# Checking Functional Dependency Satisfaction in XML

Millist W. Vincent and Jixue Liu

School of Computer and Information Science
University of South Australia
{millist.vincent,jixue.liu}@unisa.edu.au

**Abstract.** Recently, the issue of functional dependencies in XML (XFDs) have been investigated. In this paper we consider the problem of checking the satisfaction of an XFD in an XML document. We present an efficient algorithm for the problem that is linear in the size of the XML document and linear in the number of XFDs to be checked. Also, our technique can be easily extended to efficiently incrementally check XFD satisfaction.

## 1 Introduction

The eXtensible Markup Language (XML) [5] has recently emerged as a standard for data representation and interchange on the Internet. 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]. Several different classes of integrity constraints for XML have been defined including key constraints [6], path constraints [8], and inclusion constraints [10, 11] and properties such as axiomatization and satisfiability have been investigated for these constraints. However, one topic that has been identified as an open problem in XML research [16] and which has been little investigated is how to extend the oldest and most well studied integrity constraint in relational databases, namely a *functional dependency* (FD), to XML and then how to develop a normalization theory for XML. This problem is not of just theoretical interest. The theory of FDs and normalization forms the cornerstone of practical relational database design and the development of a similar theory for XML will similarly lay the foundation for understanding how to design XML documents.

Recently, two approaches have been given for defining functional dependencies in XML (called XFDs). The first [1–3], proposed a definition based on the notion of a 'tree tuple' which in turn is based on the total unnesting of a relation [4]. More recently, we have proposed an alternative 'closest node' definition [14], which is based on paths and path instances that has similarity with the approach in [6] to defining keys in XML. This relationship between keys as defined in [6] and XFDs as defined in [14] extends further, as it was shown in [14] that in the

S. Bressan et al. (Eds.): XSym 2005, LNCS 3671, pp. 4–17, 2005.

case of simple paths, keys in XML are a special case of XFDs in the same way that keys in relational databases are a special case of FDs.

In general, the two approaches to defining XFDs are not comparable since they treat missing information in the XML document differently and the approach in [1–3] assumes the existence of a DTD whereas the approach in [14] does not. Howevever, we have recently shown that [15], in spite of the very different approaches used in [1–3] and [14], the two aproaches coincide for a large class of XML documents. In particular, we have shown that the definitions coincide for XML documents with no missing information conforming to a nonrecursve, disjunction free DTD. This class includes XML documents derived from complete relational databases using any 'non pathological' mapping. It has also been shown that in this situation, for mappings from a relation to an XML document defined by first mapping to a nested relation via an arbitrary sequence of nest and unnest operations, then followed my a direct mapping to XML, FDs in relations map to XFDs in XML. Hence there is a natural correspondence between FDs and XFDs.

In this paper we address the problem of developing an efficient algorithm for checking whether an XML document satisfies a set of XFDs as defined in [14]. We develop an algorithm which requires only one pass of the XML document and whose running time is linear in the size of the XML document and linear in the size of the number of XFDs. The algorithm uses an innovative method based on a multi level extension of extendible hashing. We also investigate the effect of the size on the number of paths on the l.h.s. of the XFD and show that the running time is both linear in the number of paths and also increases quite slowly with the number of paths.

Although the issue of developing checking the satisfaction of 'tree tuple' XFDs was not addressed in [1–3], testing satisfaction using the definitions in [1–3] directly is likely to be quite expensive. This is because there are three steps involved in the approach of [1–3]. The first is to generate a set of tuples from the total unnesting of an XML document. This set is likely to be much larger than the original XML document since unnesting generates all possible combinations amongst elements. The second step is to generate the set of tree tuples, since not all tuples generated from the total unnesting are 'tree tuples'. This is done by generating a special XML tree (document) from a tuple and checking if the document so generated is subsumed by the original XML tree (document). Once again this is likely to be an expensive procedure since it may require that the number of times the XML document is scanned is the same as the number of tuples in the total unnesting. In contrast, our method requires only one scan of the XML document. Finally, the definition in [1–3] requires scanning the set of tree tuples to check for satisfaction in a manner similar to ordinary FD satisfaction. This last step is common also to our approach.

The rest of this paper is organized as follows. Section 2 contains some preliminary definitions that we need before defining XFDs. We model an XML document as a tree as follows. In Section 3 the definition of an XFD is presented and the essential ideas of our algorithm are presented. Section 4 contains details of experiments that were performed to assess the efficiency of our approach and Section 5 contains concluding comments.

