Schema Design and Normalization Algorithm for XML Databases Model

doi:10.3991/ijet.v4i2.768

Hosam F. El-Sofany1, and Samir A. El-Seoud[2]
[1]Department of Computer Science and Engineering, College of Engineering, Qatar University
[2]Computer Science Department, Princess Sumaya University for Technology

***Abstract*—In this paper we study the problem of schema design and normalization in XML databases model. We show that, like relational databases, XML documents may contain redundant information, and this redundancy may cause update anomalies. Furthermore, such problems are caused by certain functional dependencies among paths in the document. Based on our research works, in which we presented the functional dependencies and normal forms of XML Schema, we present the decomposition algorithm for converting any XML Schema into normalized one, that satisfies X-BCNF.**

***Index Terms*—XML Databases Design, Functional Dependencies, Normal Forms, Normalization, Algorithms**

I. INTRODUCTION

The eXtensible Markup Language (XML) has recently emerged as a standard for data representation and interchange on the Internet [1]. Although many XML documents are views of relational data, the number of applications using *native XML* documents is increasing rapidly. Such applications may use native XML storage facilities [2], and update XML data [3]. Updates, like in relational databases, may cause anomalies if data is redundant. In the relational world, anomalies are avoided by developing a well-designed database schema. XML has its version of schema too; such as *DTD* (Document Type Definition), and *XML Schema* [4]. Our goal is to find the principles for good XML Schema design. We believe that it is important to do this research now, as a lot of data is being put on the web. Once massive web databases are created, it is very hard to change their organization; thus, there is a risk of having large amounts of widely accessible, but at the same time poorly organized data.

Normalization is a process which eliminates redundancy, organizes data efficiently and improves data consistency. Whereas normalization in the relational world has been quite explored, it is a new research area in *native XML databases*. Even though native XML databases mainly work with *document-centric* XML documents, and the structure of several XML document might differ from one to another, there is room for redundant information. This redundancy in data may impact on document updates, efficiency of queries, etc. Figure 1, shows an overview of the XML normalization algorithms that we propose [10-12].

This paper focus on the normal form theory. This theory concerns the old question of *well-designed* databases or in other words the syntactic characterization of semantically desirable properties. These properties are tightly connected with dependencies such as *keys*, *functional dependencies*, *weak functional dependencies*, *equality generating dependencies*, *multi-valued dependencies*, *inclusion dependencies*, *join dependencies*, etc. All these classes of dependencies have been deeply investigated in the context of the relational data model [5-8]. The work now requires its generalization to XML (trees like) model.

Our goal is to apply the concepts of relational database normalization to XML Schema design. We show how to transfer an XML Schema *X*, that based on a set of functional dependencies *F*, into a new specification (*X′*, *F′*) that is in XML normal form (*X-BCNF*) and contains the same information.

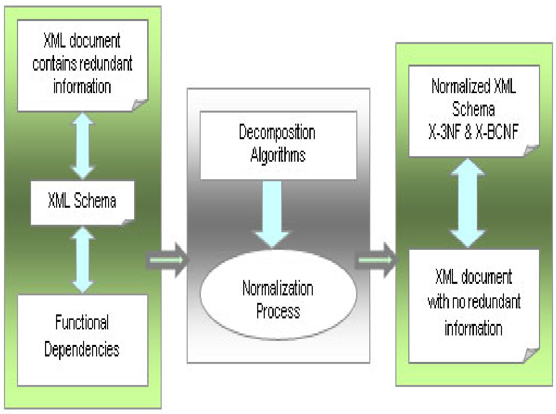


Figure 1. An overview of the XML normalization algorithms

II. MOTIVATING EXAMPLE

In this section, through an example, we show that, like relational databases, XML documents may contain redundant information, and this redundancy may cause update anomalies.

Example 1: Consider the following XML Schema that describes a part of a "*university*" database. For every course, we store its number (`cno`), its title and the list of students taking the course. For each student taking a course, we store the student number (`sno`), name, and the grade in the course.

An example of an XML document (tree) that conforms to this XML Schema is shown in Figure 2 [9]. This document satisfies the following constraint:

"any two student elements with the same sno value must have the same name".

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="courses">
 <xs:complexType>
  <xs:sequence>
     <xs:element name ="course" type ="course" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="course">
 <xs:complexType>
  <xs:sequence>
     <xs:element name="title" type="xs:string" />
     <xs:element name="taken_by" type="taken_by" maxOccurs="unbounded" />
  </xs:sequence>
   <xs:attribute name="cno" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
<xs:element name="taken_by">
 <xs:complexType>
  <xs:sequence>
    <xs:element name="student"  type="student" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="student">
 <xs:complexType>
  <xs:sequence>
    <xs:element name="name"  type="xs:string" />
    <xs:element name="grade" type="xs:string" />
  </xs:sequence>
    <xs:attribute name="sno" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>

</xs:schema>
```

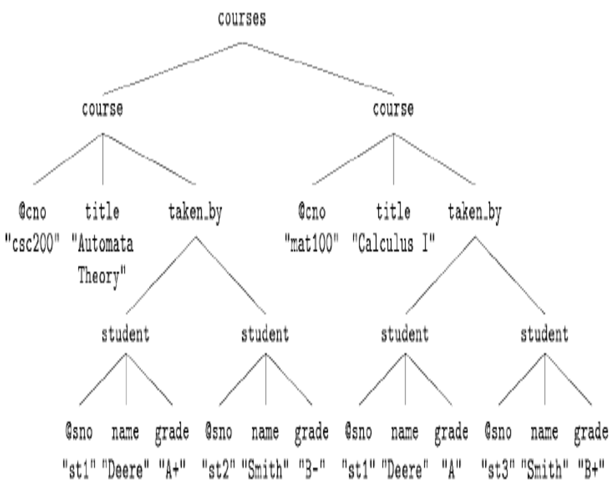


Figure 2. A document containing redundant information

This constraint (which looks like a FD), causes the document to store redundant information: for example, the name `Deere` for student `st1` is stored twice, as in relational databases, such redundancies can lead to update anomalies: for example, updating the name of `st1` for only one course results in an inconsistent document, and removing the student from a course may result in removing that student from the document altogether.

In order to eliminate redundant information, we use a technique similar to the relational one, and split the information about the name and the grade. Since we deal with just one XML document, we must do it by creating an extra element of `ComplexType`, called `info`, for student information, as shown in the figure below.

Each `info` element has (as children) one `name` and a sequence of `number` elements, with `sno` as an attribute. Different students can have the same `name`, and we group all student numbers `sno` for each `name` under the same `info` element. A restructured document that conforms to this XML Schema is shown in Figure 3 [9]. Note that `st2` and `st3` are put together because both students have the same name.

This example remembers us with the bad relational design caused by nonkey FDs, and how the database designer solve this problem by modifying the schema.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="courses">
 <xs:complexType>
  <xs:sequence>
     <xs:element name ="course" type ="course" maxOccurs="unbounded" />
     <xs:element name="info" type="info" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="course">
 <xs:complexType>
  <xs:sequence>
     <xs:element name="title" type="xs:string" />
     <xs:element name="taken_by" type="taken_by" maxOccurs="unbounded" />
  </xs:sequence>
   <xs:attribute name="cno" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
<xs:element name=" taken_by">
 <xs:complexType>
  <xs:sequence>
    <xs:element name="student" type="student" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="student">
 <xs:complexType>
  <xs:sequence>
    <xs:element name="grade" type="xs:string" />
  </xs:sequence>
    <xs:attribute name="sno" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
<xs:element name="info">
 <xs:complexType>
  <xs:sequence>
     <xs:element name="number" type="xs:string" maxOccurs="unbounded" />
     <xs:element name="name"/ type="xs:string">
  </xs:sequence>
    <xs:attribute name="sno" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>

</xs:schema>
```

