, we will discuss the semantics of such updates and how the index structure can be incrementally maintained.

4.1 Updates

Update operations in relational databases typically include an insert and delete operation; a modify operation also exists in many systems, but can be modeled by an insert followed by a delete. For XML trees, the natural analog of these operations is to insert a new tree below a given node in the tree, or to delete a subtree rooted at a given node. More complex operations (such as move and copy) have been proposed for describing the edit script between two trees [16, 17, 18] and for updating XML [19, 20]. However, for the purposes of this paper, insert and delete are sufficient.

Formally, let $T_1 = (V_1, lab_1, ele_1, att_1, val_1, r_1)$ and $T_u = (V_u, lab_u, ele_u, att_u, val_u, r_u)$. An *insert* operation is denoted by $insert(T_1, n, T_u)$, where T_1 is the initial tree and $n \in V_1$ is the graft node that becomes the parent of r_u. Note that n must be an element node. When we apply this operation, we will get a new tree $T_2 = (V_2, lab_2, ele_2, att_2, val_2, r_2)$, where $V_2 = V_1 \cup V_u$; lab_2 agrees with lab_1 (lab_u) for nodes in V_1 (V_u); $ele_2(n) = ele_1(n) \cup r_u$, and agrees with ele_1 (ele_u) for nodes in $V_1 - n$ (V_u); att_2, val_2 agrees with att_1 (att_u), val_1 (val_u) for nodes in V_1 (V_u), respectively; and $r_2 = r_1$.

Acting as the inverse of insert, the *delete* operation is denoted by $delete(T_1, n)$, where $n \in V_1$. When we apply this operation, we will get a new tree $T_2 = (V_2, lab_2, ele_2, att_2, val_2, r_2)$, where V_2 = V_1 - *desc(n)*; lab_2 agrees with lab_1 for nodes in V_1 - *desc(n)*; ele_2(*parent(n)*)= ele_2(*parent(n)*) - $\{n\}$; agrees with ele_1 for nodes in V_1 - (*parent(n)* $\cup$ *desc(n)*); att_2, val_2 agrees with att_1, val_1 for nodes in V_1 - *desc(n)*, respectively ; and $r_2 = r_1$.

To identify the node on which the operation acts, we use value-based keys to specify an update path. This idea can be undertood intuitively as follows: We start by identifying some node n_0 along the path from the root to n using a key specification $KS_0 \in \Sigma$ and the key value which identifies n_0. Note that KS_0 will always be an absolute key since its context is relative to the root. Within the context of the subtree rooted at n_0, some node n_1 is then identified in the same way; note that $KS_1 \in \Sigma$ will now have a context path which accepts the string of labels from the root of the original XML document to n_0 and hence will be a relative key. This is repeated until n is completely specified.

For example, we could identify node 2 in figure 2 as follows: First we identify node 1 using KS_1 as $ISBN = \{$"0123456789"$\}$. Within the context of node 1, we then identify node 2 using KS_2 as $firstname = \{$"Bob"$\}$ and $lastname = \{$"$Smith$"$\}$. The update path would then be $book(ISBN = \{$"0123456789"$\})._* .author(firstname = \{$"$Bob$"$\}, lastname = \{$"$Smith$"$\})$

To formalize the idea of an update path, we first define an identifier of a node for a set of key paths.

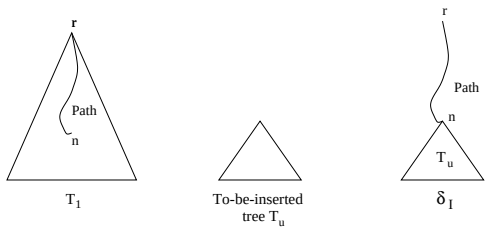

Figure 6: Delta XML tree for insertion

Definition 4.1: An *identifier* of a node for a set of key paths P is a set of matches of form $M = \{V_1, ..., V_l\}$, where $M \in P$ and V_j is the serialized form of a key tree value. ■

For example, the identifier of node 2 is: $(firstname = \{\text{“}Bob\text{”}\}, lastname = \{\text{“}Smith\text{”}\})$.