## 2 Preliminary Definitions

**Definition 1.** *Assume a countably infinite set* $\mathbf{E}$ *of element labels (tags), a countably infinite set* $\mathbf{A}$ *of attribute names and a symbol* $S$ *indicating text. An* XML tree *is defined to be* $T = (V, lab, ele, att, val, v_r)$ *where:*

1. $V$ *is a* finite *set of* nodes*;*
2. $lab$ *is a total function from* $V$ *to* $\mathbf{E} \cup \mathbf{A} \cup \{S\}$*;*
3. $ele$ *is a partial function from* $V$ *to a sequence of nodes in* $V$ *such that for any* $v \in V$*, if* $ele(v)$ *is defined then* $lab(v) \in \mathbf{E}$*;*
4. $att$ *is a partial function from* $V \times \mathbf{A}$ *to* $V$ *such that for any* $v \in V$ *and* $a \in \mathbf{A}$*, if* $att(v, a) = v_1$ *then* $lab(v) \in \mathbf{E}$ *and* $lab(v_1) = a$*;*
5. $val$ *is a function such that for any node in* $v \in V, val(v) = v$ *if* $lab(v) \in \mathbf{E}$ *and* $val(v)$ *is a string if either* $lab(v) = S$ *or* $lab(v) \in \mathbf{A}$*;*
6. *We extend the definition of* $val$ *to sets of nodes and if* $V_1 \subseteq V$*, then* $val(V_1)$ *is the set defined by* $val(V_1) = \{val(v) | v \in V_1\}$*;*
7. $v_r$ *is a distinguished node in* $V$ *called the root of* $T$*;*
8. *The parent-child edge relation on* $V$*,* $\{(v_1, v_2) | v_2$ *occurs in* $ele(v_1)$ *or* $v_2 = att(v_1, a)$ *for some* $a \in \mathbf{A}\}$ *is required to form a tree rooted at* $v_r$*;*

Also, the set of ancestors of a node $v \in V$ is denoted by $ancestor(v)$ and the parent of a node $v$ by $parent(v)$.

We now give some preliminary definitions related to paths.

**Definition 2.** *A* path *is an expression of the form* $l_1. \cdots .l_n$*,* $n \geq 1$*, where* $l_i \in \mathbf{E}$ *for* $1 <= i <= n-1$ *and* $l_n \in \mathbf{E} \cup \mathbf{A} \cup \{S\}$ *and* $l_1 = root$*. If* $p$ *is the path* $l_1. \cdots .l_n$ *then* $Last(p) = l_n$*.*

For instance, if $\mathbf{E} = \{$`root, Dept, Section, Emp`$\}$ and $\mathbf{A} = \{$`Project`$\}$ then `root`, `root.Dept` and `root.Dept.Section` are all paths.

**Definition 3.** *Let* $p$ *denote the path* $l_1. \cdots .l_n$*. The function* $Parent(p)$ *is the path* $l_1. \cdots .l_{n-1}$*. Let* $p$ *denote the path* $l_1. \cdots .l_n$ *and let* $q$ *denote the path* $q_1. \cdots . q_m$*. 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* $p \subset q$*, 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 but* $p_1 \cap p_2$*, to be the maximal common prefix of both paths. It is clear that the intersection of two paths is also a path.*

For instance, if $\mathbf{E} = \{$`root, Dept, Section, Emp`$\}$ and $\mathbf{A} = \{$`Project`$\}$ then `root.Dept` is a strict prefix of
`root.Dept.Section` and `root.Dept.Section.Emp` $\cap$
`root. Dept.Section.Project` $=$ `root.Dept.Section`.

**Definition 4.** *A* path instance *in an XML tree* $T = (V, lab, ele, att, val, v_r)$ *is a sequence* $v_1.\cdots.v_n$ *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* $v_1.\cdots.v_n$ *is said to be* defined over the path $l_1.\cdots.l_n$ *if for all* $v_i, 1 \leq i \leq n$, $lab(v_i) = l_i$*. Two path instances* $v_1.\cdots.v_n$ *and* $v'_1.\cdots.v'_n$ *are said to be* distinct *if* $v_i \neq v'_i$ *for some* $i$, $1 \leq i \leq n$*. The path instance* $v_1.\cdots.v_n$ *is said to be a* prefix *of* $v'_1.\cdots.v'_m$ *if* $n \leq m$ *and* $v_i = v'_i$ *for all* $i, 1 \leq i \leq n$*. The path instance* $v_1.\cdots.v_n$ *is said to be a* strict prefix *of* $v'_1.\cdots.v'_m$ *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* $Paths(p)$*.*