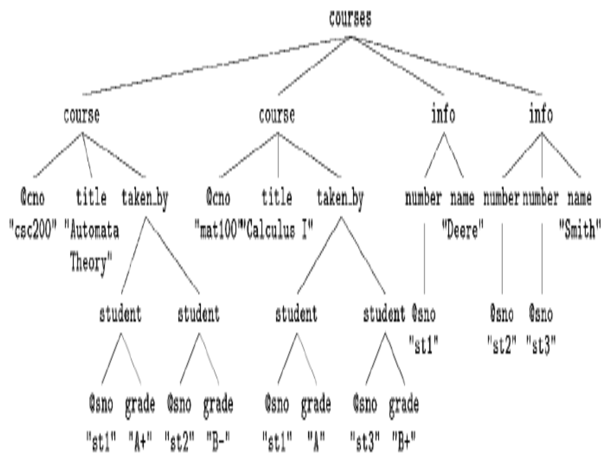


Figure 3. A well-designed document

III. PRIMARILY DEFINITIONS

To extend the notions of FDs to the XML model, we represent XML trees as sets of tuples [9], and find the correspondence between documents and relations that leads to the definition of functional dependency.

We first describe the formal definitions of XML Schema (*XSchema*) and the conforming of XML tree to XSchema. The definition of XSchema is based on regular tree grammar theory that introduced in [14]. Assume that we have the following disjoint sets:

- $\hat{E}$: set of element names,
- $\hat{A}$: set of attribute names,
- *DT*: set of atomic *data types* (e.g., ID, IDREF, IDREFS, string, integer, date, etc).
- *Str:* set of possible values of string-valued attributes

- *Vert: set* of node identifiers

All *attribute names* start with the symbol @. The symbols φ and S represent element type declarations `EMPTY` (null) and `#PCDATA`, respectively.

***Definition 1 (XSchema)*:** An XSchema is denoted by 6-tuple: $X = (E, A, M, P, r, \sum)$, where:

- $E \subseteq \hat{E}$, is a finite set of *element names*.
- $A \subseteq \hat{A}$, is a finite set of *attribute names*.
- M is a function from E to its *element type definitions*: *i.e.*, $M(e) = \alpha$, where $e \in E$, and α is a regular expression: $\alpha ::= \varepsilon \mid t \mid \alpha + \alpha \mid \alpha, \alpha \mid \alpha^* \mid \alpha^? \mid \alpha^+$ where ε denotes the *empty element*, $t \in DT$, "+" for the *union*, "," for the *concatenation*, α^* for the *Kleene closure*, $\alpha^?$ for $(\alpha + \varepsilon)$ and α^+ for (α, α^*)
- P is a function from an *attribute name a* to its *attribute type definition*: *i.e.*, $P(a) = \beta$, where β is a 4-tuple (t, n, d, f), where:
  - $t \in DT$,
  - n is either "?" (*nullable*) or "¬?" (*not nullable*),
  - d is a finite set of valid *domain values* of a or ε if not known, and
  - f is a default value of a or ε if not known.
- $r \subseteq E$ is a finite set of *root elements*,
- $\sum$ is a finite set of *integrity constraints* for XML model. The integrity constraints we consider are *keys* (*P.K, F.K,…*), and *dependencies* (*functional, and inclusion*).

***Definition 2 (Path in XSchema)*:** Given an XSchema $X = (E, A, M, P, r, \sum)$, a string $p = p_1 \ldots p_n$, is a *path* in X if, $p_1 = r$, p_i is in the alphabet of $M(p_{i-1})$, for each $i \in [2, n-1]$, and p_n is in the alphabet of $M(p_{n-1})$ or $p_n = @l$ for some $@l \in P(p_{n-1})$.

- We define *length*(p) as n and *last*(p) as p_n.
- We let *paths*(X) stand for the set of all paths in X, and *EPaths*(X) for the set of all paths that ends with an element type (rather than an attribute or S), that is: $EPaths(X) = \{ p \in paths(X) \mid last(p) \in E \}$.
- An XSchema is called *recursive* if *paths*(X) is infinite.

***Definition 3 (XML Tree)*:** An *XML tree T* is defined to be a tree, $T = (V, lab, ele, att, root)$
Where:

- $V \subseteq Vert$ is a finite set of *vertices* (*nodes*).
- $lab : V \rightarrow \hat{E}$.
- $ele : V \rightarrow Str \cup V^*$
- *att* is a partial function $V \times \hat{A} \rightarrow Str$. For each $v \in V$, the set $\{@l \in \hat{A} \mid att(v, @l) \text{ is defined}\}$ is required to be finite.
- $root \in V$ is called *the root of T*.

The *parent-child* edge relation on V, $\{(v_1, v_2) \mid v_2$ occurs in $ele(v_1)\}$, is required to form a rooted tree. Note that, the children of an element node can be either zero or more element nodes or one string.