Definition 4.2: An *update path expression* has form $n_0(v_0).n_1(v_1).....n_u(v_u)$ where each n_i is a path expression such that

$$n_0.n_1.....n_{i-1} \subseteq Q, n_i \subseteq Q'$$

for some $(Q, (Q', P)) \in \Sigma$, and v_i is an identifier of n_i for P. ■

A few things about update path specifications should be noted: First, the notion relies on the fact that the set of keys used to define a node n is transitive. Second, a node may be identifiable by more than one update path. Third, an update path may skip nodes along the physical path to the node, as dictated by the key specifications used, and will always identify a unique node.

4.2 Maintaining the index

The algorithm for maintaining the index incrementally is a generalization of the bulk loading algorithm presented in the previous section. It takes as input a delta XML tree, which reflects the changes to the initial XML document, and modifies the initial index so that is is correct with respect to the updated XML tree.

A *delta XML tree* can be understood as follows: Given an update $insert(T_1, n, T_u)$, we create a tree T_n which is the path in T_1 from the root to n. The delta XML tree δ_I is then generated by grafting T_u as a child of n in T_n (see figure 6). Given an update $delete(T_1, n)$, the delta XML tree δ_D is formed by grafting the subtree rooted at n onto T_n. In both cases, T_n can be efficiently constructed using the parent and label functions discussed in section 2.1.

For an update to affect the index for a key specification $KS = (Q, (Q', P))$, the string represented by T_n must interact with the path expressions of KS, specifically those defined by Q, $Q.Q'$ or $Q.Q'.P_j$ for some $P_j \in P$. In addition to these three cases, we may be modifying the key value of an existing target node. In the following, $Path$ is the concatenation of labels from the root to n in T_n, and $Path \in PPrefix(P)$ ($Path \in Prefix(P)$) means that $Path$ is a proper prefix (prefix) of some path in the language defined by path expression P. Since the cases overlap (i.e. if $Path \in PPrefix(Q)$ then $Path \in PPrefix(Q, Q')$) the first case that is matched dictates the action performed.

1. $Path \in PPrefix(Q)$: Entries at the context level are inserted (for δ_I) or deleted (for δ_D).

   An example of this case would be the effect on KS_2 for the update $insert(T_1, r, T_u)$, where the content of T_u is `<book> <ISBN> 2345678901 </ISBN> </book>`.

   Note that bulk loading is a special case in which δ_I is the entire tree.

2. $Path \in PPrefix(QQ')$: One or more target nodes along with their key values are inserted (for δ_I) or deleted (for δ_D) within some existing key context. For insertion, we also need to check for violations of KS.

   The effect on KS_2 for the update $insert(T_1, book(ISBN = \{\text{“0123456789”}\}), T_u)$, where the content of T_u is `<author> <lastname> Dandy </lastname> </author>`, is an example of this case

3. $Path \in PPrefix(QQ'P_j)$: One or more key value(s) of an existing target node under some context are inserted (for δ_I) or deleted (for δ_D). For insertion, we must also check if KS is still valid.

   $insert(T_1, author(@ID = \{\text{“931228”}\}), T_u)$, where the content of T_u is `<firstname> Bob </first-name>`, is an example of this case for KS_2.

4. $QQ'P_j \in Prefix(Path)$: In this case we are inserting or deleting a subtree to a key node under an existing target node, hence the key value is changed. In this case we must efficiently recompute the new key value to be stored in the index without referring to the original XML document (recall that the index stores serialized key values). Details can be found in the appendix.

   For example, consider a modified version of the tree in figure 2 in which the first and last names of authors are grouped under a “name” element (e.g. node 2 has a child with label “name” with nodes 4 and 6 as children). Suppose we also have a new key specification: $(book, (_*.author, \{name\}))$. The key value for the modified node 2 is now
   $name[firstname[[Bob]]lastname[[Smith]]]$
   If we delete the “firstname” of “author”, the key value of “author” is changed to:
   $name[lastname[[Smith]]]$

   As we can see from this example, since deletion may change a key value we must check the validity of key specifications for both insertion and deletion.

