On the Performances of Checking XML Key and Functional Dependency Satisfactions*

Md. Sumon Shahriar and Jixue Liu

Data and Web Engineering Lab
School of Computer and Information Science
University of South Australia, SA-5095, Australia
shamy022@students.unisa.edu.au, jixue.liu@unisa.edu.au

Abstract. We propose an implementation and analyze the performances of checking XML key and functional dependency (XFD) satisfaction. The work is based on the definitions of XML key and XFD we proposed for the ordered XML model. It investigates how the number of occurrences of elements in the document and the number of paths in the constraints affect the behavior of XML key and XFD satisfaction checking. The results of the study show that both XML key and XFD can be checked in linear time with the number of tuples and with the number of paths involved in key and XFD. Also, XFD can be checked linearly with the number of XFDs.

1 Introduction

XML[13] is widely used as a data representation and storage format over the web. It is also used in many data intensive activities such as data integration[1,2], data warehousing[3], data exchange, data translation[4], data publishing[5]. As the use of XML has increased immensely in recent years, the research for XML in the context of database perspective has been getting much attention. One such area is the development of constraint mechanisms for XML[6,7].

There are many proposals on XML constraints such as XML keys(see [26] for an overview of proposals) and XML functional dependencies [21,15,16,22,23,24]. Buneman et al. [8] presented the concept of keys for XML documents where wild cards are allowed in the target path, but not allowed in the key paths. Besides some inference rules of keys, they proposed the concepts of absolute keys and relative keys. At the same time, they discussed two types of satisfactions: strong key satisfaction and weak key satisfaction. They also showed the expressiveness of their proposal for keys over the XML Schema[14]. Later Buneman et al.[9] presented the reasoning (satisfiability and axiomatization) of their proposed notion of XML keys. Fan et al.[10] proposed an extended XML key notation which allows wildcards, especially the upward wildcards in the key paths. A further work [11] of Fan and Simeon studied integrity constraints for XML like keys,

* This research is supported with Australian Research Council(ARC) Discovery Project(DP 0559202) Fund.

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part II, LNCS 5871, pp. 1254–1271, 2009.

foreign keys, inclusion constraints, inverse constraints. Another research work by Fan and Libkin[12] showed the analysis of the implication problems of various XML integrity constraints[11] in the presence of DTDs with the main focus being the implication problem of keys and foreign keys. In [7], W. Fan reviewed the XML constraints with specifications, analysis, and applications. Recently, S. Hartmann et al.[25] investigated the structural key in XML.

With regard to XML functional dependencies, Arenas and Libkin[21] showed the functional dependencies for XML in terms of tree tuples. Millist et al.[22] proposed their notion of functional dependencies for complete XML documents. Another paper[24] defined the functional dependencies for XML considered restricted features of DTDs. Liu et al.[23] showed the notion of XML functional dependency that actually subsumes the functional dependencies for relational database. In another paper[15], Liu et al. presented the notion XML functional dependencies considering the functional dependencies for incomplete relations and categorized the functional dependencies of XML into two types: global functional dependencies(GXFDs)[15] and local functional dependencies(LXFDs)[16].

Recently, we proposed XML key [27] for an ordered XML model. This proposal is necessary because if we use the key definition of [8,9] against the ordered model, incorrect results would be generated. Our definition uses $\#PCDATA$ ended paths as key paths(also called fields). The importance of using $\#PCDATA$ ended paths as key fields is two folded. *Firstly*, by doing so, the key field values are explicit texts which can be specified by users. This is in contrast to using node identifiers as values [8,9]. *Secondly*, a tree can have only one node (the root) and thus may have no text values. Using $\#PCDATA$ ended paths prevents tuples from having no text values. In other words, it avoids null values for key fields and therefore avoids unclear semantics. The satisfaction of our key definition is between the strong key satisfaction and the weak key satisfaction defined in [8]. Strong key[14] definition allows only one tuple of fields in the key to appear under each target node. While the weak key definition allows multiple tuples for the key fields under each target node and some of these tuples can be the same (duplicate) but tuples between different target nodes must not have common values for all fields. Our definition is between the two definitions in the sense that we allow multiple tuples, but all tuples of the key fields must be distinct in the target and in the whole tree. We then extended our key definition to XFD in [28] for the same ordered XML model.

In this paper, we study the implementation and performance of checking XML key and XFD proposed in [27,28]. The study is motivated by the fact that like in the relational database, the implementation of XML keys and XFDs is critical to the quality of data in the XML database. Every time when there is a new instance for the database, we like to check the constraints against the new instance to ensure proper data is added or removed from the database. At the same time, the performance of the implementation is important to the efficiency of the database. Different ways of implementing the same mechanism will result in different performances. To a database management system, the efficiency of all processes is always critical to the success of the system.

In the literature, the checking of XML key and foreign keys using SAX was studied in [19] based on the proposal presented in [8]. An indexing based on paths on key is used in checking and the performance was shown as linear. Another study in [20] showed the XML key satisfaction checking using XPath based on DOM[29]. The study showed the checking of XML key can be done in polynomial time. We also use DOM(contrasting the use of SAX in [19]) for parsing XML document, but our implementation is different from the studies [19,20] because we use a novel method of pairing the close values of elements to capture the correct semantics in tuple generation while parsing the document.

In case of XFD satisfaction checking, Liu et al. conducted performance studies of checking XFDs [17] and checking XML multiple value dependencies [18]. The work in this paper is different from both [17] and [18] in that the data model assumed in this paper allows repeated occurrences of a nested type structure while this is not assumed in [17] and [18]. We use an example to explain repeated occurrences of a nested type structure. The DTD `<!ELEMENT multi-choice-questions (question,(choice,is-answer)*)*>` allows the nested type structure `(choice, is-answer)` to repeat for multiple times for each question in a document conforming to this DTD. Because of the allowance of repetition of a nested structure, the parsing of a document for values relating to checking of a key and an XFD becomes different and the time spent on this part of the performance analysis counts a great amount of the overall time spent on checking, as shown later. Because of this fundamental difference in assumed data models, this work is different from [17] and [18].

Our paper is organized as follows. We give the basic definitions and notation in Section 2. The algorithms for tuple generation and its implementation are given in Section 3. We present the study on checking XML key satisfaction in Section 4. In Section 5, we also study the checking the XFD satisfaction. We conclude with some remarks in Section 6.

2 Basic Definitions

We now review some basic definitions proposed in [27,28] which are critical to guarantee the self-containment of the paper. Before defining XML key, we introduce the notation for DTD and paths on DTD using examples.