***Definition 4 (Path in XML Tree)*:** Given an XML tree T, a string: $p_1 \ldots p_n$ with $p_1, \ldots, p_{n-1} \in \hat{E}$ and $p_n \in \hat{E} \cup \hat{A} \cup \{S\}$ is a *path* in T if there are vertices $v_1 \ldots v_{n-1} \in V$ *s.t.*

- $v_1 = root$, v_{i+1} is a child of v_i $(1 \le i \le n-2)$, $lab(v_i) = p_i$ $(1 \le i \le n-1)$.
- If $p_n \in \hat{E}$, then there is a child v_n of v_{n-1} *s.t.* $lab(v_n) = p_n$. If $p_n = @l$, with $@l \in \hat{A}$, then $att(v_{n-1}, @l)$ is defined. If $p_n = S$, then v_{n-1} has a child in *Str*.
- We let *paths*(T) stand for the set of paths in T.

Now, we give a definition of *a tree conforming* to the *XSchema* ($T \models X$), and *a tree compatible* with X ($T \triangleleft X$).

***Definition 5*:** Given an *XSchema* $X = (E, A, M, P, r, \sum)$ and an XML tree $T = (V, lab, ele, att, root)$, we say that T *is valid w.r.t. X* (*or T conforms to X*) *written as* ($T \models X$) if,

- $lab: V \rightarrow E$.
- For each $v \in V$, if $M(lab(v)) = S$, then $ele(v) = [s]$, where $s \in Str$. Otherwise, $ele(v) = [v_1, \ldots, v_n]$, and the string $lab(v_1) \ldots lab(v_n)$ must be in the regular language defined by $M(lab(v))$.
- *att* is a partial function, $att: V \times A \rightarrow Str$, s.t. for any $v \in V$ and $@l \in A$, $att(v, @l)$ is defined iff $@l \in P(lab(v))$.
- $lab(root) = r$.
- We say that T *is compatible with X* (written $T \triangleleft X$) iff $paths(T) \subseteq paths(X)$.
- Clearly, $T \models X \Rightarrow T \triangleleft X$.

***Definition 6*:** Given two XML trees $T_1 = (V_1, lab_1, ele_1, att_1, root_1)$ and $T_2 = (V_2, lab_2, ele_2, att_2, root_2)$, we say that T_1 is *subsumed* by T_2, written as $T_1 \le T_2$ if :

- $V_1 \subseteq V_2$.
- $root_1 = root_2$.
- $lab_2|_{V1} = lab_1$.
- $att_2|_{V1 \times \hat{A}} = att_1$.
- $\forall v \in V_1$, $ele_1(v)$ is a sub-list of a permutation of $ele_2(v)$.

***Definition 7*:** Given two XML trees T_1 and T_2, we say that T_1 is *equivalent to* T_2 written $T_1 \equiv T_2$, *iff* $T_1 \le T_2$ and $T_2 \le T_1$ (*i.e.*, $T_1 \equiv T_2$ *iff* T_1 and T_2 are equal as unordered trees).

- We define $[T]$ to be the ≡-equivalence class of T.
- We write: $[T] \models X$ if $T_i \models X$ for some $T_i \in [T]$.
- It is easy to see that for any $T_1 \equiv T_2$, $paths(T_1) = paths(T_2)$, hence,
- $T_1 \triangleleft X$ *iff* $T_2 \triangleleft X$.

- We shall also write $T_1 < T_2$ when $T_1 \leq T_2$ and $T_2 \nleq T_1$.

In the following definition, we extend the notion of tuple for relational databases to the XML model. In a relational database, a tuple is a function that assigns to each attribute a value from the corresponding domain. In our setting, a tree tuple t in a XML Schema X is a function that assigns to each path in X a value in $Vert \cup Str \cup \{\varphi\}$ in such a way that t represents a finite tree with paths from X containing at most one occurrence of each path. In this section, we show that an XML tree can be represented as a set of tree tuples.

***Definition 8 (Tree tuples)*:** Given XML Schema $X = (E, A, M, P, r, \sum)$, a *tree tuple* $t \in X$ is a function,

$t : paths(X) \rightarrow Vert \cup Str \cup \{\varphi\}$ such that

- For $p \in EPaths(X)$, $t(p) \in Vert \cup \{\varphi\}$, and $t(r) \neq \varphi$
- For $p \in paths(X) - EPaths(X)$, $t(p) \in Str \cup \{\varphi\}$
- If $t(p_1) = t(p_2)$ and $t(p_1) \in Vert$, then $p_1 = p_2$
- If $t(p_1) = \varphi$ and p_1 is a prefix of p_2, then $t(p_2) = \varphi$
- $\{ p \in paths(X) \mid t(p) \neq \varphi \}$ is finite

$\mathcal{T}(X)$ is defined to be *the set of all tree tuples in X*. For a tree tuple t and a path p, we write $t.p$ for $t(p)$.

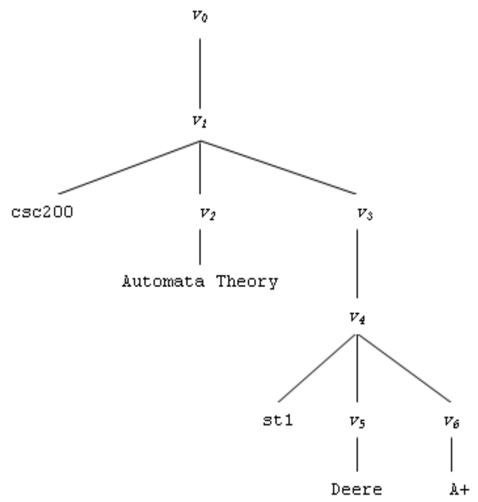
Example 2: Suppose that X is the XML Schema shown in example 1. Then a tree tuple in X assigns values to each path in $paths(X)$ such as:

```
t(courses) = v0
t(courses.course) = v1
t(courses.course.@cno) = csc200
t(courses.course.title) = v2
t(courses.course.title.S) = Automata Theory
t(courses.course.taken_by) = v3
t(courses.course.taken_by.student) = v4
t(courses.course.taken_by.student.@sno) = st1
t(courses.course.taken_by.student.name) = v5
t(courses.course.taken_by.student.name.S) = Deere
t(courses.course.taken_by.student.grade) = v6
t(courses.course.taken_by.student.grade.S) = A+
```

***Definition 9* (*tree$_X$*):** Given XML Schema $X = (E, A, M, P, r, \sum)$ and a tree tuple $t \in \mathcal{T}(X)$, $tree_X(t)$ is defined to be an XML tree (V, lab, ele, att, $root$), where:

- $root = t.r$
- $V = \{v \in Vert \mid \exists\ p \in paths(X)$ such that $v = t.p\}$
- If $v = t.p$ and $v \in V$, then $lab(v) = last(p)$
- If $v = t.p$ and $v \in V$, then $ele(v)$ is defined to be the list containing
- $\{t.p' \mid t.p' \neq \varphi$ and $p' = p.\tau$, $\tau \in E$, or $p' = p.S$, ordered lexicographically
- If $v = t.p$, $@l \in A$ and $t.p.@l \neq \varphi$, then $att(v, @l) = t.p.@l$

Example 3: Let X be the XML Schema from example 1 and t the tree tuple from Example 2. Then, t gives rise to the following XML tree:



Proposition 1. *If $t \in \mathcal{T}(X)$, then $tree_X(t) \lhd X$.* □

We would like to describe XML trees in terms of the tuples they contain. For this, we need to select tuples containing the maximal amount of information. This is done via the usual notion of ordering on tuples (relations).