Note that in the last two cases, a new key value for a node is inserted in the index. It turns out that it is quite inefficient to check if this causes a key constraint violation using only the index structure presented in the previous section since it entails retrieving all the key values of the updated node from the index structure. For example, consider a key specification $KS = (Q, Q', \{P_1, P_2\}))$, and suppose there are 100 key values for P_1 and 200 key values for P_2 in the existing index. If node m already has a key value v_2 for P_2 and a new key value v_1 for P_1 is inserted, we must obtain the set of nodes S_2 which share v_2 for P_2 with m. The only way to do this using the index from the previous section is to look at all the $KVSC$ classes for P_2 in KS to see which class(es) m belongs to. If the average size of the $KVSC$ class is 3, this means that 600 nodes must be traversed to calculate S_2. We then compute the intersection of S_1 (the set of nodes which share a key value with n for P_1) and S_2 to get the set S of nodes which violate KS with m.

To retrieve the $KVSC$ classes that m belongs to and compute the violation set S efficiently, we build an auxiliary index on the key index. Figure 7 illustrates the interaction between the key index (shown to the left inside the box) and the auxiliary index (shown to the right inside the box). The auxiliary structure indexes target nodes (m) under their context node (n). For each key path P_j, it keeps a pointer to the key values that target node m has. In the example above, it would keep a pointer to v_2 for key path P_2 for node m. We now only need to follow the pointer in the auxiliary index structure and perform one more lookup in the original index to retrieve S_2.

The original algorithm $KeyIndexBL$ can be revised straightforwardly following the discussion above. Details of the resulting algorithm $KeyIndexGen$ are deferred to a technical report [14].

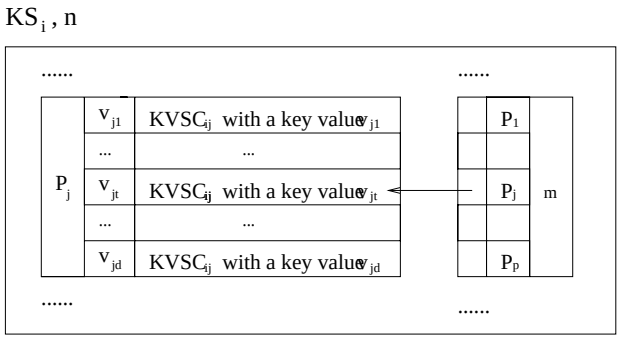

Figure 7: **Index entry for key** KS_i, n

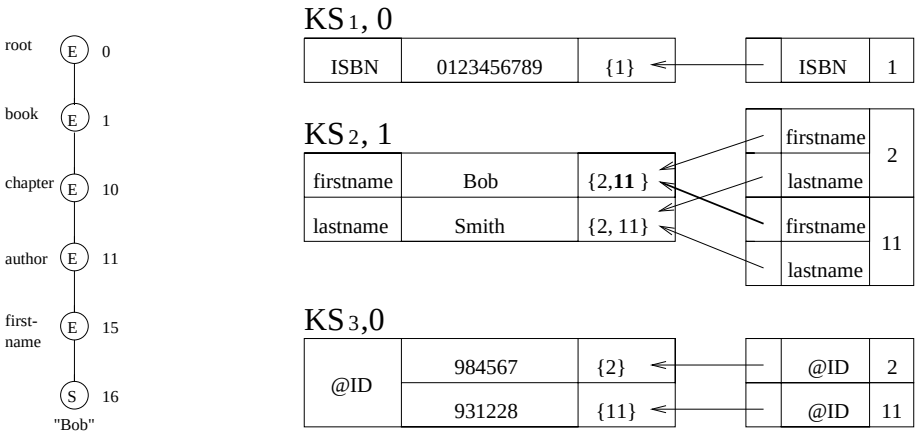

Figure 8: Delta tree and updated key index example

Example 4.1: Consider an update to the book repository of figure 2 which gives the author with ID "931228" a first name "Bob":

$$insert(T_1, author@ID = \text{“}931228\text{”}, T_u)$$

where the content of T_u is: `<firstname> Bob </firstname>`. This update inserts a key value under node 11 for key path "$firstname$". It is easy to see that this update does not affect key specifications KS_1 and KS_3. It does, however, affect KS_2 since $Path(n) = book.chapter.author$, $QQ'P_1 = book.\_ * .author.firstname$, and $Path \in PPrefix(QQ'P_1)$ (case 3 of the updates described earlier). After $KeyIndexGen$ processes the delta XML tree (shown in figure 8) for this update, the resulting index structure would appear as in figure 8. Upon checking the validity of KS_2, however, a violation is discovered and the update is rolled back. ■

