

Available from: http://dx.doi.org/10.1007/978-3-540-31849-1_32

Copyright © 2005 Springer-Verlag Berlin Heidelberg. The original publication is available at www.springer.com.

This is the author’s version of the work. It is posted here with the permission of the publisher for your personal use. No further distribution is permitted. If your library has a subscription to these conference proceedings, you may also be able to access the published version via the library catalogue.
Checking Multivalued Dependencies in XML

Jixue Liu1  Millist Vincent1  Chengfei Liu2  Mukesh Mohania3

1 School of Computer and Information Science, University of South Australia
2 Faculty of Inf. and Comm. Technologies, Swinburne University of Technology
3 IBM Research Laboratories, New Delhi, India

email: {jixue.liu,millist.vincent}@unisa.edu.au, cliu@swin.edu.au
mkmukesh@in.ibm.com

Abstract. Recently, the issues of how to define functional dependencies (XFDs) and multivalued dependencies (XMVDs) in XML have been investigated. In this paper we consider the problem of checking the satisfaction of a set of XMVDs in an XML document. We present an algorithm using extensible hashing to check whether an XML document satisfies a given set of XMVDs. The performance of the algorithm is shown to be linear in relation to the number of tuples of the XML document, a measure which is related to, but not the same as, the size of the XML document.

1 Introduction

XML has recently emerged as a standard for data representation and interchange on the Internet [18, 1]. While providing syntactic flexibility, XML provides little semantic content and as a result several papers have addressed the topic of how to improve the semantic expressiveness of XML. Among the most important of these approaches has been that of defining integrity constraints in XML [7, 6]. Several different classes of integrity constraints for XML have been defined including key constraints [6, 5], path constraints [8], and inclusion constraints [9, 17], and properties such as axiomatization and satisfiability have been investigated for these constraints. Following these, some other types of constraints such as functional dependencies [2, 4, 3, 15, 12, 16], multivalued dependencies (XMVDs) [14], axioms, and normal forms have also been investigated. Once such constraints have been defined, one important issue that arises is to develop efficient methods of checking an XML document for constraint satisfaction, which is the topic of this paper. In this paper we address the problem of developing an efficient algorithm for checking whether an XML document satisfies a set of XMVDs. The problem is addressed in several aspects. Firstly, we propose an algorithm that is based on a modification of the extensible hashing technique. Another key idea of the algorithm is an encryption technique which reduces the problem of checking XMVD satisfaction to the problem of checking MVD satisfaction. Secondly, we show that this algorithm scans a document only

---

0 Supported by Australian Research Council Discovery Project DP0559202
once and all the information required for XMVD checking can be extracted in this single scan, even when there are multiple XMVDs to be checked. At the same time, we show that the algorithm runs in linear time in relation to the number of tuple of the XML document. The number of tuples of a document is different (though related) to the size of the XML document and is the number of combinations (we call them tuples) of the path values involved in the XMVDs.

2 Preliminary Definitions

In this section we review some preliminary definitions.

Definition 2.1 Assume a countably infinite set $E$ of element labels (tags), a countable infinite set $A$ of attribute names and a symbol $S$ indicating text. An XML tree is defined to be $T = (V, \text{lab}, \text{ele}, \text{att}, \text{val}, v_r)$ where $V$ is a finite set of nodes in $T$; lab is a function from $V$ to $E \cup A \cup \{S\}$; ele is a partial function from $V$ to a sequence of $V$ nodes such that for any $v \in V$, if $\text{ele}(v)$ is defined then $\text{lab}(v) \in E$; att is a partial function from $V \times A$ to $V$ such that for any $v \in V$ and $l \in A$, if $\text{att}(v, l) = v_1$ then $\text{lab}(v) \in E$ and $\text{lab}(v_1) = l$; val is a function such that for any node in $v \in V, \text{val}(v) = v$ if $\text{lab}(v) \in E$ and $\text{val}(v)$ is a string if either $\text{lab}(v) = S$ or $\text{lab}(v) \in A$; $v_r$ is a distinguished node in $V$ called the root of $T$ and we define $\text{lab}(v_r) = \text{root}$. Since node identifiers are unique, a consequence of the definition of val is that if $v_1 \in E$ and $v_2 \in E$ and $v_1 \neq v_2$ then $\text{val}(v_1) \neq \text{val}(v_2)$. We also extend the definition of val to sets of nodes and if $S_v \subseteq V$, then $\text{val}(S_v)$ is the set defined by $\text{val}(S_v) = \{\text{val}(v) | v \in S_v\}$.