A DTD is defined in our notation as $D = (EN, \beta, \rho)$ where EN contains element names, ρ is the root of the DTD and β is the function defining the types of elements. For example, the DTD D in Fig.1(a) is represented in our notation as $\beta(root) = [A]^+$, $\beta(A) = [B^+{}_\times[M^+{}_\times E^+]]^+$, $\beta(M) = [C^*{}_\times D^*]^+$, $\beta(B) = \beta(C) = \beta(D) = \beta(E) = Str$ where Str is $\#PCDATA$, $EN = \{root, A, B, C, D, E, M, Str\}$, and $\rho = root$. An element name and a pair of squared brackets '[ ]' each, with its multiplicity, is called a *component*. For example, $[A]^+$, $[B^+{}_\times[M^+{}_\times E^+]]^+$ are two components. A conjunctive or disjunctive sequence of components, often denoted by g, is called a *structure*. For example, the structure $g_1 = B^+{}_\times[M^+{}_\times E^+]$ is a conjunctive sequence and $g_1 = [A|B]^+$ is a disjunctive sequence. A structure is further decomposed into *substructures* such as g_1 is decomposed into g_a and g_b where $g_a = B^+$ and $g_b = [M^+{}_\times E^+]$.

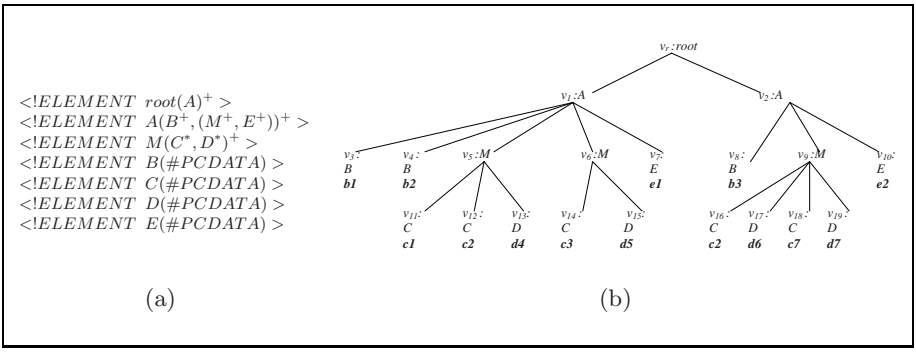

Fig. 1. (a) An XML DTD D and (b)An XML document T

Now we define paths on the DTD. In Fig.1(a), we say $A/M/C$ is a *simple path* and $root/A/M/C$ is a *complete path*. A complete path starts with the root element of the DTD. The function $beg(A/M/C)$ returns A, $last(A/M/C)$ returns C and $par(M)$ returns A.

Now we are ready to define the XML key.

Definition 1 (XML Key). *Given a DTD D = (EN,β, ρ), an XML key on D is defined as* $\Bbbk(Q, \{P_1, \cdots, P_l\})$*, where* $l \geq 0$*, Q is a complete path called the* **selector***, and* $\{P_1, \cdots, P_i, \cdots, P_l\}$ *(often denoted by P) is a set of* **fields** *where each* P_i *is defined as:*

(a) $P_i = \wp_{i1} \cup \cdots \cup \wp_{in_i}$*, where "∪" means disjunction and* $\wp_{ij}$ $(j \in [1, \cdots, n_i])$ *is a simple path on D, and* $\beta(last(\wp_{ij})) = Str$*, and* $\wp_{ij}$ *has the following syntax:*
$\wp_{ij} = seq$
$seq = e \mid e/seq$ *where* $e \in EN$*;*
(b) $Q/\wp_{ij}$ *is a complete path.* □

For example, $\Bbbk(root/A, \{M/C, M/D\})$ is a key on the DTD D in the Fig.1. The selector $root/A$ is a complete path, M/C and M/D are simple paths, $root/A/M/C$ and $root/A/M/D$ are also complete paths, $\beta(last(M/C)) = Str$ and $\beta(last(M/D)) = Str$. We say $\Bbbk$ is *valid* on D as it follows the definition 1.

Before defining XML key satisfaction, we introduce some definitions and notation using examples.

An XML document is represented as an XML tree $T = (v : e\ (T_1 T_2 \cdots T_f))$ if element e encloses other elements or $T = (v : e : txt)$ if e encloses the string value txt where v is the node identifier which is omitted when the context is clear, e is the element name labeled on the node, $T_1 \cdots T_f$ are subtrees. For example, in Fig.1(b), the document T is represented as $T_{v_r} = (v_r : root(T_{v_1} T_{v_2}))$. Then $T_{v_1} = (v_1 : A(T_{v_3} T_{v_4} T_{v_5} T_{v_6} T_{v_7}))$, $T_{v_2} = (v_2 : A(T_{v_8} T_{v_9} T_{v_{10}}))$, $T_{v_3} = (v_3 : B : b1), T_{v_4} = (v_4 : B : b2), \cdots$ Other subtrees can be expressed in the same way. We say $T_{v_{12}} =_v T_{v_{16}}$ where $T_{v_{12}} = (v_{12} : C : c2)$ and $T_{v_{16}} = (v_{16} : C : c2)$.

Now we give an example to show the important concept *hedge* which is a sequence of adjacent subtrees for a type structure. Consider the structure $g_1 = [C^* \times D^*]^+$ in Fig.1(a). The trees $T_{v_{11}} T_{v_{12}} T_{v_{13}}$ form a hedge conforming to g_1 under node v_5, the trees $T_{v_{14}} T_{v_{15}}$ form a hedge under node v_6, and the trees $T_{v_{16}} T_{v_{17}} T_{v_{18}} T_{v_{19}}$ form a hedge under v_9. However, when we consider $g_2 = [C^* \times D^*]$ (without $^+$ compared to g_1), there are two sequence conforming to g_2 for node v_9: $T_{v_{16}} T_{v_{17}}$ and $T_{v_{18}} T_{v_{19}}$. To reference various structures and their conforming sequences, we introduce the concept *hedge*, denoted by H^g, which is a sequence of trees conforming to the structure g.

We now give the formal definition of hedge.

Definition 2 (Hedge). *A hedge H is a sequence of adjacent primary sub trees $T_1 T_2 \cdots T_n$ of the same node that conforms to a specific type construct g.* □

Now we introduce two concepts *minimal structure* and *minimal hedge*. A minimal structure of one or more elements is the structure that is encompassed within '[]' bracket containing those elements. Consider $\beta(A) = [B^+ \times [M^+ \times E^+]]^+$ for D in Fig.1(a). The minimal structure of B and E is $g_3 = [B^+ \times [M^+ \times E^+]]$ meaning that both elements B and E are encompassed within the outermost '[]' bracket. Thus the minimal hedge conforming to g_3 is $H_1^{g_3} = T_{v_3} T_{v_4} T_{v_5} T_{v_6} T_{v_7}$ for node v_1 and $H_2^{g_3} = T_{v_8} T_{v_9} T_{v_{10}}$ for node v_2 in the document T in Fig.1(b). But the minimal structure of C and D is $g_2 = [C^* \times D^*]$. So the the minimal hedges conforming to g_2 are $H_1^{g_2} = T_{v_{11}} T_{v_{12}} T_{v_{13}}$ for node v_5, $H_2^{g_2} = T_{v_{14}} T_{v_{15}}$ for node v_6, $H_3^{g_2} = T_{v_{16}} T_{v_{17}}$ and $H_4^{g_2} = T_{v_{18}} T_{v_{19}}$ for node v_9 in T.

We now give the formal definitions of minimal structure and minimal hedge.