- If we have two tree tuples t_1, t_2, we write $t_1 \subseteq t_2$ if whenever $t_1.p$ is defined, then $t_2.p$ is also defined, and $t_1.p \neq \varphi \Rightarrow t_1.p = t_2.p$.
- As usual, $t_1 \subset t_2$ means $t_1 \subseteq t_2$ and $t_1 \neq t_2$.
- Given two sets of tree tuples, Y and Z, we write: $Y \subseteq^b Z$, if: $\forall\ t_1 \in Y\ \exists\ t_2 \in Z$ *s.t.* $t_1 \subseteq t_2$.

***Definition 10* (*tuples$_X$*):** Given XML Schema X and an XML tree T such that $T \lhd X$, $tuples_X(T)$ is defined to be the set of *maximal tree tuples t* (with respect to $\subseteq$), *s.t.* $tree_X(t)$ is subsumed by T, that is:

$max_{\subseteq} \{ t \in \mathcal{T}(X) \mid tree_X(t) \leq T \}$

Note that:

- $T_1 \equiv T_2$ implies $tuples_X(T_1) = tuples_X(T_2)$.
- Hence, $tuples_X$ applies to equivalence classes: $tuples_X([T]) = tuples_X(T)$.
- The following proposition lists some simple properties of $tuples_X(\cdot)$

Proposition 2. *If $T \lhd X$, then $tuples_X(T)$ is a finite subset of $T(X)$. Furthermore, $tuples_X(\cdot)$ is monotone: $T_1 \leq T_2$ implies $tuples_X(T_1) \subseteq^b tuples_X(T_2)$.*

Proof. We prove only monotonicity. Suppose that $T_1 \leq T_2$ and $t_1 \in tuples_X(T_1)$. We have to prove that $\exists\ t_2 \in tuples_X(T_2)$ such that $t_1 \subseteq t_2$. If $t_1 \in tuples_X(T_2)$, this is obvious, so assume that $t_1 \notin tuples_X(T_2)$. Given that $t_1 \in tuples_X(T_1)$, $tree_X(t_1) \leq T_1$, and therefore, $tree_X(t_1) \leq T_2$. Hence, by definition of $tuples_X(\cdot)$, there exists $t_2 \in tuples_X(T_2)$ such that $t_1 \subset t_2$, since $t_1 \notin tuples_X(T_2)$. □

Example 4: In example 1, we saw the XML Schema X and a tree T conforming to X. In example 2, we saw one tree tuple t for that tree, with identifiers assigned to some of the element nodes of T. If we assign identifiers to the rest of the nodes, we can compute the set $tuples_X(T)$.

{(v_0, v_1, csc200, v_2, Automata Theory, v_3, v_4, st1, v_5, Deere, v_6, A+),
(v_0, v_1, csc200, v_2, Automata Theory, v_3, v_7, st2, v_8, Smith, v_9, B-),
(v_0, v_{10}, mat100, v_{11}, Calculus I, v_{12}, v_{13}, st1, v_{14}, Deere, v_{15}, A),
(v_0, v_{10}, mat100, v_{11}, Calculus I, v_{12}, v_{16}, st3, v_{17}, Smith, v_{18}, B+)}.

Finally, we define the trees represented by a set of tuples Y as the minimal, with respect to $\leq$, trees containing all tuples in Y.

***Definition 11* (*trees$_X$*):** Given XML Schema X and a set of tree tuples $Y \subseteq T(X)$, $trees_X(Y)$ is defined to be:

$min_{\leq}\{ T \mid T \lhd X$ and $\forall\ t \in Y, tree_X(t) \leq T \}$.

Notice that, if $T \in trees_X(Y)$ and $T' \equiv T$, then T' is in $trees_X(Y)$. The following shows that every XML document can be represented as a set of tree tuples, if we consider it as an unordered tree. That is, a tree T can be reconstructed from $tuples_X(T)$, up to equivalence $\equiv$.

Theorem 1. *Given XML Schema X and an XML tree T, if $T \lhd X$, then $trees(tuples_X([T])) = [T]$.*

Proof: Every XML tree is finite, and, therefore, $tuples_X([T]) = \{t_1, \ldots, t_n\}$, for some n. Suppose that $T \notin trees_X(\{t_1, . . . , t_n\})$. Given that $tree_X(t_i) \leq T$, for each $i \in [1, n]$, there is an XML tree T' *s.t.* $T' \leq T$ and $tree_X(t_i) \leq T'$, for each $i \in [1, n]$. If $T' < T$, then there is at least one node, string or attribute value contained in T which is not contained in T'. This value must be contained in some tree tuple t_j ($j \in [1, n]$), which contradicts $tree_X(t_j) \leq T'$. Therefore, $T \in trees_X(tuples_X([T]))$.

Let $T' \in trees_X(tuples_X([T]))$. For each $i \in [1, n]$, $tree_X(t_i) \leq T'$. Thus, given that, $tuples_X(T) = \{t_1, \ldots, t_n\}$, we conclude that $T \leq T'$, and, therefore, by definition of $trees_X$, $T' \equiv T$. □

Example 5: It could be the case that for some set of tree tuples Y there is no an XML tree T such that for every $t \in Y$, $tree(t) \leq T$. For example, let X be XML Schema, $X = (E, A, M, P, r, \sum)$, where $E = \{r, a, b\}$, $A = \varphi$, $M(r) = (a|b)$, $M(a) = \varepsilon$ and $M(b) = \varepsilon$. Let $t_1, t_2 \in \mathcal{T}(X)$ be defined as:

$$t_1.r = v_0 \qquad t_2.r = v_2$$
$$t_1.r.a = v_1 \qquad t_2.r.a = \varphi$$
$$t_1.r.b = \varphi \qquad t_2.r.b = v_3$$

Since $t_1.r \neq t_2.r$, there is no an XML tree T such that, $tree_X(t_1) \leq T$ and $tree_X(t_2) \leq T$.