For any $v \in V$, if $\text{ele}(v)$ is defined then the nodes in $\text{ele}(v)$ are called subelements of $v$. For any $l \in A$, if $\text{att}(v, l) = v_1$ then $v_1$ is called an attribute of $v$. The set of ancestors of a node $v$, is denoted by $\text{Ancestor}(v)$ and the parent node of $v$ is denoted by $\text{parent}(v)$ where the suffix $V$ means that the parent is a vertex (node).

Definition 2.2 (path) A path is an expression of the form $l_1, \ldots, l_n$, $n \geq 1$, where $l_i \in E \cup A \cup \{S\}$ for all $i, 1 \leq i \leq n$ and $l_1 = \text{root}$. If $p$ is the path $l_1, \ldots, l_n$ then $\text{end}(p) = l_n$.


Definition 2.3 Let $p$ denote the path $l_1, \ldots, l_n$ and the function $\text{parent}(p)$ return the path $l_1, \ldots, l_{n-1}$. Let $q$ denote the path $q_1, \ldots, q_m$. The path $p$ is said to be a prefix of the path $q$, denoted by $p \subseteq q$, if $n \leq m$ and $l_1 = q_1, \ldots, l_n = q_n$. Two paths $p$ and $q$ are equal, denoted by $p = q$, if $p$ is a prefix of $q$ and $q$ is a prefix of $p$. The path $p$ is said to be a strict prefix of $q$, denoted by $p < q$, if $p$ is a prefix of $q$ and $p \neq q$. We also define the intersection of two paths $p_1$ and $p_2$, denoted by $p_1 \cap p_2$, to be the maximal common prefix of both paths. It is clear that the intersection of two paths is also a path.

Definition 2.4 A path instance in an XML tree $T$ is a sequence $v_1, \ldots, v_n$ such that $v_1 = v_r$ and for all $v_i, 1 < i \leq n, v_i \in V$ and $v_i$ is a child of $v_{i-1}$. A path instance $v_1, \ldots, v_n$ is said to be defined over the path $l_1, \ldots, l_n$ if for all $v_i, 1 \leq i \leq n, lab(v_i) = l_i$. Two path instances $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_m$ are said to be distinct if $v_i \neq v'_i$ for some $i, 1 \leq i \leq n$. The path instance $v_1, \ldots, v_n$ is said to be a prefix of $v'_1, \ldots, v'_m$ if $n \leq m$ and $v_i = v'_i$ for all $i, 1 \leq i \leq n$. The path instance $v_1, \ldots, v_n$ is said to be a strict prefix of $v'_1, \ldots, v'_m$ if $n < m$ and $v_i = v'_i$ for all $i, 1 \leq i \leq n$. The set of path instances over a path $p$ in a tree $T$ is denoted by $\text{instances}(p)$. For a node $v$, we use $\text{instnodes}(v)$ to denote all nodes of the path instance ended at $v$. □

For example, in Figure 1, $v_r.v_1.v_3$ is a path instance defined over the path root.Dept.Section and $v_r.v_1.v_3$ is a strict prefix of $v_r.v_1.v_3.v_4$.  

\[
\text{instances(root.Dept)} = \{v_r.v_1, v_r.v_2\},
\]

\[
\text{instnodes}(v_2) = \{v_r, v_2\}.
\]

Fig. 1. An XML tree.

Definition 2.5 A set $P$ of paths is consistent if for any path $p \in P$, if $p_1 \subset p$ then $p_1 \in P$. □

This is natural restriction on the set of paths and any set of paths that is generated from a DTD will be consistent. We now define the notion of an XML tree conforming to a set of paths $P$.

Definition 2.6 An XML tree $T$ is said to conform to a set $P$ of paths if every path instance in $T$ is a path instance over a path in $P$. □

The next definition is to limit XML trees from having missing information.