Definition 3 (Minimal Structure). *Given a DTD definition $\beta(e)$ and two elements e_1 and e_2 in $\beta(e)$, the minimal structure g of e_1 and e_2 in $\beta(e)$ is the pair of brackets that encloses e_1 and e_2 and any other structure in g does not enclose both.* □

Definition 4 (Minimal Hedge). *Given a hedge H of $\beta(e)$, a minimal hedge of e_1 and e_2 is one of H^gs in H.* □

We then use minimal structure and minimal hedge to produce *tuple* for the paths in P(**P-tuple**). Consider an XML key $\Bbbk(root/A, \{B, M/C, M/D\})$ on the DTD D in Fig.1(a). Here, the selector path is $root/A$ and the fields are B, M/C and M/D. The minimal structure for element names B, M/C and M/D is $g_4 = [B^+ \times [M^+ \times E^+]]$. The minimal hedges for g_4 are $H_1^{g_4} = T_{v_3} T_{v_4} T_{v_5} T_{v_6} T_{v_7}$ under node v_1 and $H_2^{g_4} = T_{v_8} T_{v_9} T_{v_{10}}$ under node v_2 in T. The P-tuples for the first hedge $H_1^{g_4}$ are $F_1 = (T_{v_3} T_{v_{11}} T_{v_{13}}) = ((v_3 : B : b1)(v_{11} : C : c1)(v_{13} : D : d4))$, $F_2 = (T_{v_3} T_{v_{12}} T_{v_{13}}) = ((v_3 : B : b1)(v_{12} : C : c2)(v_{13} : D : d4))$, $F_3 = (T_{v_3} T_{v_{14}} T_{v_{15}}) = ((v_3 : B : b1)(v_{14} : C : c3)(v_{15} : D : d5))$, $F_4 = (T_{v_4} T_{v_{11}} T_{v_{13}}) = ((v_4 : B : b2)(v_{11} : C : c1)(v_{13} : D : d4))$, $F_5 = (T_{v_4} T_{v_{12}} T_{v_{13}}) = ((v_4 : B : b2)(v_{12} : C : c2)(v_{13} : D : d4))$ and $F_6 = (T_{v_4} T_{v_{14}} T_{v_{15}}) = ((v_4 : B : b2)(v_{14} : C : c3)(v_{15} : D : d5))$. Similarly, P-tuples for the hedge $H_2^{g_4}$ are $F_7 = (T_{v_8} T_{v_{16}} T_{v_{17}}) = ((v_8 : B : b3)(v_{16} : C : c2)(v_{17} : D : d6))$ and $F_8 = (T_{v_8} T_{v_{18}} T_{v_{19}}) = ((v_8 : B : b3)(v_{18} : C : c7)(v_{19} : D : d7))$.

We define some additional notation. T^e means a tree rooted at a node labeled by the element name e. Given path $e_1/\cdots/e_m$, we use $(v_1 : e_1).\cdots.(v_{m-1} : e_{m-1}).T^{e_m}$ to mean the tree T^{e_m} with its ancestor nodes in sequence, called the *prefixed tree* or the *prefixed format* of T^{e_m}. Given path $\wp = e_1/\cdots/e_m$, $T^{\wp} = (v_1 : e_1).\cdots.(v_{m-1} : e_{m-1}).T^{e_m}$. $\langle T^{\wp}\rangle$ is the set of all $T^{\wp}$ and $\langle T^{\wp}\rangle = \{T_1^{\wp}, \cdots, T_f^{\wp}\}$. $|\langle T^{\wp}\rangle|$ returns the number of $T^{\wp}$ in $\langle T^{\wp}\rangle$. Because $P_i = \wp_{i1} \cup \cdots \cup \wp_{in_i}$, we use $\langle T^{P_i}\rangle$ to mean all $T^{\wp_{ij}}$s and $T^{P_i} = T^{\wp_i}$ to mean one of $T^{\wp_{ij}}$s. We use $T^{\wp_i} \in T^Q$ to mean that $T^{\wp_i}$ is a sub tree of T^Q. Similarly, $\langle T^{P_i}\rangle \in T^Q$ means that all trees T^{P_i} are sub trees of T^Q.

We now give the formal definition of P-tuple.

Definition 5 (P-tuple). *Given a path Q and a set of relative paths $\{P_1, ..., P_l\}$ and a tree T^Q. A P-tuple under T^Q is a sequence of pair-wise-close subtrees $(T^{P_1}\cdots T^{P_l})$ where 'pair-wise-close' is defined next.*

Let $\wp_i = e_1/\cdots/e_k/e_{k+1}/\cdots/e_m \in P_i$ and $\wp_j = e'_1/\cdots/e'_k/e'_{k+1}/\cdots/e'_n \in P_j$ be two key paths for any P_i and P_j. Let $prec(T^{P_i}) = (v_1 : e_1).\cdots.(v_k : e_k).(v_{k+1} : e_{k+1}).\cdots.(v_m : e_m)$ and $prec(T^{P_j}) = (v'_1 : e'_1).\cdots.(v'_k : e'_k).(v'_{k+1} : e'_{k+1}).\cdots..(v'_m : e'_m)$. T^{P_i} and T^{P_j} are pair-wise-close if, for $k = 1, \cdots, m$, $e_1 = e'_1$, $\cdots$, $e_k = e'_k$, $e_{k+1} \neq e'_{k+1}$, then $v_k = v'_k$, $(v_{k+1} : e_{k+1})$ and $(v'_{k+1} : e'_{k+1})$ are two nodes in the same minimal hedge of e_{k+1} and e'_{k+1} in $\beta(e_k)$. □

We denote $F[P] = (T^{\wp_1}\cdots T^{\wp_l})$ called a P-tuple. A P-tuple $F[P]$ is complete if $\forall\, T^{\wp_i} \in (T^{\wp_1}\cdots T^{\wp_l})(T^{\wp_i}$ is complete)).

We noted in the introduction that our definition on key is for ordered XML model. The P-tuple can capture the correct semantics of ordered XML by producing the correct tuples. For example, in the Fig.1(b), under node v_9, we do not produce the P-tuples $(T_{v_{16}}T_{v_{19}})$ and $(T_{v_{18}}T_{v_{17}})$ for the paths ended with elements C and D.

We are now ready to define XML key satisfaction. We assume that the DTD D always conforms to the XML document T.

Definition 6 (XML Key Satisfaction). *An XML tree T satisfies a key $\Bbbk(Q, \{P_1, ..., P_l\})$, denoted by $T \prec \Bbbk$, if the followings are hold:*

(i) If $\{P_1, ..., P_l\} = \phi$ in $\Bbbk$, then T satisfies $\Bbbk$ iff there exists one and only one T^Q in T;

(ii) else,

(a) $\forall\, T^Q \in \langle T^Q\rangle$ (exists at least one P-tuple in T^Q);

(b) $\forall\, T^Q \in \langle T^Q\rangle$ (every P-tuple in T^Q is complete);

(c) $\forall\, T^Q \in \langle T^Q\rangle$ (every P-tuple in T^Q is value distinct);

(d) $\forall T_1^Q, T_2^Q \in \langle T^Q\rangle$(exists two P-tuples $(T_1^{P_1}\cdots T_1^{P_l}) \in T_1^Q \wedge (T_2^{P_1}\cdots T_2^{P_l}) \in T_2^Q \wedge (T_1^{P_1}\cdots T_1^{P_l}) =_v (T_2^{P_1}\cdots T_2^{P_l})$ $\Rightarrow T_1^Q \equiv T_2^Q$). This requires that P-tuples under different selector nodes must be distinct. □

Consider the key $\Bbbk(root/A, \{B, M/C, M/D\})$ on the DTD D in the Fig. 1(a). We already showed how to produce P-tuples for field paths $B, M/C, M/D$ using

minimal structure and minimal hedge. We see that P-tuples are complete and value distinct in T_{v_1} and T_{v_2}. Thus the key $\Bbbk$ is satisfied by the document T in Fig.1(b).