It is easy to verify that $KeyIndexGen$ is correct in the sense that the resulting index is the same as that which would be generated by from $KeyIndexBL$ using the updated XML tree as input. It should also be pointed out that while the time complexity of $KeyIndexGen$ is not much worse than that for $KeyIndexBL$, we are paying for it with the auxiliary index described above; we must also have an implementation of the $parent$ function. (The $parent$ function was not used in section 3.) A detailed discussion of the efficiency of both algorithms will be discussed in the next section.

n : total number of nodes in the input XML tree file
For bulk load, n is the size of the original XML tree;
for incremental maintenance, it is the size of the delta XML tree.
n_k: average number of nodes in the key tree of a key path
a: degree of key tree of a key path (key tree is assumed to be complete)
q : average number of context nodes for a key specification
q': average number of target nodes under some context node for a key specification
p: average number of key paths in a key specification
c: average number of children of some target node, following some $p_j (j = 1, ..., p)$
d: the number of distinct key value of some key specification, under some context node,
following key path p_j
k : the number of key specifications
C_0: percentage of keys that are not affected by an update
C_1 (C_2, C_3, C_4): percentage of keys affected by a case 1 (2,3,4) update
s: average size of a $KVSC$ ($s = q' * c/d$).

Figure 9: Parameters for complexity analysis

5 Complexity

In this section, we discuss the analytical performance of the key index. Since the index generation algorithm for bulk loading is a special case of the incremental maintenance algorithm (case 1 of insertions), we will focus on the incremental maintenance algorithm.

As discussed in section 2, the syntax for a key $(Q, (Q', \{P_1, ..., P_p\}))$ specifies that Q and Q' are path expressions, while the key paths P_j are simple paths. To simplify the analysis in this section, we first consider the case in which Q and Q' are simple paths, and then move to the more general case. The reason for this is that when Q, Q' are path expressions there may be several context nodes activated simultaneously as parsing occurs, which adds a layer of complexity to the analysis.

5.1 Complexity of Key Checking with Simple Paths

The parameters to be used in the analysis can be found in figure 9.

Time Complexity for Insertion. We begin with one key specification only. The running time of $KeyIndexGen$ includes the time to parse the XML file, get the key values, build the index, and check the key constraints. For a case 1 insertion update, one or more entries will be inserted into the context level. Now we consider the time complexity. The cost to serialize a key tree into a string is proportional to the size of the key tree, $O(n_k * \log a)$, where a is a constant representing the degree of the key tree (assumed to be complete). The proof is deferred to the appendix. For one key specification, there are q context nodes, each of which has q' target nodes; each target node has p key paths, which has c children each in average. So the total time to construct key values for one key specification is: $O(q * q' * p * c * n_k * \log a)$. Since the number of nodes in all the key trees is bounded by the total number of nodes in the file, i.e. $q * q' * p * c * n_k \leq n$, this can be relaxed to $O(q * q' * p * c * n_k \log a) \subseteq O(n)$.

We must also consider the time complexity for key checking and index construction. To check some target node $m \in [[QQ']]$, we union all the KVSCs that share the same key value with m, following some key path P_j. Since on average m has c children following p_j, and the average size of the equivalence class is s, the complexity of the union operation is $O(s*c)$. According to the definition, two nodes violate a key constraint if and only if they share some key value for all key paths. We must then compute the intersection of the p sets of nodes that share some value following some key path, which has complexity $O(s*c*p)$. The time complexity to check one key specification over the whole file is therefore $O(s*c*p*q*q') \subseteq O(s*n/n_k)$. Therefore, the total time to process one key specification is bound by $O((s/n_k+1)*n)$. Although s/n_k differs for various XML file and key specifications, in the experiments we have run it increases very slowly as n increases. We can therefore consider the time complexity to be almost linear in the size of the affected context.

For a case 2 insertion, we insert one or more target nodes. The time complexity is therefore at most that of inserting a context node, $O((s/n_k+1)*n/q)$, where n/q is the size of the affected context. For a case 3 insertion, we insert key values for a target node, and the time complexity is bound by $O((s/n_k+1)*n/(q*q'))$. Case 4 is similar to case 3, except that we must also restore a serialized key value in the index to an XML tree so that the updated subtree can be grafted in, and then serializae the new key tree. The time complexity for the procedure to restore the XML tree is $O(n_k)$ (see appendix), hence the overall time complexity is still $O((s/n_k+1)*n/(q*q'))$.