Definition 2.7 Let $P$ be a consistent set of paths, let $T$ be an XML tree that conforms to $P$. Then $T$ is defined to be complete if whenever there exist paths $p_1$ and $p_2$ in $P$ such that $p_1 \subset p_2$ and there exists a path instance $v_1, \ldots, v_n$ defined over $p_1$, in $T$, then there exists a path instance $v'_1, \ldots, v'_m$ defined over $p_2$ in $T$ such that $v_1, \ldots, v_n$ is a prefix of the instance $v'_1, \ldots, v'_m$. □
Let \( \text{val} \) be \( \{ \text{root}, \text{root.Dept}, \text{root.Dept.Section}, \text{root.Dept.Section.Project} \} \) then the tree in Figure 1 conforms to \( P \) and is complete.

The next function returns the end nodes of the path instances of a path \( p \) under a given node. It is a restriction to \( \text{endnodes}(p) \).

**Definition 2.8** Let \( P \) be a consistent set of paths, let \( T \) be an XML tree that conforms to \( P \). The function \( \text{endnodes}(p) \), where \( p \in P \), is the set of nodes defined by \( \text{endnodes}(p) = \{ v | v \in \text{instances}(p) \land v = v_n \} \).

For example, in Figure 1, \( \text{endnodes(root.Dept)} = \{ v_1, v_2 \} \).

**Definition 2.9** Let \( P \) be a consistent set of paths, let \( T \) be an XML tree that conforms to \( P \). The function \( \text{bEndnodes}(v, p) \) (meaning branch end nodes), where \( v \in V \) and \( p \in P \), is the set of nodes in \( T \), defined by \( \text{bEndnodes}(v, p) = \{ x | x \in \text{nodes}(p) \land v \in \text{endnodes}(x) \} \).

For example in Figure 1, \( \text{bEndnodes}(v_1, \text{root.Dept.Section.Emp}) = \{ v_4, v_5 \} \).

We also define a partial ordering on the set of nodes and the set of paths as follows. We use the same symbol for both orderings but this causes no confusion as they are being applied to different sets.

**Definition 2.10** The partial ordering \( > \) on the set of paths \( P \) is defined by \( p_1 > p_2 \) if \( p_2 \) is a strict prefix of \( p_1 \), where \( p_1 \in P \) and \( p_2 \in P \). The partial ordering \( > \) on the set of nodes \( V \) in an XML tree \( T \) is defined by \( v_1 > v_2 \) if \( v_2 \in \text{Ancestor}(v_1) \), where \( v_1 \in V \) and \( v_2 \in V \).

### 3 XMVDs in XML

In this section, we present the XMVD definition and then give two examples.

**Definition 3.1** Let \( P \) be a consistent set of paths and let \( T \) be an XML tree that conforms to \( P \) and is complete. An XMVD is a statement of the form \( p_1, \ldots, p_k \rightarrow q_1, \ldots, q_m | r_1, \ldots, r_s \) where \( p_1, \ldots, p_k, q_1, \ldots, q_m \) and \( r_1, \ldots, r_s \) are paths in \( P \) and \( \{ p_1, \ldots, p_k \} \cap \{ q_1, \ldots, q_m \} \cap \{ r_1, \ldots, r_s \} = \emptyset \). A tree \( T \) satisfies \( p_1, \ldots, p_k \rightarrow q_1, \ldots, q_m | r_1, \ldots, r_s \) if whenever there exists a \( q_i, 1 \leq i \leq m, \) and two distinct path instances \( v_1^i, \ldots, v_n^i \) and \( w_1^i, \ldots, w_n^i \) in \( \text{instances}(q_i) \) such that:

(i) \( \text{val}(v_n^i) \neq \text{val}(w_n^i) \);  
(ii) there exists a \( r_j, 1 \leq j \leq s, \) and two nodes \( z_1, z_2, \) where \( z_1 \in \text{bEndnodes}(x_{i_j}, r_j) \) and \( z_2 \in \text{bEndnodes}(y_{i_j}, r_j) \) such that \( \text{val}(z_1) \neq \text{val}(z_2) \);
(iii) for all $p_l, 1 \leq l \leq k$, there exists two nodes $z_3$ and $z_4$, where $z_3 \in \text{branEndnodes}(x_{i_l}, p_l)$ and $z_4 \in \text{branEndnodes}(y_{i_l}, p_l)$, such that $\text{val}(z_3) = \text{val}(z_4)$;

then:

(a) there exists a path instance $v_{j_1}^{z_1}, \ldots, v_{j_n}^{z_1}$ in instances($q_i$) such that $\text{val}(v_{j_1}^{z_1}) = \text{val}(v_{j_n}^{z_1})$ and there exists a node $z_1^j$ in $\text{branEndnodes}(x_{i_l}, r_i)$ such that $\text{val}(z_1^j) = \text{val}(z_3)$ and there exists a node $z_3^j$ in $\text{branEndnodes}(x_{i_l}, r_i)$ such that $\text{val}(z_3^j) = \text{val}(z_4)$;

(b) there exists a path instance $w_{j_1}^{z_1}, \ldots, w_{j_n}^{z_1}$ in instances($q_i$) such that $\text{val}(w_{j_1}^{z_1}) = \text{val}(w_{j_n}^{z_1})$ and there exists a node $z_1^j$ in $\text{branEndnodes}(y_{i_l}, r_i)$ such that $\text{val}(z_1^j) = \text{val}(z_1)$ and there exists a node $z_4^j$ in $\text{branEndnodes}(y_{i_l}, r_i)$ such that $\text{val}(z_4^j) = \text{val}(z_4)$.

where $x_{i_l} = \{v|v \in \{v_{j_1}^{z_1}, \ldots, v_{j_n}^{z_1}\} \wedge v \in \text{instnodes}(\text{endnodes}(r_j \cap q_i))\}$ and $y_{i_l} = \{v|v \in \{w_{j_1}^{z_1}, \ldots, w_{j_n}^{z_1}\} \wedge v \in \text{instnodes}(\text{endnodes}(r_j \cap q_i))\}$ and $x_{i_l} = \{v|v \in \{v_{j_1}^{z_1}, \ldots, v_{j_n}^{z_1}\} \wedge v \in \text{instnodes}(\text{endnodes}(p_l \cap r_j \cap q_i))\}$ and $y_{i_l} = \{v|v \in \{w_{j_1}^{z_1}, \ldots, w_{j_n}^{z_1}\} \wedge v \in \text{instnodes}(\text{endnodes}(p_l \cap r_j \cap q_i))\}$ and $x_{i_l} = \{v|v \in \{v_{j_1}^{z_1}, \ldots, v_{j_n}^{z_1}\} \wedge v \in \text{instnodes}(\text{endnodes}(r_j \cap q_i))\}$ and $y_{i_l} = \{v|v \in \{w_{j_1}^{z_1}, \ldots, w_{j_n}^{z_1}\} \wedge v \in \text{instnodes}(\text{endnodes}(r_j \cap q_i))\}$ and $x_{i_l} = \{v|v \in \{v_{j_1}^{z_1}, \ldots, v_{j_n}^{z_1}\} \wedge v \in \text{instnodes}(\text{endnodes}(p_l \cap r_j \cap q_i))\}$ and $y_{i_l} = \{v|v \in \{w_{j_1}^{z_1}, \ldots, w_{j_n}^{z_1}\} \wedge v \in \text{instnodes}(\text{endnodes}(p_l \cap r_j \cap q_i))\}$.

We note that since the path $r_j \cap q_i$ is a prefix of $q_i$, there exists only one node in $v_{j_1}^{z_1}, \ldots, v_{j_n}^{z_1}$ that is also in $\text{branEndnodes}(r_j \cap q_i)$ and so $x_{i_l}$ is always defined and is a single node. Similarly for $y_{i_l}, x_{i_l}, y_{i_l}, x_{i_l}, y_{i_l}, x_{i_l}, y_{i_l}$. We also note that the definition of an XMVD is symmetrical, i.e. the XMVD $p_1, \ldots, p_k \rightarrow q_1, \ldots, q_m| r_1, \ldots, r_s$ holds if and only if the XMVD $p_1, \ldots, p_k \rightarrow q_1, \ldots, q_m| r_1, \ldots, r_s$ holds. We now illustrate the definition by some examples.