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 $v_r.v_1.v_3.v_4$

We now assume the existence of a *finite* set of legal paths $P$ for an XML application. Essentially, $P$ defines the semantics of an XML application in the same way that a set of relational schema define the semantics of a relational application. $P$ may be derived from the DTD, if one exists, or $P$ be derived from some other source which understands the semantics of the application if no DTD exists. In a sense we are assuming that XFDs and DTDs are orthogonal, in a similar fashion to that used in [6] where keys and DTDs are assumed to be orthogonal. We note that because of the restriction that $P$ is finite, if $P$ is derived from a DTD then the DTD must be non recursive. Next, we place the following restriction on the set of paths.

**Definition 5.** *A set* $P$ *of paths is* downward closed *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 downward closed.

We now define the notion of an XML tree conforming to a set of paths $P$.

**Definition 6.** *Let* $P$ *be a downward closed set of paths and let* $T$ *be an XML tree. Then* $T$ *is said to* conform *to* $P$ *if every path instance in* $T$ *is a path instance over a path in* $P$*.*

We note that if the set of paths is derived from a DTD, then requiring that the XML document conform to the set of paths is a much weaker condition than requiring that it conform to the DTD.

The next issue that arises in developing the machinery to define XFDs is the issue of missing information. This is addressed in [14] where missing nodes are considered and XFDs are defined using an extension of the strong satisfaction approach used in defining FD satisfaction in incomplete relations [4]. However, in this paper we take the simplifying assumption that there is no missing information in the XML tree. More precisely, we have the following definition.

**Definition 7.** *Let* $P$ *be a downward closed 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* $v_1.\cdots.v_n$ *defined over* $p_1$*, in* $T$*, then there exists a path instance* $v'_1.\cdots.v'_m$ *defined over* $p_2$ *in* $T$ *such that* $v_1.\cdots.v_n$ *is a prefix of the instance* $v'_1.\cdots.v'_m$*.*

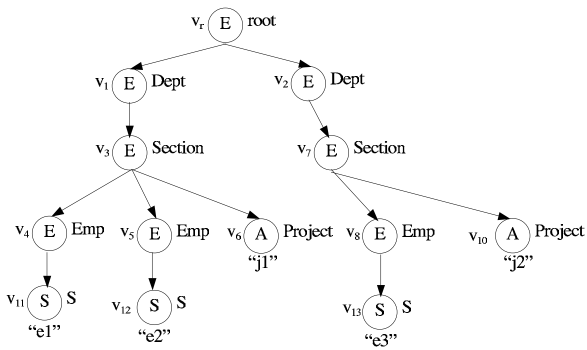


**Fig. 1.** A complete XML tree

For example, if we take $P$ to be {`root, root.Dept, root.Dept.Section, root.Dept.Section.Emp`, `root.Dept.Section.Emp.S root.Dept.Section.Project`} then the tree in Figure 1 conforms to $P$ and is complete.

One important comment to make on completeness is that if the set of paths is derived from a DTD and if we consider trees that conform to the DTD, and not just to $P$, then complete trees correspond only to disjunction free DTDs as shown in [3].

The next function returns all the final nodes of the path instances of a path $p$ in $T$.

**Definition 8.** *Let $P$ be a downward closed set of paths, let $T$ be an XML tree that conforms to $P$. The function $N(p)$, where $p \in P$, is the set of nodes defined by $N(p) = \{v | v_1. \cdots .v_n \in Paths(p) \wedge v = v_n\}$.*

For example, in Figure 1, $N(\texttt{root.Dept}) = \{v_1, v_2\}$.

We now need to define a function that returns a node and its ancestors.

**Definition 9.** *Let $P$ be a downward closed set of paths, let $T$ be an XML tree that conforms to $P$. The function $AAncestor(v)$, where $v \in V$, is the set of nodes in $T$ defined by $AAncestor(v) = v \cup Ancestor(v)$.*

For example in Figure 1, $AAncestor(v_3) = \{v_r, v_1, v_3\}$. The next function returns all nodes that are the final nodes of path instances of $p$ and are descendants of $v$.