The total time for processing k key specifications is therefore
$((C_1 + C_2/q + (C_3+C_4)/(q*q')) * (s/n_k+1) * n * k)$
which is almost linear in the size of the affected context and therefore close to optimal.

Note that the k key specifications can be processed in parallel. Also note that this analysis does not consider the I/O cost of storing a validated portion of the key index to disk.

Time complexity for deletion. For cases 1,2 and 3, it is impossible for a deletion to invalidate the key, and we therefore do not perform key checking. So the time complexity for case 1 is $O(1)$, the time complexity for case2 is $O(c*p)$, and that for case3 is $O(c)$. The time complexity for case 4 for deletion is the same as that for insertion, $O((s/n_k+1)*n/(q*q'))$. So, the total time complexity is:
$O((C_1 + c*p*C_2 + c*C_3 + ((s/n_k+1)*n/(q*q'))*C_4)*k)$. Again, this is almost linear in the size of the affected context and therefore close to optimal.

Space complexity (main memory). Since the definition of keys is based on context, we only need to keep one context in main memory for each key specification. The context includes a main index and a auxiliary index.

For the main index, the size of a KVSC is s and the size of a key value is n_k. So the size of one entry is $s+n_k$ for each key value. Since there are d distinct values for each key path and p key paths for each context node, the size of the main index structure for one key is $O((s+n_k)*d*p)$. The auxiliary index structure maintains pointers for every key value that each target node has, so the space needed for one key specification is $O(q'*p*c)$. The total space needed for k key specifications is therefore
$O(((s+n_k)*d*p + q'*p*c)*k*(C_1+C_2+C_3+C_4))$. Now let $C = C_1+C_2+C_3+C_4$. Since $s = q'*c/d$, $d \leq q'*c$, and $q*q'*p*c*n_k \leq n$, the above can be rewritten as $O((n_k*d+q'*c)*p*k*C) \subseteq O(q'*c*n_k*p*k*C) = O(n/q*k*C)$.

As we can see, the space complexity for the index construction and maintenance algorithm is linear in the affected context and the number of key specifications.

5.2 Complexity of Key Checking with Path Expressions

When Q and Q' are path expression, there may be several context nodes that are activated simultaneously while checking an XML tree, each of which may have several target nodes that are activated simultaneously. This differs from the case where Q, Q' are simple path, where at any time, at most one context/target node is activated.

Assuming we have enough resources, we can process all the activated context/target nodes in parallel. Specifically, the DFA for a Q path will activate a new process $DFA\_Q'Path$ to parse the following Q' path as long as it encounters a context node, which will run in parallel with processes for other activated context nodes. Similarly, the DFA for Q' path will activate a new process $DFA\_KeyPath$ for each key path as long as it encounters a target node, which will run in parallel with processes for other activated target nodes.

Now define an MCP (max context node along a path) as a context node which does not have any ancestor in the document which satisfies the path expression Q. Similarly, define an MTP(max target node along a path) as a target node which does not has any ancestor under its context node which satisfies the path expression Q'. According to this definition, every context/target node which is not an MCP/MTP node has an MCP/MTP node as its ancestor. Since a DFA process is forked when the begin tag of a context/target node is encountered and terminated when the corresponding end tag is received, the lifetime of non-MCP/MTP processes can be paralleled with that of its MCP/MTP ancestor node. Ignoring the cost of forking new processes and the loss of parallelism within the $KeyCheck$ procedure while locking the context of an index to perform updates, we only need to compute the time used by MCP/MTP processes.

To compute the time used by MCP/MTP processes, we first modify the definitions of q and $\dot{q}$ as follows:

q : average number of MCP for one key specification
q': average number of MTP under an MCP

Since there are no common descendents of any two MCP/MTP nodes, then $q * q' * p * c * n_k \leq n$, so we get the same time complexity as for simple paths.