Example 3.1 Consider the XML tree shown in Figure 2 and the XMVD $C: \text{root.A.Course} \rightarrow \rightarrow \text{root.A.B.Teacher.S[root.A.C.Text.S]$. Let $v_{j_1}^{z_1}, \ldots, v_{j_n}^{z_1}$ be the path instance $v_{r,v_{12},v_{2},v_{4},v_{8}}$ and let $w_{j_1}^{z_1}, \ldots, w_{j_n}^{z_1}$ be the path instance $v_{r,v_{12},v_{2},v_{5},v_{9}}$. Both path instances are in instances($\text{root.A.B.Teacher.S}$) and $\text{val}(v_{j_1}^{z_1}) \neq \text{val}(v_{j_1}^{z_1})$. Moreover, $x_{i_1} = \{v_{12}, v_{12}, v_{r,v_{12},v_{2},v_{4},v_{8}}\} \wedge v_{12} \in \text{instnodes}(\text{endnodes}(\text{root.A.Course}))) \wedge \text{val}(z_1) = \text{val}(z_4)$. Hence conditions (i), (ii) and (iii) of the definition of an XMVD are satisfied. If we let $v_{j_1}^{z_1}, \ldots, v_{j_n}^{z_1}$ be the path $v_{r,v_{12},v_{2},v_{4},v_{8}}$ we firstly have that $\text{val}(v_{j_1}^{z_1}) = \text{val}(v_{j_1}^{z_1})$ as required. Also, since the path instances are the same we have that $x_{i_1} = x_{i_1}$ and $x_{i_1} = x_{i_1}$. So if we let $x_{i_1} = x_{i_1}$ then $z_1 \in \text{branEndnodes}(x_{i_1}, \text{root.A.Course})$ and $\text{val}(z_1) = \text{val}(z_2)$ and if we let $x_{i_1} = x_{i_1}$ then $z_3 \in \text{branEndnodes}(x_{i_1}, \text{root.A.Course})$ and $\text{val}(z_3) = \text{val}(z_3)$. So part (a) of the definition of an XMVD is satisfied.
Next if we let \(w_1', \ldots, w_n'\) be the path \(v_1, v_2, \ldots, v_9\) then we firstly have that \(val(w_1') = val(w_n')\) since the paths are the same. Also, since the paths are the same we have that \(y_{11} = y_{11}'\) and \(y_{111} = y_{111}'\). So if we let \(z_2' = v_{10}\) then \(z_2' \in \text{branEndnodes}(y_1', \text{root.A.Text.S})\) and \(val(z_2') = val(z_1)\) and if we let \(z_4' = v_1\) then \(z_4' \in \text{branEndnodes}(x_1', \text{root.A.Course})\) and \(val(z_4') = val(z_4)\). Hence part (b) on the definition of an XMVD is satisfied and so \(T\) satisfies the XMVD \(C\).

![Fig. 2. Satisfaction of XMVD](image)

More example showing the satisfaction and dissatisfaction of XMVDs can be found in [14].

4 Algorithm of checking XMVD

In this section, we present our algorithm for checking XMVD satisfaction.

Given an XMVD \(P \rightarrow Q | R\), where \(P = \{p_1, \ldots, p_n\}\), \(Q = \{q_1, \ldots, q_n\}\), and \(R = \{r_1, \ldots, r_n\}\), we let \(S = \{s_1', \ldots, s_n'\} = P \cup \overline{Q} \cup \overline{R}\). We call \(n\) the **number of paths involved** in the XMVD. To check this XMVD against a document, we firstly parse the document to extract values for \(s_i\), \(1 \leq i \leq n\). These values are then combined into tuples of the form \(<v_1, \ldots, v_n>\) where \(v_i\) is a value for the path \(s_i\). Finally, the tuples are used to check the satisfaction of the XMVD. For parsing a document, we need to define a control structure based on the paths involved.