**Definition 10.** *Let $P$ be a downward closed set of paths, let $T$ be an XML tree that conforms to $P$. The function $Nodes(v, p)$, where $v \in V$ and $p \in P$, is the set of nodes in $T$ defined by $Nodes(v, p) = \{x | x \in N(p) \wedge v \in AAncestor(x)\}$*

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

## 3 Checking XFDs

We firstly recall the definition of an XFD from [14], restricted to the situation where the XML document is complete.

**Definition 11.** *Let $P$ be a set of downward closed paths and let $T$ be a complete XML tree that conforms to $P$. An XML functional dependency (XFD) is a statement of the form: $p_1, \ldots, p_k \to q$, $k \geq 1$, where $p_1, \ldots, p_k$ and $q$ are paths in $P$. $T$ satisfies the XFD if there exists $p_i$, for some $i, 1 \leq i \leq k$, such that $p_i = q$ or whenever there exists two distinct path instances $v_1.\cdots.v_n$ and $v'_1.\cdots.v'_n$ defined over $q$ in $T$ such that $val(v_n) \neq val(v'_n)$, then $\exists i, 1 \leq i \leq k$, such that $val(Nodes(x_i, p_i)) \cap val(Nodes(y_i, p_i)) = \emptyset$, where: $x_i = \{v | v \in AAncestor(v_n) \wedge v \in N(p_i \cap q)\}$ and $y_i = \{v | v \in AAncestor(v'_n) \wedge v \in N(p_i \cap q)\}$.*

We now illustrate the definition by an example.

*Example 1.* Consider the XFD
`root.publication.publisher.S` $\to$ `root.publication.title` in Figure 2. Then $v_4 \in N($`root.publication.title`$)$ and $v_6 \in N($`root.publication.title`$)$ and $val(v_4) =$ `"t1"` $\neq val(v_6) =$ `"t2"`.
So `root.publication.title`$\cap$ `root.publication.publisher.S` $=$ `root.publication` and so $N($`root.publication`$) = \{v_1, v_2\}$. Thus $x_{1_1} = v_1$ and $y_{1_1} = v_2$ and so $Nodes(x_{1_1},$`root.publication.publisher.S`$) = v_{17}$ and thus $val(Nodes(x_{1_1},$`root.publication.publisher.S`$)) = \{$`"p1"`$\}$. Also, $Nodes(y_{1_1},$`root.publication.publisher.S`$) = v_{19}$ and so $val(Nodes(y_{1_1},$`root.publication.publisher.S`$)) = \{$`"p1"`$\}$ and so the XFD `root.publication.publisher.S` $\to$ `root.publication.title` is violated because $val(Nodes(x_{1_1},$`root.publication.publisher.S`$))$ $\cap\ val(Nodes(y_{1_1},$`root.publication.publisher.S`$)) \neq \emptyset$. We note that if the $val$ of node $v_4$ was changed to `"t2"` then the XFD would be satisfied.

Consider next the XFD `root.publication.title` $\to$ `root.publication.publisher.S`. The only nodes in $N($`root.publication.publisher.S`$)$ are $v_{17}$ and $v_{19}$ and $val(v_{17}) =$ `"p1"` and $val(v_{19}) =$ `"p1"` and so the XFD `root.publication.title` $\to$ `root.publication.publisher.S` is satisfied.

We now present an algorithm for checking whether an XML document satisfies a set of XFDs. The algorithm has two major steps. The first step is to produce what we call tuples. The second step is to hash the tuples to check if the document satisfies the XFDs.

### 3.1 Tuple Generation

We start with the definition of some terms.

**Definition 12 (Relevant Path).** Given a set $\Sigma$ of FDs $\{f_1, \ldots, f_m\}$ we use $relev(\Sigma)$, called the set of *relevant paths*, to denote the list of *distinct* paths defined as the following:

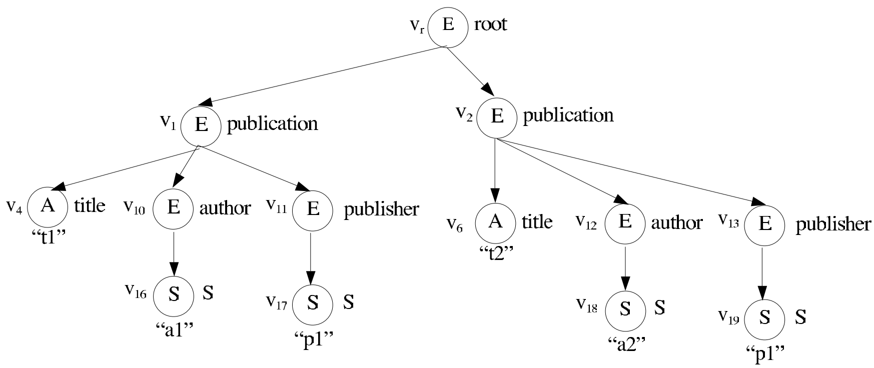


**Fig. 2.** An XML tree

- all paths involved in $\Sigma$, including those on the LHS of the XFDs and also those on the RHS;
- if $p_1, p_2 \in relev(\Sigma)$ then $p_1 \cap p_2 \in relev(\Sigma)$;
- the order of the paths and path intersections in the list agrees with the order of their appearances in documents.