We now extend our XML key definition to XML functional dependency.

Definition 7 (XML functional dependency). *An XML functional dependency over the XML DTD is defined as $\Phi(S, P \rightarrow Q)$ where S is a complete path, P is a set of simple paths as $\{\wp_1, \cdots, \wp_i, \cdots \wp_l\}$, and Q is a simple path or empty path. S is called the* **scope***, P is called the* **LHS** *or* **determinant***, and Q is called the* **RHS** *or* **dependent***. $S/\wp_i$ $(i = 1 \cdots l)$ and S/Q are complete paths. If $Q = \epsilon$, $\Phi(S, P \rightarrow \epsilon)$ means that P determines S. An XFD following the above definition is valid, denoted as $\Phi \sqsubset D$.*

For example, consider an XFD $\Phi(root/A, \{B, M/C\} \rightarrow M/D)$. $root/A$ is the scope, $B, M/C$ are the determinant, and M/D is the dependent. The paths $root/A$, $root/A/B$, $root/A/M/C$ and $root/A/M/D$ are the complete paths on the DTD D in Fig.1(a). Consider another XFD $\Phi(root/A, \{M/C, M/D\} \rightarrow \epsilon)$. It implies that $\Phi(root/A, \{M/C, M/D\} \rightarrow A)$ where $last(root/A) = A$.

Before defining XFD satisfaction, we give some definitions and notation using examples.

Consider an XFD $\Phi(root/A, \{B, M/C\} \rightarrow M/D)$ on the DTD D in Fig.1(a). Here, $root/A$ is the scope, $B, M/C$ are the LHS paths(P paths) and M/D is the RHS path(Q path). Now we need to produce PQ-tuples. So we take the paths in LHS and RHS together $B, M/C$ and M/D to find the minimal structure. Thus the minimal structure is $[B^+{}_\times[M^+{}_\times E^+]]$. The minimal hedge is $T_{v_3}T_{v_4}T_{v_5}T_{v_6}T_{v_7}$ for node v_1 and $T_{v_8}T_{v_9}T_{v_{10}}$ for node v_2 in Fig.1(b).

We already showed how to produce P-tuples in checking key satisfaction. We now produce PQ-tuples(tuples for paths P and Q together). We note that the paths $M/C, M/D$ has common path M and thus we need to find the products of the elements C and D under each node of the element M. Then we need to combine pair-wise values of C, D with B values. The PQ-tuples for node v_1 are $F_1[PQ] = \langle(v_3 : B : b1)(v_{11} : C : c1)(v_{13} : D : d4)\rangle$, $F_2[PQ] = \langle(v_3 : B : b1)(v_{12} : C : c2)(v_{13} : D : d4)\rangle$, $F_3[PQ] = \langle(v_4 : B : b2)(v_{11} : C : c1)(v_{13} : D : d4)\rangle$, $F_4[PQ] = \langle(v_4 : B : b2)(v_{12} : C : c1)(v_{13} : D : d4)\rangle$, $F_5[PQ] = \langle(v_3 : B : b1)(v_{14} : C : c3)(v_{15} : D : d5)\rangle$, $F_6[PQ] = \langle(v_4 : B : b2)(v_{14} : C : c3)(v_{15} : D : d5)\rangle$. The PQ-tuples for node v_2 are $F_7[PQ] = \langle(v_8 : B : b3)(v_{16} : C : c2)(v_{17} : D : d6)\rangle$ and $F_8[PQ] = \langle(v_8 : B : b3)(v_{18} : C : c7)(v_{19} : D : d7)\rangle$. We say P-tuple $F[P]$ for paths P and Q-tuple $F[Q]$ for paths Q. For example, consider $F_1[PQ] = \langle(v_3 : B : b1)(v_{11} : C : c1)(v_{13} : D : d4)\rangle$. So $F_1[P] = (v_3 : B : b1)(v_{11} : C : c1)$ and $F_1[Q] = (v_{13} : D : d4)$.

Now we define XFD satisfaction.

Definition 8 (XFD satisfaction). *Given a DTD D, an XML document T satisfies the XML functional dependency $\Phi(S, P \rightarrow Q)$, denoted as $T \prec \Phi$ if the followings are held.*

(a) If $Q = \epsilon$, then $\forall F[P] \in T^S$, $F[P]$ is complete.
(b) Else

(i) $\exists(F[P], F[Q]) \in T^S$ *and* $F[P], F[Q]$ *are complete.*
(ii) For every pair of tuples F_1 *and* F_2 *in* T^S*, if* $F_1[P] =_v F_2[P]$*, then* $F_1[Q] =_v F_2[Q]$.

Now consider the XFD $\Phi(root/A, \{B, M/C\} \rightarrow M/D)$. We have already shown how to produce PQ-tuples for paths $B, M/C, M/D$. We see that all PQ-tuples satisfy the conditions of the XFD satisfaction. So the XFD Φ is satisfied by the document T in Fig.1(b).

We have just finished the definitions of key and functional dependency for XML. Now we give the experiments on checking both key and XFD. All experiments are implemented in Java using a PC with Intel(R) Centrino Duo CPU T2050 at 1.60GHz, 1.49GB RAM and Microsoft Windows XP.

3 Performance Analysis of P-Tuple Generation

In the previous section, we defined XML key and XFD satisfaction. XML key satisfaction is based on distinctness of P-tuples while XFD satisfaction is based on value equivalence of PQ-tuples. Here two types of tuples are involved. However, for simplicity, we use P-tuples for both types of tuples as the generation of two types of tuples follows the same algorithm. As we mentioned before, checking the satisfaction of XML keys and XFDs includes two parts. The first part is the generation of P-tuples. The second part is to checking the distinctness or the value equivalence of these P-tuples. We now present the algorithm for P-tuple generation and analyze its performance. This algorithm will be used both in XML key and XFD satisfaction checking.

3.1 Algorithm for P-Tuple Generation

In P-tuple generation, we accomplish two tasks: parsing the document and pairing the values of elements to produce P-tuples while parsing. Here the term pairing means the process of computing the product of relevant hedges. For example, if the relevant hedges are $H_a = T_1T_2$, $H_b = T_3T_4$ and $H_c = T_5T_6$, pairing produces the tuples (T_1, T_3, T_5), (T_1, T_3, T_6), (T_1, T_4, T_5), (T_1, T_4, T_6), (T_2, T_3, T_5), (T_2, T_3, T_6), (T_2, T_4, T_5), and (T_2, T_4, T_6). Product calculation itself is not difficult, but in the process of pairing, product calculation has to be combined with parsing.