4.1 Defining parsing control structure

We sort \(S = P \cup Q \cup R\) by using string sorting and denote the result by \(S' = [s_1, \ldots, s_n]\). We call the set of end elements of the paths in \(S'\) **prime end elements** and denoted by \(PE\), i.e., \(PE = \{endL(s) | s \in S\}\) where \(endL()\) is defined in Definition 2.2. Note that \(S'\) being a list can simplify the calculation of all the intersections of path in \(S'\) as shown below. Consider the example in Figure 3 which will be a running example in this section. Let an XMVD be \(r.A.C.F \rightarrow r.A.C.D.E|r.A.B\). Then \(S' = [r.A.B, r.A.C.D.E, r.A.C.F] \) and \(PE = \{B, E, F\}\).
Let $H = \{h'_1, \cdots, h'_m\}$ be a set of paths that are the intersections of paths in $So$ where $h'_i = s_i \cap s_j$, $(i = 1, \cdots, m)$, $j = i + 1$ and $m = n - 1$. We call the elements ending the paths in $H$ intersection end elements and denote the set by $IE$. We call both intersection end elements and prime end elements key end elements and denote them by $KE$. In Figure 3, $H = \{r.A, r.A.C\}$, $IE = \{A, C\}$, $KE = \{A, C, B, E, F\}$.

Now we define and calculate the contributing elements of intersection elements. Let $iE$ be the intersection end element of an intersection path $h$, i.e., $iE = \text{endL}(h)$. The contributing elements of $iE$, denoted by $CE(iE)$, are defined to be all key end elements of $So$ and $H$ under $h$ but not contributing elements of any other intersection end elements. To calculate the contributing elements, we first sort the paths in $H$ by applying string sorting and put the result in $Ho$ as $[h_1, \cdots, h_m]$. We then follow the following algorithm to calculate contributing elements.

**Algorithm 4.1 (calculation of contributing elements)**

**Input:** $So = [s_1, \cdots, s_n]$ and $Ho = [h_1, \cdots, h_m]$  
**Do:** For $h = h_m, \cdots, h_1$ in order,  
- Foreach $s$ in $So$, if $s$ is not marked and if $\text{endL}(s)$ is a descendent of $h$, put $\text{endL}(s)$ in $CE(\text{endL}(h))$ and mark $s$.  
- Foreach $h_x \in Ho$, if $h_x$ is not marked and if $\text{endL}(h_x)$ is a descendent of $h$, put $\text{endL}(h_x)$ in $CE(\text{endL}(h))$ and mark $h_x$.  
**Output:** $CE(\text{endL}(h))$ for all $h \in Ho$.

In the running example of Figure 3, the results of applying Algorithm 4.1 are $CE(C) = \{E, F\}$ and $CE(A) = \{B, C\}$.

### 4.2 Parsing a document

We define the function $\text{val}(k)$ to mean the value set of a key end element. Note that if $k$ is a prime end element, $\text{val}(k)$ will be accumulated if there are multiple presences of the same elements under a same node. If $k$ is an intersection end element, $\text{val}(k)$ is a set of tuples generated by the production of the values and/or value sets of contributing elements of $k$. We call each element of the production a tuple. Obviously, $\text{val}(k)$ changes as the parsing progresses. We now show some
examples of val(k) w.r.t Figure 3. When parsing reaches tag <C> in Line 3, val(C), val(E) and val(F) are all set to empty. When parsing has completed Line 6, val(E) = {"e1", "e2"} and val(F) = {"f1", "f2"}. When parsing of Line 7 is completed, val(C) = {<"e1", "f1">, <"e1", "f2">, <"e2", "f1">, <"e2", "f2">}. We further define some notation. stk denotes a stack while stkTop is used to refer to the element currently at the top of the stack. For simplicity, we define e ∈ X, where e is an element and X is a path set, to be true if there exists a path x ∈ X such that endL(x) = e. With all these definitions, we present the algorithm that parses a document. Also for simplicity, we use val(h) to mean val(endL(h)) where h is a path.

Algorithm 4.2 (parsing documents)
Input: S0, Ho, CE, PE, an empty stk, and a document
Do: Foreach element e in the document in the order of presence
    if e ∈ Ho and e closes stkTop, do production of the contributing attributes as the following:
    let CE(e) = {e1, ..., ec}, then
    val(e) = val(e) ∪ {val(e1) × ... × val(ec)}
    Note that some val(ei) can be sets of tuples while the others can be sets of values.
    else if e ∈ Ho
    push e to stk
    reset val(e1), ..., val(ec) to empty where e1, ..., ec ∈ CE(e)
    else if e ∈ PE
    read the value v of e and add v to val(e)
    if e is a leaf element, v is the constant string on the node.
    if e is an internal node, v is ’null’.
    else
    ignore the element and keep reading
Output: val(h1) where h1 is the first element in Ho - the shortest intersection path.
Note that the algorithm scans the document only once and all tuples are generated for the XMVD.