- We say that $Y \subseteq \mathcal{T}(X)$ is *X-compatible* if there is an XML tree T: $T \lhd X$ and $Y \subseteq tuples_X(T)$.
- For X-compatible set of tree tuples Y, there is always an XML tree T: for every $t \in Y$, $tree_X(t) \leq T$.

Proposition 3. *If $Y \subseteq \mathcal{T}(X)$ is X-compatible, then:*

(a) *There is an XML tree T such that $T \lhd X$ and $trees_X(Y) = [T]$*, and

(b) $Y \subseteq^b tuples_X(trees_X(Y))$.

Proof:

(a) Suppose that $X = (E, A, M, P, r, \sum)$. Since Y is X-compatible, $\exists$ an XML tree $T' = (V', lab', ele', att', root')$ *s.t.* $T' \lhd X$ and $Y \subseteq tuples_X(T')$. We use T' to define an XML tree $T = (V, lab, ele, att, root)$ *s.t.* $trees_X(Y) = [T]$.
For each $v \in V'$, if there is $t \in Y$ and $p \in paths(X)$ *s.t.* $t.p = v$, then v is included in V. Furthermore, for each $v \in V$, $lab(v)$ is defined as $lab'(v)$, $ele(v) = [s_1, . . . , s_n]$, where each $s_i = t'.p.S$ or $s_i = t'.p.\tau$ for some $t' \in Y$ and $\tau \in E$ *s.t.*, $t'.p = v$. For each $@l \in A$ *s.t.*, $t'.p.@l \neq \varphi$ and $t'.p = v$ for some $t' \in Y$, $att(v, @l)$ is defined as $t'.p.@l$. Finally, $root$ is defined as $root'$. It is easy to see that $trees_X(Y) = [T]$.

(b) Let $t \in Y$ and T be an XML tree *s.t.* $trees_X(Y) = [T]$. If $t \in tuples_X([T])$, then the property holds trivially. Suppose that $t \notin tuples_X([T])$. Then, given that $tree_X(t) \leq T$, there is $t' \in tuples_X([T])$ *s.t.* $t \subset t'$. In either case, we conclude that there is $t' \in tuples_X(trees_X(Y))$ *s.t.* $t \subseteq t'$. □

The example below shows that it could be the case that $tuples_X(trees_X(Y))$ properly dominates Y, that is, $Y \subseteq^b tuples_X(trees_X(Y))$ and $tuples_X(trees_X(Y)) \not\sqsubseteq^b Y$. In particular, this example shows that the inverse of Theorem 1 does not hold, that is, $tuples_X(trees_X(Y))$ is not necessarily equal to Y for every set of tree tuples Y, even if this set is X-compatible. Let X be as in example 5 and $t_1, t_2 \in \mathcal{T}(X)$ be defined as:

$$t_1.r = v_0 \qquad t_2.r = v_0$$
$$t_1.r.a = v_1 \qquad t_2.r.a = \varphi$$
$$t_1.r.b = \varphi \qquad t_2.r.b = v_2$$

Let t_3 be a tree tuple defined as:

$t_3.r = v_0$, $t_3.r.a = v_1$ and $t_3.r.b = v_2$.

Then, $tuples_X(trees_X(\{t_1, t_2\})) = \{t_3\}$ since $t_1 \subset t_3$ and $t_2 \subset t_3$, and, therefore, $\{t_1, t_2\} \subseteq^b tuples_X(trees_X(\{t_1, t_2\}))$ and $tuples_X(trees_X(\{t_1, t_2\})) \not\sqsubseteq^b \{t_1, t_2\}$.

IV. NORMAL FORMS OF XML SCHEMA

In this section, and by using the definitions of the previous sections, we present the normal forms of XML documents. Our goal is to see what relational concepts we can usefully apply to XML. Can the normal forms that guide database design be applied meaningfully to XML document design?

***Definition 12 (functional dependencies)*:** Given an XML Schema X, a *functional dependency* (*FD*) over X is an expression of the form: $S_1 \rightarrow S_2$ where $S_1, S_2 \subseteq paths(X)$, $S_1, S_2 \neq \varphi$. The set of all FDs over X is denoted by $FD(X)$.

- For $S \subseteq paths(X)$, and $t, t' \in \mathcal{T}(X)$, $t.S = t'.S$ means $t.p = t'.p\ \forall\ p \in S$. Furthermore, $t.S \neq \varphi$ means $t.p \neq \varphi\ \forall\ p \in S$.

***Definition 13*:** If $S_1 \rightarrow S_2 \in FD(X)$ and T is an XML tree *s.t.* $T \triangleleft X$ and $S_1 \cup S_2 \subseteq paths(T)$, we say that T *satisfies* $S_1 \rightarrow S_2$ (written $T \models S_1 \rightarrow S_2$), if $\forall\ t_1, t_2 \in tuples_X(T)$, $t_1.S_1 = t_2.S_1$ and $t_1.S_1 \neq \varphi \Rightarrow t_1.S_2 = t_2.S_2$.

- Note that: if tree tuples t_1, t_2 satisfy an FD $S_1 \rightarrow S_2$, then for every path $p \in S_2$, $t_1.p$ and $t_2.p$ are either both null or both not null.

***Definition 14*: :** If for every pair of tree tuples t_1, t_2 in an XML tree T, $t_1.S_1 = t_2.S_1$ implies they have a null value on some $p \in S_1$, then the FD is *trivially satisfied* by T.

- The previous definitions extends to the equivalence classes, since, for any FD f, and $T \equiv T'$, $T \models f$ *iff* $T' \models f$.
- We write $T \models \mathcal{F}$, for $\mathcal{F} \subseteq FD(X)$, if $T \models f$ for each $f \in \mathcal{F}$, and we write $T \models (X, \mathcal{F})$, if $T \models X$ and $T \models \mathcal{F}$.