The subsection presents two algorithms. The algorithm 1 shows the parsing and the algorithm 2 shows the pairing and P-tuple generation. In parsing, we first find the nodes QN for the selector path. We then proceed to find the occurrences of elements for fields paths in order of DTD under a selector node. Note that paths in field of a key can appear as a set. But we order the paths of field of a key according to the order of the elements of DTD that involve key(we omit this process from the algorithm 1 for simplicity). We keep track of the occurrences of the elements which are the last elements of field paths so that the pairings can be done to produce P-tuples.

Data: An XML document T, An XML key $\Bbbk(Q, \{P_1, \cdots, P_n\})$
Result: A set of tuples,$F = (T^{P_1} \cdots T^{P_n})$
Let QN= all Q nodes in T
foreach node in QN do
  FIND_TUPLES(node);
end
//procedure
FIND_TUPLES(node)
{
if (all of $P_1, \cdots, P_n$ occurred with order and any pairing before) then
  MAKE_TUPLES($array\_P_1[], \cdots, (pair_{rs}[]), \cdots, array\_P_n[]$);
end
foreach j=1 to n do
  foreach k=j to n-1 do
    Check any need to pair as **PAIRING**($array\_P_j[], array\_P_{k+1}[]$);
  end
end
Store value for path P_i to $array\_P_i[]$ according to the order of the fields and keep track of order;
}

Algorithm 1. Parsing the document

We now give an example to show how the algorithms work. Consider the DTD D in Fig.1(a), the document T in Fig.1(b) conforming to the DTD D, and a valid key $\Bbbk(root/A, \{B, M/C, M/D, E\})$ on the DTD D. To check the satisfaction of the key, we take the document T and the key $\Bbbk$ as input for algorithm 1 to produce P-tuples.

After finding the selector nodes v_1 and v_2 with node name A, we call the procedure $FIND\_TUPLES(v_1)$ to produce all P-tuples for node v_1. In procedure $FIND\_TUPLES$, we traverse all nodes under node v_1 and check the occurrences of nodes for last element of all fields with order. We see that the nodes v_3 and v_4 are occurred for element B which is the last element of first field B. After that we get nodes v_{11}, v_{12} for C which is the last element of second field M/C and the node v_{13} for D which is the last element of third field M/D. We see that B and C appear two times. We term the multiple occurrences of C as *repeating structure* in the document. When we advance, we encounter again nodes v_{14} for C and v_{15} for D. The elements C, D appear multiple times and thus (C, D) is a repeating structure as a group of elements. So we call the procedure $PAIRING$ for previous (C, D) values. Thus we get $(v_{11} : C : c1)(v_{13} : D : d4)$, $(v_{12} : C : c2)(v_{13} : D : d4)$ and $(v_{14} : C : c3)(v_{15} : D : d5)$ as pair wise values for (C, D). At last, we call the procedure $MAKE\_TUPLES$ with B values, pair wise (C, D) values and E values. In similar way, the P-tuples are generated for the selector node v_2. The P-tuples are shown in Fig.2. We note that we consider only the $\#PCDATA$ values in P-tuple omitting the node identifier v and node label e because we use value comparison in key satisfaction.

```
//procedure
PAIRING(array_P_r[], array_P_s[])
{
foreach i=1 to sizeof(array_P_r[]) do
    foreach j=1 to sizeof(array_P_{r+1}[]) do
        ...
        foreach k=1 to sizeof(array_P_s[]) do
            pair_rs[][1]=array_P_r[i];
            pair_rs[][2]=array_P_{r+1}[j];
            ...
            pair_rs[][s-r]=array_P_s[k];
        end
        ...
    end
end
}
//procedure
MAKE_TUPLES(array_P_1[], ..., (pair_rs[]), ..., array_P_n[])
{
foreach i=1 to sizeof(array_P_1[]) do
    foreach j=1 to sizeof(pair_rs[][]) do
        foreach k=1 to sizeof(array_P_n[]) do
            tuple_{1...n}[][1] = array_P_1[i];
            ...
            tuple_{1...n}[][r] = pair_rs[j][1];
            ...
            tuple_{1...n}[][s] = pair_rs[j][s-r];
            ...
            tuple_{1...n}[][n] = array_P_n[k];
        end
    end
end
}
```

Algorithm 2. Pairing of values for fields and generation of tuples

For non-repeating structure, the complexity is $O(2n|e|)$ where n is the number of fields and $|e|$ is the average number of occurrences of an element in a hedge. The cost of parsing is $n|e|$. As we use breadth first search in traversing the document, thus we need to traverse $n|e|$ element nodes. In this case, we assume all element nodes are at the same level under a parent(context) node. The cost of pairing is $n|e|$ for close elements in the hedge. This is because in each pair, there is only one occurrence of each element.

For the repeating structure in the document, the complexity is $O(n\sqrt[n]{|e|}+|e|)$ where n is the number of fields and $|e|$ is the average number of occurrences of an element in a hedge. The cost $n\sqrt[n]{|e|}$ is for parsing using breadth first search. The cost $|e|$ is for pairing because we do the production of elements.

selector node	P-tuple F	B	C	D	E
v_1	$F_1 =$	((b1)	(c1)	(d4)	(e1))
v_1	$F_2 =$	((b1)	(c2)	(d4)	(e1))
v_1	$F_3 =$	((b1)	(c3)	(d5)	(e1))
v_1	$F_4 =$	((b2)	(c1)	(d4)	(e1))
v_1	$F_5 =$	((b2)	(c2)	(d4)	(e1))
v_1	$F_6 =$	((b2)	(c3)	(d5)	(e1))
v_2	$F_7 =$	((b3)	(c2)	(d6)	(e2))
v_2	$F_8 =$	((b3)	(c7)	(d7)	(e2))

Fig. 2. Tuples for $(B,(C,D),E)[last(B) = B$,$last(M/C) = C$,$last(M/D) = D$ and $last(E) = E]$

3.2 Performance of P-Tuple Generation

First, we analyze the time of P-tuple generation. We take the DTD D in Fig.1(a) and we generate XML documents with different structures and different sizes. By different structures, we mean *repeating* structure and $non-repeating$ structure. By repeating structure, we mean the multiple occurrences of elements in the hedge and by non-repeating structure, we mean the single occurrence of the elements in the hedge in the document. With single occurrence, an element or a group of elements appears only once while with multiple occurrence, an element or a group of elements appear more than one time in a hedge.

In the case of repeating structure, elements need to be combined from different occurrences to form a P-tuple. For the same number of P-tuples, if the structure is non-repeating, we need larger number of elements in the document, which means larger document size. In contrast, if the structure is repeating, because of production, a small document will make the number of tuples.