Consider the example in Figure 3.
Let $\Sigma = \{root.A, root.A.B \rightarrow root.A.G.C,\ root.A.G.C, root.A.G.D \rightarrow root.A.B\}$. Then $relev(\Sigma) = [A, B, G, C, D]$. Note that for simplicity, we abbreviate paths by their end labels, which will not introduce confusion in the presentation.

We further use $pathroot(\Sigma)$, called the *path root*, to mean the shortest path in $relev(\Sigma)$. We call a subtree rooted at a node labelled by $pathroot(\Sigma)$ a *relevant* tree. We call the nodes in a relevant tree labelled by the end labels of the paths in $relev(\Sigma)$ *relevant nodes*. Given a relevant node $v$, $path(v)$ is the path on the path instance reaching $v$. Given a path $p \in relev(\Sigma)$, $posi(p)$ is the sequential number of $p$ in $relev(p)$ and if $p$ is the first element in $relev(\Sigma)$, then $posi(p)$ is 1.

In Figure 3, $pathroot(\Sigma) = root.A$. The subtree rooted at $v_1$ is relevant tree. All nodes labelled by $A, B, C, D, G$ are relevant nodes. $posi(root.A.G.D) = 5$ and $posi(root.A) = 1$, $path(v_4) = root.A.G$.

The concept *tuple* defined following is an important construct used to model the result of document parsing.

**Definition 13 (Tuple).** Given a set $\Sigma$ of XFDs and a relevant tree $bT$, a tuple $t$ of $bT$ over $relev(\Sigma)$ is defined as $t =< val_1, ..., val_n >$ where $n$ is the number of paths in $relev(\Sigma)$ and for each $i$ in $[1, ..., n]$, $p_i \in relev(\Sigma) \wedge val_i = val(v_u) \wedge v_u \in bT \wedge lab(v_u) = last(p_i)$.

We define the following terms to be used to indicate the directions of parsing in relation to the paths in $relev(\Sigma)$.

tuples

| A | B | G | C | D |
|---|---|---|---|---|
| $v_1$ | b1 | $v_4$ | c1 | d1 |
| $v_1$ | b2 | $v_4$ | c1 | d1 |
| $v_1$ | b1 | $v_4$ | c1 | d2 |
| $v_1$ | b2 | $v_4$ | c1 | d2 |
| $v_1$ | b1 | $v_5$ | c2 | d3 |
| $v_1$ | b2 | $v_5$ | c2 | d3 |

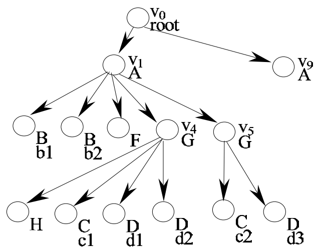


**Fig. 3.** An XML tree and its tuples

**Definition 14.** Let $v_l$ be the last visited relevant node and $v$ be the current visited node. Then:

- $v$ is called a *down node* if $posi(path(v)) > posi(path(v_l))$;
- $v$ is called a *up node* if $posi(path(v)) < posi(path(v_l))$;
- $v$ is called a *across node* if $posi(path(v)) =$ $posi(path(v_l))$.

Note that in this definition, the directions, *down*, *up*, and *across*, are defined relative to the order in $relev(\Sigma)$, not the directional positions in a tree. This is important because during parsing, we do not care about irrelevant nodes but only concentrate on relevant nodes.