In the algorithm, the structures S0, Ho, CE, PE, an empty stk form a structure group. If there are multiple XMVDs to be checked at the same time, we create a structure group for each XMVD. During document parsing, for each element e read from the document, the above algorithm is applied to all structure groups. This means for checking multiple XMVDs, the same document is still scanned once.

4.3 Tuple attribute shifting and XMVD checking
For each tuple t in val(h1), where h1 is the first element in Ho - the shortest intersection path, it contains a value for each paths of the XMVD, but the
values are in the order of their presence in the document. This order is different from the order of paths in the XMVD. For example in Figure 3, the tuple <"b1","e1","f1"> is in val(root.A) and the values of the tuple are for the paths r.A.B, r.A.C.D.E, r.A.C.F in order. This order is different from the order of paths in the XMVD: r.A.B, r.A.C.D.E r.A.C.F. We define the operation shiftAttr() to rearrange the order of values of t so that the values for the paths in P are moved to the beginning of t, the values for the paths in Q are moved to the middle of t, and the values for the paths in R are at the end of t. The shiftAttr() operation is applied to every tuple of val(h1) and the result is denoted by Tsa = shiftAttr(val(h1)).

We define a further function removeDuplicates(Tsa) to remove duplicating tuples in Tsa and we denote the returned set as Tdist. Thus, the following algorithm checks whether an XML document satisfies an XMVD and this algorithm is one of the main results of our proposal. The basic idea of checking is to group all the tuples for the XMVD so that tuples with the same P value is put into one group. In each group, the number of distinct Q values, |Q|, and the number of distinct R values, |R|, are calculated. Then if |Q| × |R| is the same as the number of distinct tuples in the group, the group satisfies the XMVD; otherwise, it violates the XMVD. If all the groups satisfy the XMVD, the document satisfies the XMVD.

**Algorithm 4.3 (checking mvd)**

Input: Tdist

Do: violated = 0

For G be each set of all tuples in Tdist having the same P value

let |G| be the number of tuples in G

let dist(Q) and dist(R) be the numbers of distinct Q values and of distinct R values in G respectively

if ( |G| ! = dist(Q) × dist(R) ) then violated + +;

Output: if ( violated == 0 ) return TRUE; otherwise return FALSE.

Note that this algorithm assumes that G is the set of all tuples having the same P value. To satisfy this assumption, one has to group tuples in Tdist so that tuples with the same P value can be put together. A quick solution to this is not direct as show in the next section and therefore the way of achieving the assumption greatly affects the performance of whole checking algorithm.

5 Implementation and Performance

In this section, we present the performance results of our tests using an adapted hashing implementation. The tests were done on a Pentium 4 computer with 398 MB of main memory. The tests used XMVDs involving 12 paths, 9 paths, 6 paths, and 3 paths respectively. The XML documents used in the tests have random string values of about 15 characters for each path. This means that if
there are three paths involved in an XMVD, the length of a tuple is about 45
caracters while if there are twelve paths in an XMVD, the length of a tuple is
around 180 characters.

In Algorithm 4.3, an assumption is taken that all tuples having the same
$P$ value are grouped together. At the same time, in each group, the number of
distinct $Q$ values and the number of distinct $R$ values need to be calculated. To
obtain these numbers, we choose to use an adapted hashing technique which is
based on the standard extensible hashing [11]. In the standard extensible hashing
technique, each object to be hashed has a distinct key value. By using a hash
function, the key value is mapped to an index to a pointer, pointing to a fixed
size basket, in a pointer directory. The object with the key value will then be
put into the pointed basket. Every time a basket becomes full, the directory
space is doubled and the full basket is split into two. In our implementation,
we use the digests, integers converted from strings, of $P$ values of tuples as the
key values of the standard extensible hashing. Our modification to the standard
extensible hashing technique is the following.

The baskets we use are extensible, meaning that the size of each basket is
not fixed. We allow only the tuples with the same key value to be put into a
basket. We call the key value of the tuples in the basket the basket key. Tuples
with different keys are said conflicting. If placing a tuple into an existing basket
causes a conflict, a new basket is created and the conflicting tuple is put into the
new basket. At the same time, the directory is doubled and new hash codes are
calculated for both the existing basket key and the new basket key. The doubling
process continues until the two hash codes are different. Then the existing and
the new baskets are connected to the pointers indexed by the corresponding hash
codes in the directory.