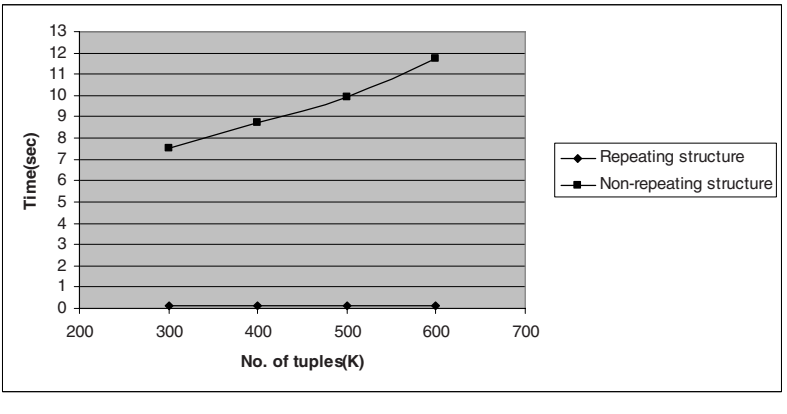

Fig. 3. Tuple generation time when the number of fields is fixed to 4

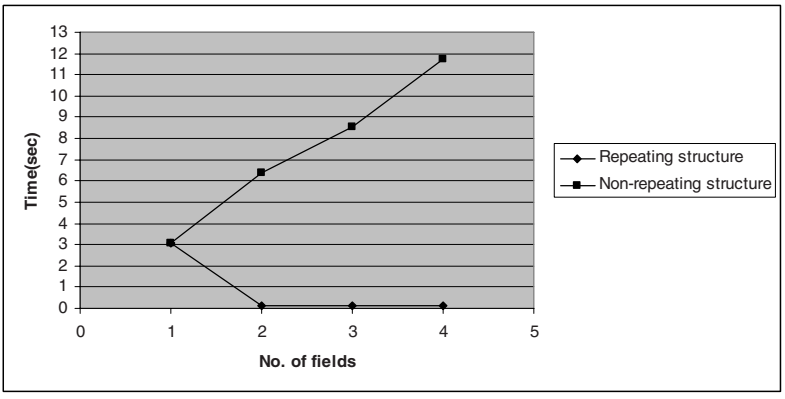

Fig. 4. Tuple generation time when the number of tuples is fixed to 600K

In Fig.3, we show P-tuple generation time where we fix the number of fields but we vary the number of tuples. For the non-repeating structure in the document, the time required for tuple generation is increasing linearly with the number of tuples. But for the repeating structure in the document, the time required for tuple generation is linear and nearly constant.

In Fig.4, we show P-tuple generation time where we fix the number of tuples but we vary the number of fields in the key. The time required for tuple generation with non-repeating structure is increasing linearly to the number of fields. But the time required for tuple generation with repeating structure is linear with constant values from two fields. We observe that for one field, the time for tuple generation either for non-repeating structure or for non-repeating structure is the same.

4 Checking XML Key Satisfactions

In this section, we study the performances of checking key satisfaction.

In checking satisfaction of key, we take generated P-tuples and then apply hashing techniques to find if all P-tuples are value distinct. In hashing, we use Java *Hashtable* where each P-tuple is used as a *Key* in the hash table. In this case, we take the P-tuples generated from the non-repeating structure of elements in the document because the time required for P-tuple generation for non-repeating structure is higher that of repeating structure. We then use hashing for checking whether the P-tuples are distinct.

In Fig.5, the time of checking key satisfaction which is actually the sum of the tuple generation time and the hashing time is shown with fixed number of fields but varying the number of tuples. The hashing time is linear with the number of tuples but is increasing slightly because of the increasing number of tuples to be checked in the hash table. We observe that the hashing time is smaller that P-tuple generation time. We have already shown the tuple generation time

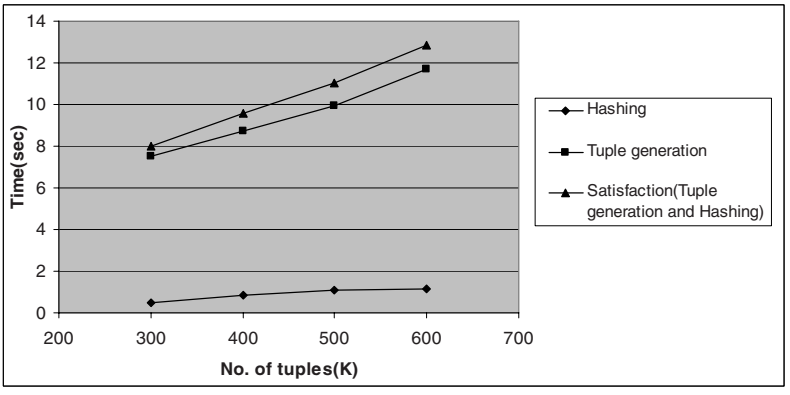

Fig. 5. Satisfaction time when the number of fields is fixed to 4

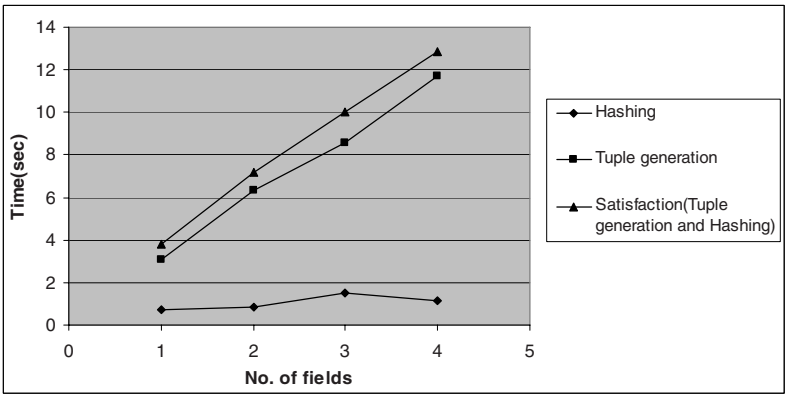

Fig. 6. Satisfaction time when the number of tuples is fixed to 600K

which is linear in Fig.3 for the non-repeating structure of elements. Thus key satisfaction time is also linear for fixed number of fields with varying number of tuples.

In Fig.6, we show the time for key satisfaction checking for the fixed number of tuples with varying number of fields in key. The hashing time is nearly linear with the number of fields and the tuple generation time is also linear with the number of fields. Thus the satisfaction time is also linear with the number of fields.

5 Checking XML Functional Dependency Satisfactions

In this section, we study the checking of XML functional dependency $\Phi(S, P \rightarrow Q)$ satisfaction. Like checking XML key satisfaction, there are also two important tasks in checking XFD satisfaction. Firstly, we need to produce PQ-tuples for

both the determinant P and the dependent Q together under the scope S. The second task is to check if the values for any two P-tuples are value equivalent, then their corresponding values for Q-tuples are also value equivalent.