We now propose the parsing algorithm. The algorithm reads text from a document and generates the tuples for a set of XFDs. After a line of text is read, the algorithm tokenizes the line into tokens of tags and text strings. If a token is a tag of a relevant path, then the parsing direction is determined. If the direction is downward, content of the element will be read and put into the current tuple. If it is across, new tuples are created. In the algorithm, there are two variables $openTuple$ and $oldOpenTuple$ used to deal with multiple occurrences of a node. For example in Figure 3, there are multiple $B$ nodes. Multiple tuples need to be created so that each occurrence can be combined with values from other relevant nodes like $C$ nodes and $D$ nodes. In the algorithm, we discuss only elements but not attributes. Attributes are only specially cases of elements when parsed and the algorithm can be easily adapted to attributes.

```
Algorithm 1
INPUT: An XML document T and relev(Σ)
OUTPUT: a set of tuples
Let lastPosi = 1, curPosi = 1,
    openTuple = 1, lastOpenTuple = 1
Let reading will read and tokenize input to one of the
following tokens: start tags, closing tags, and texts
Foreach token in T in order,
   if token is text: set token as value to the
      position curPosi of the last openTuple
      of tuples
   let curPosi = posi(tag)
   if curPosi = 0 (NOT relevant): next token
   if token is a closing tag
      if current is the last in relev(Σ)
         openTuple = oldOpenTuple = 1
      next noken;
   if curPosi > lastPosi (down)
      lastOpenTuple = openTuple
      lastPosi = curPosi,  next token
   if curPosi = lastPosi (across)
      create oldOpenTuple new tuples
      copy the first lastPosi − 1 values from
      the previosu tuple to the new tuples
      openTuple = openTuple + lastOpenTuple
      next token
   if curPosi < lastPosi (up)
      lastPosi = curPosi, next token
end foreach
```

**Observation 1:** The time for the above algorithm to generate tuples is linear in the size of the document.

### 3.2 Hashing and Adaption

Once we get the tuples, we use hashing to check if the XFDs are satisfied by the document which is now represented by the tuples. Hashing is done for each XFD. In other words, if there are $m$ XFDs, $m$ hash tables will be used. Let $Tup$ be the tuples generated by Algorithm 1. We project tuples in $Tup$ onto the paths of an XFD $f := \{p_1, ..., p_n\} \rightarrow q$ to get a projected bag of tuples denoted by $Tup(f)$. For each tuple $t$ in $Tup(f)$, $f(p)$ denotes the projection $t[p_1, ..., p_n]$ and $f(q)$ denotes $t[q]$. Then a hash table is created for each XFD as follows.

The hash key of each hash table is $f(p)$ and the hash value is $f(q)$. When two tuples with the same $f(p)$ but different $f(q)$s are hashed into a bucket, the XFD is violated. This criteria corresponds exactly to the definition of an XFD but with the condition that there is no collision.

We define a collision to be the situation where two tuples get the same hash code which puts the two tuples in the same bucket. Based on the criteria above, this means that the two tuples make the XFD violated but in fact they do not. For example, if the two tuples for $< f(p), f(q) >$ are $< 10...0, 1 >$ and $< 20...0, 2 >$ where '...' represent 1000 zeros. If a hash function is the modular operator with the modular being 1 million indicating there are 1 million buckets in the hash table, then the two tuples will be put into the same bucket of the hash table which indicate that the XFD is not satisfied based on the criteria presented above. However, the tuples satisfy the XFD. With normal extendible hashing the traditional solution to this problem is to double the size of the hash table, but this means that memory space can be quickly doubled while the two tuple are still colliding. In fact with only two tuples that collide, we can exhaust memory, no matter how large, if there is no appropriate collision handling mechanism.

With our implementation, we use two types of collision handling mechanisms. The first one is doubling the size of the hash table. As discussed above, this only works for a limited number of cases. The second technique is used if the table size cannot be doubled. The second method involves the use of overflow buckets and is illustrated in Figure 4.

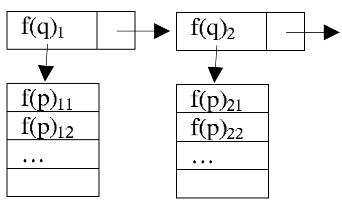


**Fig. 4.** Bucket structure of hash table

In the figure, a bucket has a section, denoted by $bask(q)$, to store $f(q)$ and a downward list, denoted by $bask(p)$, to store $f(p)$'s if there are multiple tuples having the same $f(q)$ but different $f(p)$'s because of a collision. It is also possible that multiple tuples having different $f(q)$'s come into the same bucket, as we discussed before, because of a collision. In this case, these tuples are stored, based on their $f(q)$ values, in the extended buckets which are also buckets connected to the main bucket horizontally.

With this extension, the following algorithm is used to check if the XFDs is satisfied.