Figure 4 shows the directory and a basket. Note that because of directory
space doubling, a basket may be referenced by multiple pointers. In the diagram,
p stands for the basket key, the three spaces on the right of p are lists storing
distinct $Q$ values, distinct $R$ values and distinct $Q$ and $R$ combinations. On top
of the lists, three counters are defined. $nq$ stands for the number of distinct $Q$
values, $nr$ the number of distinct $R$ values and $nqr$ the number of distinct $Q$
and $R$ combinations. After hashing is completed, the three counters are used to
check the XMVD as required by Algorithm 4.3. When a new tuple $t = < p, q, r >$,
where $p$, $q$ and $r$ are values for $P$, $Q$ and $R$ respectively, with the same $p$ value
is inserted to a basket, $q$ is checked against all existing values to see if it equals
to one of them. If yes, the $q$ value is ignored; otherwise, the $q$ value is appended
to the end of the list and the counter is stepped. Similar processes are applied
to insert $r$ and the combination $< q, r >$.

We now analyze the performance of hashing. Obviously the calculation of
hash codes to find baskets for tuples is linear in relation to the number of tuples.
When a tuple $t = < p, q, r >$ is put into a basket that has already has tuples,
then comparisons are needed to see if $q$, $r$ and the combination $< q, r >$ are
already in the lists. The performance of the comparison relates to the number
of distinct existing values. Generally, if we need to put \( n' \) values into a list that has had \( m \) distinct values, then the performance is \( O(n' * m) \) comparisons.

We conducted a number of experiments to observe the performance of the implementation and the results are given in Figure 5. In this experiment we plotted the time taken for checking XMVD satisfaction against the number of tuples in the document, for varying numbers of paths in the XMVD (we used 6, 9, and 12). In the cases of 3 paths and 6 paths, we see that the former has a higher cost. This can be explained because the overall performance contains the time for parsing documents. To have the same number of tuples in the cases of 3 paths and 6 paths, the 3 path case has a much larger file size, about 70 times of that of 6 paths and therefore the parsing time used is much larger in contrast to that of 6 path case. It is the parsing time that makes performance for 3 paths worse than that for 9 or 12 paths.

![Fig. 4. The structure of a basket and one extension](image1)

![Fig. 5. Performance of hash method](image2)

We also implemented the algorithms in a sorting based approach to compare the performance of this hashing based approach. The result of the sorting implementation and its comparison with the hashing based approach are given in the full version of this paper [10].

6 Conclusions

In this paper we have addressed the problem of developing an efficient algorithm for checking the satisfaction of XMVDs, a new type of XML constraint that has
recently been introduced [13, 14]. We have developed an extensible hash based
algorithm that requires only one scan of the XML document to check XMVDs.
At the same time its running time is linear in the size of the application which is
proved to be the number of tuples. The algorithm can check not only the cases
where there is only one XMVD, but also the cases involving multiple XMVDs.

References

1. Serge Abiteboul, Peter Buneman, and Dan Suciu. Data on the Web - From Rela-
tions to Semistructured Data and XML. Morgan Kayfman Publisher, 2000.
2. Marcelo Arenas and Leonid Libkin. A normal form for xml documents. PODS,
2002.
6. Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew
8. Peter Buneman, Wenfei Fan, and Scott Weinstein. Path constraints in semistruc-
9. Wenfei Fan and Leonid Libkin. On xml integrity constraints in the presence of

12. Millist Vincent and Jixue Liu. Functional dependencies for xml. LNCS 2642 -
13. Millist Vincent and Jixue Liu. Multivalued dependencies and a 4nf for xml. LNCS
2681 - Proc. International Conference on Advanced Information Systems Engi-
14. Millist Vincent and Jixue Liu. Multivalued dependencies in xml. LNCS 2712 -
16. Millist Vincent, Jixue Liu, and Chengfei Liu. Strong functional dependencies and
their application to normal forms in xml. Accepted on 05/Feb/2004 for publication
17. Millist Vincent, Michael Schrefl, Jixue Liu, Chengfei Liu, and Solen Dogen. Gen-