Example 6: Consider the XML Schema in example 1, we have the following FDs. Note that, `cno` is a key of `course`:

`courses.course.@cno` $\rightarrow$ `courses.course` (FD1)

Another FD says that two distinct student subelements of the same course cannot have the same sno:

{`courses.course,courses.course.taken_by.student.@sno`} $\rightarrow$ `courses.course.taken_by.student` (FD2)

Finally, to say that two student elements with the same sno value must have the same name, we use

`courses.course.taken_by.student.@sno` $\rightarrow$ `courses.course.taken_by.student.name.S` (FD3)

***Definition 15*:** Given XML Schema X, a set $\mathcal{F} \subseteq FD(X)$ and $f \in FD(X)$, we say that $(X, \mathcal{F})$ *implies* f, written $(X, \mathcal{F}) \vdash f$, if for any tree T with $T \models X$ and $T \models \mathcal{F}$, it is the case that $T \models f$. The set of all FDs *implied* by $(X, \mathcal{F})$ will be denoted by $(X, \mathcal{F})^+$.

***Definition 16*:** an FD f is *trivial* if $(X, \varphi) \vdash f$.

A. Primary and Foreign Keys of XML Schema

In this section, we present the definitions of the *primary* and *foreign keys* of the XML Schema. We observe that while there are important differences between the XML and relational models, much of the thinking that commonly goes into relational database design can be applied to XML Schema design as well.

***Definition 17 (key, foreign key, and superkey)*:** Let $X = (E, A, M, P, r, \sum)$ be XML Schema, a constraint $\sum$ over X has one of the following forms:

- ***key***: $e(\mathtt{l}) \rightarrow e$, where $e \in E$, and $\mathtt{l}$ is a set of attributes in $P(e)$. It indicates that the set $\mathtt{l}$ of attributes is a *key* of e elements .
- ***foreign key***: $e_1(\mathtt{l}_1) \subseteq e_2(\mathtt{l}_2)$ and $e_2(\mathtt{l}_2) \rightarrow e_2$ where $e_1, e_2 \in E$, and $\mathtt{l}_1$, $\mathtt{l}_2$ are non-empty sequences of attributes in $P(e_1)$, $P(e_2)$, respectively, and moreover $\mathtt{l}_1$ and $\mathtt{l}_2$ have the same length. This constraint indicates that $\mathtt{l}_1$ is a *foreign key* of e_1 elements referencing key $\mathtt{l}_2$ of e_2 elements.
- A constraint of the form $e_1(\mathtt{l}_1) \subseteq e_2(\mathtt{l}_2)$ is called an *inclusion constraint*.
- Observe that a *foreign key* is actually a pair of constraint, namely an inclusion constraint $e_1(\mathtt{l}_1) \subseteq e_2(\mathtt{l}_2)$ and a key $e_2(\mathtt{l}_2) \rightarrow e_2$.
- ***superkey***: suppose that, $e \subseteq E$, and for any two *distinct* paths p_1 and p_2 in the XML Schema X, we have the constraint that: $p_1(e) \neq p_2(e)$. The subset e is called a *superkey* of X.
- Every XML Schema has at least one default superkey - *the set of all its elements*.

B. First Normal Form for XML Schema (X-1NF)

First normal form (1NF) is now considered to be a part of the formal definition of a relation in the basic relational database model. Historically, it was defined as: *"The domain of an attribute in a tuple must be a single value from the domain of that attribute"* [13].

Of course, XML is hierarchical by nature. An XML *"tuple"* can vary from first normal form in several ways, all of them are valid by means of data modeling:

1. It can have varying numbers of fields and default values for attributes.
2. It can have multiple values for a field, through the `maxOccurs` attribute for particles.
3. It can have choices of field types instead of a straight sequence or conjunction.
4. Fields can be of *complex type*.

C. Second Normal Form of XML Schema (X-2NF)

X-2NF is based on the concept of *full functional dependency*.

Definition 18: A FD $S_1 \rightarrow S_2$, where S_1, $S_2 \subseteq$ *paths*(X) is called *full FD*, if removal of any element's path p from S_1, means that the dependency does not hold any more, (*i.e.*, for any $p \in S_1$, $(S_1\text{-}\{p\})$ does not functional determine S_2).

Definition 19: A FD $S_1 \rightarrow S_2$ is called *partial dependency* if, for some $p \in S_1$, $(S_1\text{-}\{p\}) \rightarrow S_2$ is hold.

Example 7: Consider the following part of XML Schema called "Emp_Proj"

```
<xs:complexType name "Emp_Proj">
      <xs:sequence>
       <xs: element name = "Sss" type ="string" />
        <xs: element name = "Pnumber" type ="string" />
        <xs: element name = "Hours" type ="string" />
        <xs: element name = "Ename" type ="string" />
        <xs: element name = "Pname" type ="string" />
        <xs: element name = "Plocation" type ="string" />
      </xs:sequence>
<xs:complexType>
<xs: key name = "empSssKey" >
      <xs: selector xpath = "Emp_Proj" />
      <xs: field xpath = "Sss" />
</xs:key>
<xs: key name = "ProjectNoKey" >
      <xs: selector xpath = "Emp_Proj" />
      <xs: field xpath = "Pnumber" />
</xs:key>
```

with the following FDs:

FD1:{ `Emp_Proj.Sss,Emp_Proj.Pnumber`}$\rightarrow$ `Emp_Proj.Hours`

FD2: `Emp_Proj.Sss` $\rightarrow$ `Emp_Proj.Ename`

FD3: `Emp_Proj.Pnumber` $\rightarrow${`Emp_Proj.Pname, Emp_Proj.Plocation`}

Note that:

- FD1 is a full FD (neither `Emp_Proj.Sss` → `Emp_Proj.Hours` nor `Emp_Proj.Pnumber` → `Emp_Proj.Hours` holds).
- The FD: {`Emp_Proj.Sss, Emp_Proj.Pnumber`} → `Emp_Proj.Ename` is partial because `Emp_Proj.Sss` → `Emp_Proj.Ename` holds.

Definition 20 (X-2NF): An XML Schema $X = (E, A, M, P, r, \Sigma)$ is in *second normal form* (X-2NF) if every elements $e \in E$ and attributes $l \subseteq P(e)$ are fully functionally dependent on the key elements of X.