The performance of the algorithm is basically linear. There is a cost to run the "bucket loop" in the algorithm. However, the cost really depends on the number of collisions. From our experiments, we observed that collision occurred, but the number of buckets involved in collisions is very low. At the same time, more collisions means a higher probability of violating the XFDs.

**Algorithm 2**
**INPUT:** A set $Tup(f)$ of tuple for XFD $f$
**OUTPUT:** true or false

```
Set the hash table size to the maximum allowed by
   the computer memory
Foreach t in Tup(f)
  let code = hashFunction(f(p))
  set current bucket to bucket code
  bucket loop
    if bask(q) = f(q), insert f(p) in to
       bask(p) else if it is not in it
       exit the bucket loop
    if bask(q)! = f(q), check to see if
       f(p) is in bask(p),
       if yes, exit algorithm with false,
       if not, let the current bucket be
          the next extended bucket
          go to the beginning of the
          bucket loop
  end bucket loop
end foreach
return true
```

## 4 Experiments

In this section we report on experiments with the algorithms presented in the previous section. All the experiments were run on 1.5GHz Pentium 4 machine with 256MB memory. We used the DTD given in Example 5 and artificially generated XML documents of various sizes in which the XFD was satisfied, the worst case situation for running time as in such a case all the tuples need to be checked. When documents were generated, multiple occurrences of the nodes with the same labels at all levels were considered. Also, the XFDs were defined involving paths at many levels and at deep levels.

In the first experiment, we checked the satisfaction of one XFD and fixed the number of paths on the left hand side of the XFD to 3. We varied the size of the XML document and recorded the CPU time required to check the document for the satisfaction of one XFD. The results of this experiment are shown in Figure 6. These results indicate that the running time of the algorithm is essentially linear in the number of tuples. This is to be expected as the time to perform the checking of an XFD is basically the time required to read and parse the XML document once, which is linear in the size of the document and to hash the tuples into the hash table which again is linear.

In the second experiment, we limited ourselves to only one XFD, fixed the number of tuples in the XML document to 100,000 (and so the size of the document was also fixed), but varied the number of paths on the left hand side of the XFD. The results are shown in Figure 7. The figure shows that again the time is linear in relation to the number of paths. This is also to be expected

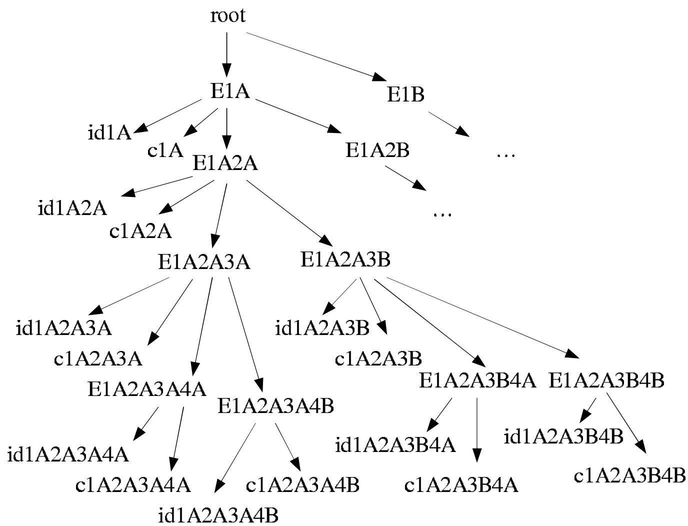


**Fig. 5.** Implementation DTD

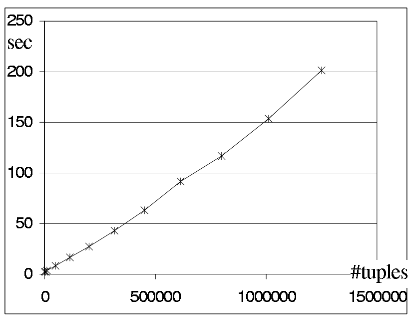


**Fig. 6.** The number of tuples vs checking time (in seconds)

because the number of paths in a XFD only increases the length of a tuple, but does not require any change to other control structures of the algorithm and therefore the times for reading, parsing, and checking are all kept linear. It is the increase of tuple length that caused the slight increase in processing time and this increase is slow and linear.