5.1 PQ-Tuple Generation

We use the same algorithm used in P-tuple generation for XML key with slight modification in getting values for the paths P and Q. For XML keys, we take the string values for key paths but in XFD, we use tree structured values for paths. This is because in XML keys, we take the paths which are ended with type Str in field P, but in XFD, paths in P or Q may not be ended with type Str. For PQ-tuple generation in XFD satisfaction, we use paths $\{P, Q\}$ together because the we want close pair values for both *determinant* P and *dependent* Q. We show values of PQ-tuples for an XFD $\Phi(root/A, \{B, M/C, M/D\} \rightarrow E)$ in Fig. 7.

$scope$	$PQ-tuple\ F[PQ]$	$\wp_1 = B$	$\wp_2 = M/C$	$\wp_3 = M/D$	$Q = E$
v_1	$F_1[PQ] =$	$\langle(v_3 : B : b1)$	$(v_{11} : C : c1)$	$(v_{13} : D : d4)$	$(v_7 : E : e1)\rangle$
v_1	$F_2[PQ] =$	$\langle(v_3 : B : b1)$	$(v_{12} : C : c2)$	$(v_{13} : D : d4)$	$(v_7 : E : e1)\rangle$
v_1	$F_3[PQ] =$	$\langle(v_3 : B : b1)$	$(v_{14} : C : c3)$	$(v_{15} : D : d5)$	$(v_7 : E : e1)\rangle$
v_1	$F_4[PQ] =$	$\langle(v_4 : B : b2)$	$(v_{11} : C : c1)$	$(v_{13} : D : d4)$	$(v_7 : E : e1)\rangle$
v_1	$F_5[PQ] =$	$\langle(v_4 : B : b2)$	$(v_{12} : C : c2)$	$(v_{13} : D : d4)$	$(v_7 : E : e1)\rangle$
v_1	$F_6[PQ] =$	$\langle(v_4 : B : b2)$	$(v_{14} : C : c3)$	$(v_{15} : D : d5)$	$(v_7 : E : e1)\rangle$
v_2	$F_7[PQ] =$	$\langle(v_8 : B : b3)$	$(v_{16} : C : c2)$	$(v_{17} : D : d6)$	$(v_{10} : E : e2)\rangle$
v_2	$F_8[PQ] =$	$\langle(v_8 : B : b3)$	$(v_{18} : C : c7)$	$(v_{19} : D : d7)$	$(v_{10} : E : e2)\rangle$

Fig. 7. PQ-tuples for paths in P and Q

5.2 Hashing

We use Java $Hashtable$ in checking XFD satisfaction. From a PQ-tuple, we use value for paths in LHS(P) as Key and value for paths in RHS(Q) as corresponding $Value$ of Key in $Hashtable$. In checking XFD satisfaction using hash, we check that if two $Keys$ are with same values, then their corresponding $Values$ are also the same in hash table.

5.3 Experiments on XFD Satisfaction

We now give the performance of checking XFD satisfaction. In checking XFD satisfaction, like XML key checking, we take the PQ-tuple generation time for non-repeating structure of the document because the time required for P-tuple generation for non-repeating structure is much higher than that of repeating structure.

In first experiment, we take an XFD $\Phi(root/A, \{B, M/C, M/D\} \rightarrow E)$ on the DTD D in Fig.1(a) with three paths in LHS and one path in RHS fixed

and vary the tuple size. As we take fixed number of paths in LHS and RHS, so for PQ-tuple generation, we fix the number of paths $(B, M/C, M/D, E)$. In Fig.8, we give the results of XFD satisfaction. The P-tuple generation time is increasing and linear with the increasing number of tuples. The hashing time is slightly increasing and linear. Thus the satisfaction time is also increasing and linear.

In the second experiment, we take XFD with varying number of paths in LHS but keeping RHS fixed. The XFDs are $\Phi(root/A, \{B\} \rightarrow M/C)$, $\Phi(root/A, \{B, M/C\} \rightarrow M/D)$ and $\Phi(root/A, \{B, M/C, M/D\} \rightarrow E)$ on the DTD D in Fig.1(a). We also keep the number of tuples fixed to 500K. The result is shown in Fig.9. The PQ-tuple generation time is increasing because the number of paths is increased. However, the tuple generation time is linear. The hashing time is slightly increasing and this is because of Java hashtable management. The hashing time is also linear and thus the satisfaction time is also linear.