We should also consider how many processes are running in parallel in the worst case. A single process DFA_QPath is needed to parse path Q, which invokes processes $DFA\_Q'Path$ to parse Q' for each context node that is reached. And the process $DFA\_Q'Path$ will invoke processes $DFA\_KeyPath$ to parse $KeyPath$ for each key node that is reached. The maximum number of such $DFA\_Q'Path$ processes that will run in parallel is bounded by the context node which is reached by the DFA_QPath process which is bounded by the height of the XML tree, h. The maximum number of $DFA\_KeyPath$ processes that will run in parallel is bounded by the number of $DFA\_KeyPath$ times the number of the key path, which is $h*p$. So the number of processes that run in parallel for each key specification is bounded by: $1+h*(1+p)$. For k key specification in total, the maximum number of processes running in parallel is therefore bounded by $(1 + h * (1 + p)) * k$. Although this number looks large, the worst case will rarely happen in practice for meaningful key specifications since it implies that every node along a path from root to a leaf is both a context node and target node.

Space complexity (main memory). As before, each context needs $O(n/q * p * C)$ space. Assuming that there are at most l context nodes in memory at the same time, then the space complexity is $O(n/q*p*C*l)$. However, it is easy to see that in the worst case l is the height of the XML tree.

6 Conclusion

In this paper, we present a novel approach to indexing hierarchical data that can be used to optimize queries which match the key specifications. In contrast to most indexes developed for hierarchical data, which are based on either only the structure or the content of the data, our index captures both. A query evaluator can therefore use information about path restriction and value conditions in the query to optimize the query. The algorithm to build the index can also be used to efficiently check the satisfaction of a set of key specifications.

We also give a syntax and semantics for updates, in particular for inserting a new subtree under a given node and deleting the subtree rooted at a given node. The given node is located by its key value using an update path. Note that an update path also gives a syntax for *foreign keys* in the context of a set of key specifications. In contrast to proposals such as [19, 20], our updates can be used to specify a single node.

After translating the updates into delta update trees, the updates can be efficiently applied to the index using an incremental algorithm. The complexity of this algorithm is in practice almost linear in the size of the affected context for updates. In this sense, the algorithm is nearly optimal. To gain this efficiency, the incremental maintenance algorithm requires, in addition to the key index, an auxiliary index over the key index and an implementation of the parent function.

Although the index has been proposed for queries on keys, in future work we plan to explore its use for more general queries. For example, for high frequency queries we could build a set of indexes which match the queries and can be used to efficiently retrieve the query result. For lower frequency queries, we can see if the key and high frequency query indexes match a portion of the query.

Although in this paper we have explored indexing XML data directly to optimize key checking, it would also be possible to map the data to a relational store and use DMBS support for keys and constraint checking. This approach is also part of our future work.

The algorithms have been implemented based on a SAX parser for XML, and we have successfully created indices for XML versions of EMBL(European Molecular Biology Laboratory) data files of 917620 nodes.

References

[1] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW10*, 2001.

[2] Extensible Markup Language (XML) 1.0 (Second Edition),W3C Recommendation, October 2000. `http://www.w3.org/TR/2000/REC-xml-200001006`.

[3] A. Layman, E. Jung, E. Maler, and H. Thompson. XML-Data, W3C Note, January 1998. `http://www.w3.org/TR/1998/NOTE-XML-data`.

[4] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. W3C Working Draft, April 2000. `http://www.w3.org/TR/xmlschema-1`.

[5] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Stanford University, Computer Science Department, 1998.

[6] D. Kha, M. Yoshikawa, and S. Uemura. An XML indexing structure with relative region coordinate. In *ICDE*, 2001.

[7] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *WebDB*, pages 47–52, 2000.

[8] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.

[9] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.

[10] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[11] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE*, 2002.

[12] DOM Level 3 core specification, September 2001. `http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913/`.

[13] J. Clark and S. DeRose. XML Path language (XPath), November 1999. `http://www.w3.org/TR/xpath`.

[14] Y.Chen, S. Davidson, and Y. Zheng. Indexing keys in hierarchical data. Technical Report MS-CIS-01-30, University of Pennsylvania, Computer and Information Science Department, 2001.

[15] XQuery 1.0: An XML query language, June 2001. `http://www.w3.org/XML/Query`.

[16] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 26–37, Portland, Oregon, 1997.

[17] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

[18] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 1995.

[19] A. Halevy I. Tatarinov, Z. Ives and D. Weld. Updating XML. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 2001.

[20] A. Laux and L. Martin. XML updates. `http://www.xmldb.org/xupdte/xupdate-wd.html`.