- The test for X-2NF involves testing for FDs whose left-hand side are part of the primary key. If the primary key contain a single element's path, the test need not be applied at all.

```
<xs:complexType name "EP1">
      <xs:sequence>
        <xs: element name = "Sss" type ="string" />
        <xs: element name = "Pnumber" type ="string" />
        <xs: element name = "Hours" type ="string" />
      </xs: element>
      </xs:sequence>
<xs:complexType>
<xs:complexType name "EP2">
      <xs:sequence>
        <xs: element name = "Sss" type ="string" />
        <xs: element name = "Ename" type ="string" />
      </xs:sequence>
<xs:complexType>
<xs:complexType name "EP3">
      <xs:sequence>
        <xs: element name = "Pnumber" type ="string" />
        <xs: element name = "Pname" type ="string" />
        <xs: element name = "Plocation" type ="string" />
        </xs: element>
      </xs:sequence>
<xs:complexType>
<xs: key name = "empSssKey" >
      <xs: selector xpath = "EP1" />
      <xs: field xpath = "Sss" />
</xs:key>
<xs: key name = "ProjectNoKey" >
      <xs: selector xpath = "EP1" />
      <xs: field xpath = "Pnumber" />
</xs:key>
<xs: key name = "emp2SssKey" >
      <xs: selector xpath = "EP2" />
      <xs: field xpath = "Sss" />
</xs:key>
<xs: key name = "Project3NoKey" >
      <xs: selector xpath = "EP3" />
      <xs: field xpath = "Pnumber" />
</xs:key>
```

Example 8: The XML Schema `Emp_Proj` in the above example is in X-1NF but is not in X-2NF. Because the FDs FD2 and FD3 make `Emp_Proj.Ename`, `Emp_Proj.Pname`, and `Emp_Proj.Plocation` partially dependent on the primary key {`Emp_Proj.Sss, Emp_Proj.Pnumber`} of `Emp_Proj`, thus violating the X-2NF test.

- Hence, the FDs FD1, FD2, and FD3 lead to the decomposition of XML Schema `Emp_Proj` to the following XML Schemas `EP1`, `EP2`, and `EP3`:

D. Third Normal Form of XML Schema (X-3NF)

X-3NF is based on the concept of *transitive dependency*.

Definition 21: A FD $S_1 \rightarrow S_2$, where S_1, $S_2 \subseteq$ *paths*(X) is *transitive dependency* if there is a set of paths Z (that is neither a key nor a subset of any key of X), and both $S_1 \rightarrow Z$ and $Z \rightarrow S_2$ hold.

Example 9: consider the following XML Schema called "Emp_Dept":

`Emp_Dept(`*`Ssn`*`, `*`Ename, Bdate, Address, Dnumber, Dname, DmgrSsn`*`)`

```
<xs:complexType name "Emp_Dept">
      <xs:sequence>
       <xs: element name = "Sss" type ="string" />
       <xs: element name = "Ename" type ="string" />
       <xs: element name = "Bdate" type ="date" />
        <xs: element name = "Address" type ="string" />
        <xs: element name = "Dnumber" type ="string" />
       <xs: element name = "Dname" type ="string" />
       <xs: element name = "DmgrSsn" type ="string" />
      </xs:sequence>
<xs:complexType>
<xs: key name = "empSssKey" >
      <xs: selector xpath = "Emp_Dept" />
      <xs: field xpath = "Sss" />
</xs:key>
```

with the following FDs:

FD1:`Emp_Dept.Ssn`→{`Emp_Dept.Ename, Emp_Dept.Bdate, Emp_Dept.Address, Emp_Dept.Dnumber` }

FD2:`Emp_Dept.Dnumber` → {`Emp_Dept.Dname, Emp_Dept.DmgrSsn`}

Note that:

- The dependency:

`Emp_Dept.Ssn`→ `Emp_Dept.DmgrSsn` is transitive through `Emp_Dept.Dnumber` in `Emp_Dept`, because both the FDs:

`Emp_Dept.Ssn` → `Emp_Dept.Dnumber` and

`Emp_Dept.Dnumber` → `Emp_Dept.DmgrSsn`

hold, and `Emp_Dept.Dnumber` is neither a key itself nor a subset of the key of `Emp_Dept`.

***Definition 22 (X-3NF)*:** An XML Schema $X = (E, A, M, P, r, \Sigma)$ is in *third normal form* (X-3NF) if it satisfies X-2NF and no (elements $e \in E$ or $l \subseteq P(e)$) is transitively dependent on the key elements of X.

Example 10: The XML Schema `Emp_Dept` in the above example is in X-2NF (since no partial dependencies on a key element exist), but `Emp_Dept` is not in X-3NF. Because of the transitive dependency of `Emp_Dept.DmgrSsn` (and also `Emp_Dept.Dname`) on `Emp_Dept.Ssn` via `Emp_Dept.Dnumber`.

- We can normalize `Emp_Dept` by decomposing it into the following two XML Schemas `ED1`, and `ED2`:
  `ED1`(*<u>Ssn</u>, Ename, Bdate, Address, Dnumber*)
  `ED2`(*<u>Dnumbe</u>r, Dname, DmgrSsn*)

```
<xs:complexType name "ED1">
      <xs:sequence>
       <xs: element name = "Sss" type ="string" />
       <xs: element name = "Ename" type ="string" />
       <xs: element name = "Bdate" type ="date" />
        <xs: element name = "Address" type ="string" />
        <xs: element name = "Dnumber" type ="string" />
     </xs:sequence>
<xs:complexType>
<xs:complexType name "ED2">
      <xs:sequence>
       <xs: element name = "Dnumber" type ="string" />
       <xs: element name = "Dname" type ="string" />
       <xs: element name = "DmgrSsn" type ="string" />
      </xs:sequence>
<xs:complexType>

<xs: key name = "empSssKey" >
      <xs: selector xpath = "ED1" />
      <xs: field xpath = "Sss" />
</xs:key>

<xs: key name = "deptNoKey" >
      <xs: selector xpath = "ED2" />
      <xs: field xpath = "Dnumber" />
</xs:key>
```

E. Boyce-Codd Normal Form of XML Schema (X-BCNF)

Boyce-Codd Normal form of XML Schema (X-BCNF), proposed as a similar form as X-3NF, but it was found to stricter than X-3NF, because every XML Schema in X-BCNF is also in X-3NF, however, an XML Schema in X-3NF is not necessarily in X-BCNF. The formal definitions of BCNF differs slightly from the definition of X-3NF

***Definition 23 (X-BCNF)*:** An XML Schema $X = (E, A, M, P, r, \Sigma)$ is in *Boyce-Codd Normal Form* (X-BCNF) if whenever a nontrivial FD $S_1 \rightarrow S_2$ holds in X, where $S_1, S_2 \subseteq$ *paths*(X), then S_1 is a *superkey* of X.

Also, we can consider the following definition of X-BCNF:

***Definition 24*:** Given XML Schema X and $F \subseteq FD(X)$, (X, F) *is in X-BCNF iff* for every nontrivial FD $f \in (X, F)^+$ of the form $S \rightarrow p.@l$ or $S \rightarrow p.S$, it is the case that, $S \rightarrow p \in (X, F)^+$.