In the third experiment, we fixed the number of paths on the left hand side of a XFD to 3 and also fixed the file size and the number of tuples, but varied the number of XFDs to be checked. The result is shown in Figure 8. This result shows that the time is linear in the number of XFDs to be checked, but the increase is steeper than that of Figure 7. This is caused by the way we do the checking. In the previous section, we said that for each XFD, we create a hash table. However, for a very large number of XFDs this requires too much memory

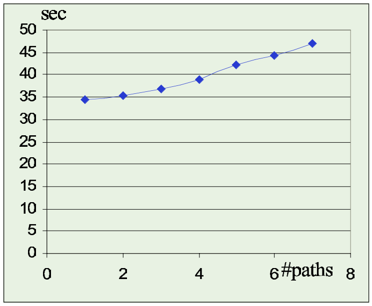


**Fig. 7.** Number of paths in the left and side of an XFD vs checking time

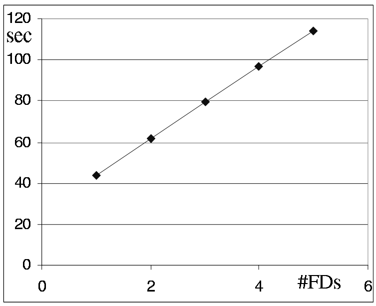


**Fig. 8.** Number of XFDs to be checked vs checking time

so in this experiment, we created one hash table, checked one XFD, and then used the same hash table to check the second XFD. Thus the time consumed is the addition of the times for checking these XFDs separately. The benefit of this algorithm is that parsing time is saved. Parsing time, based on our experience, is a little more than the time for hashing. Furthermore, the performance of the third experiment can be improved if a computer with bigger memory is used.

## 5 Conclusions

In this paper we have addressed the problem of developing an efficient algorithm for checking the satisfaction of XFDs, a new type of XML constraint that has recently been introduced [14]. We have developed a novel hash based algorithm that requires only one scan of the XML document and its running time is linear in the size of the XML document and linear in the number of XFDs. Also, our algorithms can be used to efficiently incrementally check an XML document.

There are several are other extensions to the work in this paper that we intend to conduct in the future. The first is to extend the algorithm to the case where there is missing information in the XML document as defined in [14]. The second is to extend the approach to the checking of multivalued dependencies in XML, another new XML constraint that has recently been introduced [12, 13].

## References


1. M. Arenas and L. Libkin. A normal form for XML documents. In *Proc. ACM PODS Conference*, pages 85–96, 2002.
2. M. Arenas and L. Libkin. An information-theoretic approach to normal forms for relational and XML data. In *Proc. ACM PODS Conference*, pages 15–26, 2003.
3. M. Arenas and L. Libkin. A normal form for XML documents. *TODS*, 29(1):195 – 232, 2004.
4. P. Atzeni and V. DeAntonellis. *Foundations of databases*. Benjamin Cummings, 1993.
5. T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical report, http://www.w3.org/Tr/1998/REC-XML-19980819, 1998.
6. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. *Information Systems*, 28(8):1037–1063, 2003.
7. P. Buneman, W. Fan, J. Simeon, and S. Weinstein. Constraints for semistructured data and XML. *ACM SIGMOD Record*, 30(1):45–47, 2001.
8. P. Buneman, W. Fan, and S. Weinstein. Path constraints on structured and semistructured data. In *Proc. ACM PODS Conference*, pages 129 – 138, 1998.
9. Y. Chen, S. Davidson, and Y. Zheng. Xkvalidator: a constraint validator for xml. In *CIKM*, pages 446–452, 2002.
10. W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3):368 – 406, 2002.
11. W. Fan and J. Simeon. Integrity constraints for XML. *Journal of Computer and System Sciences*, 66(1):254–291, 2003.
12. M.W. Vincent and J. Liu. Multivalued dependencies and a 4NF for XML. In *15th International Conference on Advanced Information Systems Engineering (CAISE)*, pages 14–29. Lecture Notes in Computer Science 2681 Springer, 2003.
13. M.W. Vincent, J. Liu, and C. Liu. Multivalued dependencies and a redundancy free 4NF for XML. In *International XML database symposium (Xsym)*, pages 254–266. Lecture Notes in Computer Science 2824 Springer, 2003.
14. M.W. Vincent, J. Liu, and C. Liu. Strong functional dependencies and their application to normal forms in XML. *ACM Transactions on Database Systems*, 29(3):445–462, 2004.
15. M.W. Vincent, J. Liu, C. Liu, and M.Mohania. On the definition of functional dependencies in XML. In *submitted to ACM Transactions on Internet Technology*, 2004.
16. J. Widom. Data management for XML - research directions. *IEEE data Engineering Bulletin*, 22(3):44–52, 1999.