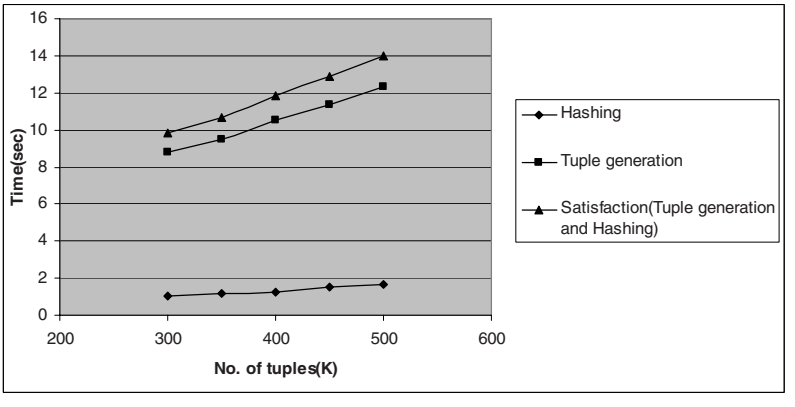

Fig. 8. Satisfaction time when the number of paths in LHS is fixed to 3

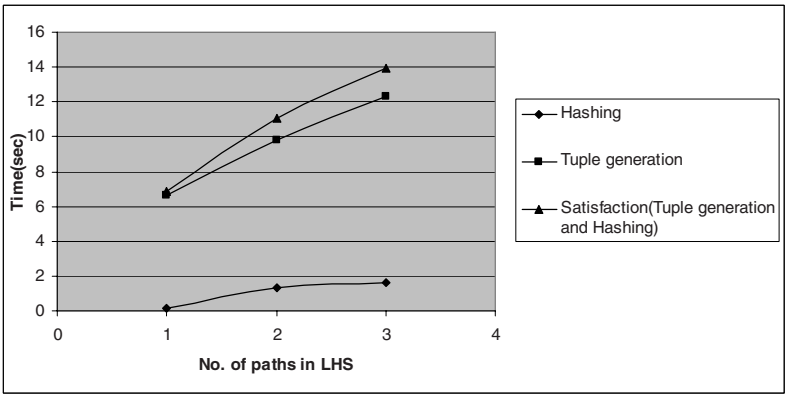

Fig. 9. Satisfaction time when the number of tuples is fixed to 500K

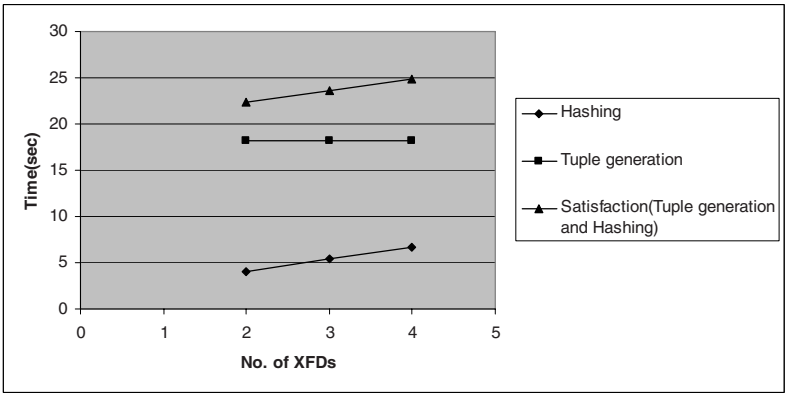

Fig. 10. Satisfaction time when the number of tuples is fixed to 600K

In third experiment, we fix the number of tuples but we vary the number of XFDs to be checked. We first take two XFDS, $\Phi(root/A, \{B, M/C, M/D\} \rightarrow E)$ and $\Phi(root/A, \{B, M/C\} \rightarrow \{M/D, E\})$. We add $\Phi(root/A, \{B, M/C\} \rightarrow M/D)$ with two XFDs to make three XFDs. Then we add $\Phi(root/A, \{B\} \rightarrow M/D)$ with three XFDs to make four XFDs. We generate the PQ-tuples for paths $(B, M/C, M/D, E)$ only once and then take PQ-tuples for hashing in checking different number of XFDs. For each XFD checking, we use one hash table. We show the experiment in Fig.10 where PQ-tuple generation is constant for different number of XFDs as we take same number of paths for PQ-tuple generation for fixed number of tuples. The hashing time is increasing with the number of XFDs to be checked and it is linear. Thus the satisfaction time is also increasing in accordance with the increasing hashing time but it is linear.

6 Conclusions

We showed the implementation of checking XML key and functional dependency. In implementation, we consider a novel technique in generating tuples using pairwise values for ordered XML documents. The experiments showed that XML key can be checked in linear time with the number of tuples and the number of fields in the key. Also, XML functional dependency can be checked in linear time with the number of tuples, the number of fields and the number of XFDs.

References

1. Poggi, A., Abiteboul, S.: XML Data Integration with Identification. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 106–121. Springer, Heidelberg (2005)
2. Yu, C., Popa, L.: Constraint-based XML query rewriting for data integration. In: SIGMOD, pp. 371–382 (2004)

3. Fankhouser, P., Klement, T.: XML for Datawarehousing Chances and Challenges. In: Kambayashi, Y., Mohania, M., Wöß, W. (eds.) DaWaK 2003. LNCS, vol. 2737, pp. 1–3. Springer, Heidelberg (2003)
4. Popa, L., Velegrakis, Y., Miller, R.J., Hernandez, M.A., Fagin, R.: Translating the web data. In: VLDB, pp. 598–609 (2002)
5. Deutsch, A., Tannen, V.: MARS: A System for Publishing XML from Mixed and Redundant Storage. In: VLDB (2003)
6. Buneman, P., Fan, W., simeon, J., Weinstein, S.: Constraints for Semistructured Data and XML. In: SIGMOD Record, pp. 47–54 (2001)
7. Fan, W.: XML Constraints: Specification, Analysis, and Applications. In: DEXA, pp. 805–809 (2005)
8. Buneman, P., Davidson, S.B., Fan, W., Hara, C.S., Tan, W.C.: Keys for XML. Computer Networks 5(39), 473–487 (2002)
9. Buneman, P., Davidson, S.B., Fan, W., Hara, C.S., Tan, W.C.: Reasoning about keys for XML. Inf. Syst. 8(28), 1037–1063 (2003)
10. Fan, W., Schwenzer, P., Wu, K.: Keys with Upward Wildcards for XML. In: Mayr, H.C., Lazanský, J., Quirchmayr, G., Vogel, P. (eds.) DEXA 2001. LNCS, vol. 2113, pp. 657–667. Springer, Heidelberg (2001)
11. Fan, W., Simeon, J.: Integrity constraints for XML. In: PODS, pp. 23–34 (2000)
12. Fan, W., Libkin, L.: On XML Integrity Constraints in the Presence of DTDs. Journal of the ACM 49, 368–406 (2002)
13. Bray, T., Paoli, J., Sperberg-McQueen, C.M.: Extensible Markup Language (XML) 1.0., World Wide Web Consortium (W3C) (Febuary 1998), `http://www.w3.org/TR/REC-xml`
14. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1:Structures, W3C Working Draft (April 2000), `http://www.w3.org/TR/xmlschema-1/`
15. Vincent, M., Liu, J.: Functional Dependencies for XML. In: Zhou, X., Zhang, Y., Orlowska, M.E. (eds.) APWEB 2003. LNCS, vol. 2642, pp. 22–34. Springer, Heidelberg (2003)
16. Liu, J., Vincent, M., Liu, C.: Local XML Functional Dependencies. In: WIDM, pp. 23–28 (2003)
17. Vincent, M.W., Liu, J.: Checking Functional Dependency Satisfaction in XML. In: Bressan, S., Ceri, S., Hunt, E., Ives, Z.G., Bellahsène, Z., Rys, M., Unland, R. (eds.) XSym 2005. LNCS, vol. 3671, pp. 4–17. Springer, Heidelberg (2005)
18. Liu, J., Vincent, M.W., Liu, C., Mohania, M.: Checking Multivalued Dependencies in XML. In: Zhang, Y., Tanaka, K., Yu, J.X., Wang, S., Li, M. (eds.) APWeb 2005. LNCS, vol. 3399, pp. 320–332. Springer, Heidelberg (2005)
19. Liu, Y., Yang, D., Tang, S., Wang, T., Gao, J.: Validating key constraints over XML document using XPath and structure checking. Future Generation Computer Systems 21(4), 583–595 (2005)
20. Chen, Y., Davidson, S.B., Zheng, Y.: XKvalidator: a constraint validator for XML. In: CIKM, pp. 446–452 (2002)
21. Arenas, M., Libkin, L.: A Normal Form for XML documents. In: ACM PODS, pp. 85–96 (2002)
22. Vincent, M., Liu, J., Liu, C.: Strong Functional Dependencies and Their Application to Normal Forms in XML. In: ACM TODS, pp. 445–462 (2004)
23. Liu, J., Vincent, M., Liu, C.: Functional Dependencies, From Relational to XML. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 531–538. Springer, Heidelberg (2003)

24. Lee, M.L., Ling, T.-W., Low, W.L.: Designing Functional Dependencies for XML. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 124–141. Springer, Heidelberg (2002)
25. Hartmann, S., Link, S.: Expressive, yet tractable XML keys. In: EDBT 2009, pp. 357–367 (2009)
26. Hartmann, S., Khler, H., Link, S., Trinh, T., Wang, J.: On the Notion of an XML Key. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 103–112. Springer, Heidelberg (2008)
27. Shahriar, M.S., Liu, J.: Preserving Functional Dependency in XML Data Transformation. In: Atzeni, P., Caplinskas, A., Jaakkola, H. (eds.) ADBIS 2008. LNCS, vol. 5207, pp. 262–278. Springer, Heidelberg (2008)
28. Shahriar, M.S., Liu, J.: Towards the Preservation of Keys in XML Data Transformation for Integration. In: COMAD 2008, pp. 116–126 (2008)
29. Java Document Object Model (DOM)
`http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/package-summary.html`