A Appendix

A.1 Correctness and complexity of the serialize algorithm

Definition 1.1: We define an equivalence relation $=_s$ on nodes in the XML tree to be: $v =_s u \iff serialize(v) = serialize(u)$, where "=" means string equality. ∎

Lemma 1.1: $v =_s u \iff v =_v u$ ∎

Proof: First, we prove that $v =_v u \Longrightarrow v =_s u$. Obviously, if $v =_v u$, then the height of the tree u equals that of the tree v. Now we prove it by induction on the height of the tree.
Base case: The height of the tree is 0, then node u and v are both either S nodes or A nodes. According the definition $v =_v u$, $val(u) = val(v)$, So we have $v =_s u$.
Suppose when the heights of the trees are $\leq k$, the claim holds. When the heights of the trees are $k+1$, if $v =_v u$, then according the definition, for any one child of v, namely, x, we can find a child of u, named as y, such as $x =_v y$, reverse is the same. Since the heights of the trees is $k+1$, the height of x and y are both less than or equal to k. so $x =_s y$. At the same time, because $v =_v u$, $lab(v) = lab(u)$. According to the definition of serialize, we have *serialize(v) =serialize(u)*, hence $v =_s u$.

Next, we need to prove that $v =_s u \Longrightarrow v =_v u$ Since $v =_s u \iff serialize(v) = serialize(u)$, we only need to prove that $serialize(v) = serialize(u) \Longrightarrow v =_v u$ Obviously, since *serialize(v)= serialize(u)*, then the nested level of [ ] of the string *serialize(v)* equals that of the string *serialize(u)*. We need to prove it by induction on the nested level.
Base case: The nested level of [] is 1. Then u, v are either S nodes or A nodes, it is obviously that $v =_v u$ from the definition of value equality.
Suppose when the nested level is less than or equals to k, $serialize(v) = serialize(u) \Longrightarrow v =_v u$. When the nested level is $k+1$, according to the definition of serialize, we know it must be of form $lab(u)[S_1.S_2...S_n]$, where the nested level of every child is less than or equal to k. So for any child of v, x, there is a child of u, y, such that $serialize(x) = serialize(y)$. By assumption, $x =_v y$, since $serialize(v) = serialize(u)$, $lab(v) = lab(u)$, we have $v =_v u$ from the definition of $=_v$.
By this lemma, when we want to decide whether two trees are value-equality, we only need to serialize these two trees and see if the result strings are same.

Currently our model assumes that the order of children nodes is irrelevant. It is easy to adapt the algorithm to support ordered data structure by using the depth-first transversal in the serialize method.

Lemma 1.2: The cost to serialize a key tree into a string is proportional to the size of the tree $O(n_k)$. ∎

Proof: Suppose the key tree is an a-nary complete tree. Let h be the height of this key tree. We have $a^h = O(n_k)$. For each level, we order all the labels lexicographically and concatenate these strings. The time for serializing a key tree into a key value is $O(a^{h-1} * a * \log a + a^{h-2} * a * \log a + +a * \log a) = O(a^h \log a) = O(n_k \log a)$

A.2 The *rev-serialize()* algorithm

The procedure *rev-serialize()* restores an XML tree from its serialized form. The implementation of *rev-serialize()* is an LR(0) parser which takes a string as input and output a tree.

In case 4 of updates, we need to modify the key value. To do this in an efficient way, we first restore an XML tree from the stored key value in the index according to procedure *rev-serialize()*, do the insertion or

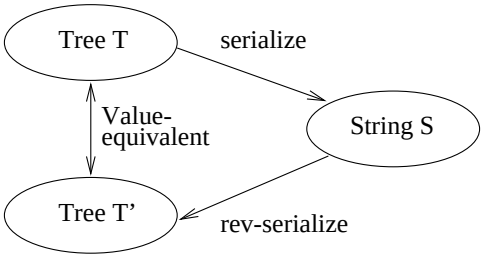

Figure 10: Illustration of procedure serialize and rev-serialize

deletion to the key tree as specified, and then apply *serialize()* to get the new key value. Thus the procedure *rev-serialize()* is the inverse of *serialize()*: $rev - serialize(\mathrm{serialize(T)}) =_v T$ (as shown in figure 10) .