In definition 24, we suppose that, f is a nontrivial FD. Indeed, the trivial FD $p.@l \rightarrow p.@l$ is always in $(X, F)^+$, but often $p.@l \rightarrow p \notin (X, F)^+$, which does not necessarily represent a bad design.
To show how X-BCNF distinguishes good XML design from bad design, we consider example 1 again, when only functional dependencies are provided.

Example 11: Consider the XML Schema from example 1 whose FDs are FD1, FD2, and FD3, shown in example 6. FD3 associates a unique name with each student number, which is therefore redundant. The design is *not* in X-BCNF, since it contains FD3 but does not imply the functional dependency:

```
courses.course.taken_by.student.@sno →
        courses.course.taken_by.student.name
```

To solve this problem, we gave a revised XML Schema in example 1. The idea was to create a new element `info` for storing information about students. That design satisfies FDs, FD1, FD2, as well as

```
courses.info.number.@sno → courses.info
```

and can be easily verified to be in X-BCNF.

V. NORMALIZATION ALGORITHM

The goal of this section is to show how to transform an XML Schema X and a set of FDs F into a new specification (X', F') that is in X-BCNF and contains the same information.

Throughout the section, we assume that the XML Schemas are non-recursive. This can be done without any loss of generality. Notice that in a recursive XML Schema X, the set of all paths is infinite. We make an additional assumption that all the FDs are of the form:

$$\{q, p_1.@l_1, \ldots, p_n.@l_n\} \to p.$$

That is, they contain at most one element path on the left-hand side. While constraints of the form $\{q, q', \ldots\}$ are not forbidden, they appear to be quite unnatural. Furthermore, even if we have such constraints, they can be easily eliminated. To do so, we create a new attribute $@l$, remove $\{q, q'\} \cup S \to p$ and replace it by $q'.@l \to q'$ and $\{q, q'.@l\} \cup S \to p$.

We shall also assume that paths do not contain the symbol S (since $p.S$ can always be replaced by a path of the form $p.@l$).

A. The Decomposition Algorithm

For introducing the decomposition algorithm, we make the following assumption: if $S \to p.@l$ is an FD that causes a violation of X-BCNF, then every time that $p.@l$ is not null, every path in S is not null. This will make our presentation simpler.

Given XML Schema X and a set of FDs F, a nontrivial FD $S \to p.@l$ is called *anomalous*, over (X, F), if it violates X-BCNF; that is, $S \to p.@l \in (X, F)^+$ but $S \to p \notin (X, F)^+$. A path on the right-hand side of an anomalous FD is called an *anomalous path*, and the set of all such paths is denoted by $APath(X, F)$.

In this sub-section we present an X-BCNF decomposition algorithm that combines two basic ideas: creating a new element type, and moving an attribute.

1) Creating New Element Types

Let $X = (E, A, M, P, r, \Sigma)$ be XML Schema and F a set of FDs over X. Assume that (X, F) contains an anomalous FD $\{q, p_1.@l_1, \ldots, p_n.@l_n\} \to p.@l$, where $q \in EPaths(X)$ and $n \geq 1$. For example, the "*university*" database shown in Example 1 contains an anomalous FD of this form (considering `name.S` as an attribute of `student`):

```
{courses, courses.course.taken_by.student.@sno} →
courses.course.taken_by.student.name.S.          (1)
```

To eliminate the anomalous FD, we create a new element type τ as a child of the last element of q, we make $\tau_1, \ldots, \tau_n$ its children, where $\tau_1, \ldots, \tau_n$ are new element types, we remove $@l$ from the list of attributes of $last(p)$ and we make it an attribute of τ and we make $@l_1, \ldots, @l_n$ attributes of $\tau_1, \ldots, \tau_n$, respectively, but without removing them from the sets of attributes of $last(p_1), \ldots, last(p_n)$, as shown in Figure 4.

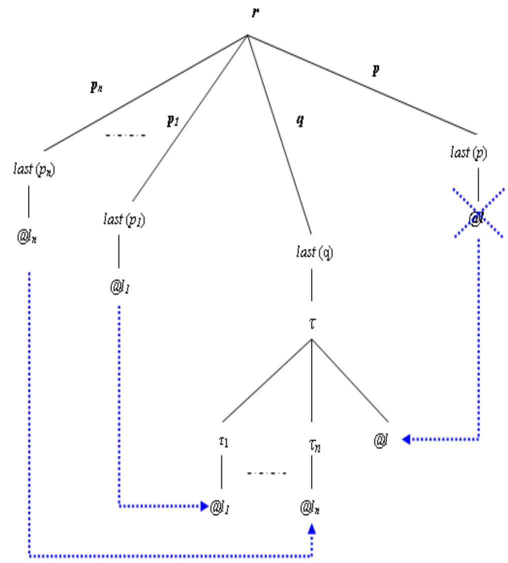


Figure 4. creating new element types

For instance, to eliminate the anomalous functional dependency (1), in example 1, we create a new element type `info` as a child of `courses`, we remove `name.S` from `student` and we make it an "attribute" of `info`, we create an element type `number` as a child of `info` and we make `@sno` its attribute. We note that we do not remove `@sno` as an attribute of `student`.

Formally, if $\tau, \tau_1, \ldots, \tau_n$ are element types that are not in E, the new XML Schema, denoted by $X[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l]]$, is $(E', A, M', P', r, \Sigma)$, where $E' = E \cup \{\tau, \tau_1, \ldots, \tau_n\}$ and

1. if $M(last(q))$ is a regular expression s, then $M'(last(q))$ is defined as the concatenation of s and τ^*, that is (s, τ^*). Furthermore, $M'(\tau)$ is defined as the concatenation of $\tau_1^*, \ldots, \tau_n^*$, $M'(\tau_i) = \varepsilon$, for each $i \in [1, n]$, and $M'(\tau') = M(\tau')$, for each $\tau' \in E - \{last(q)\}$.

2. $P'(\tau) = \{@l\}$, $P'(\tau_i) = \{@l_i\}$, for each $i \in [1, n]$, $P'(last(p)) = P(last(p)) - \{@l\}$ and $P'(\tau') = P(\tau')$ for each $\tau' \in E - \{last(p)\}$.

After transforming X into a new XML Schema $X' = X[p.@l := q.\tau\,[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l\,]]$, a new set of functional dependencies is generated. Formally, $F[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l\,]]$ is a set of FDs over X' defined as the union of the following sets of constraints:

1. $S_1 \rightarrow S_2 \in (X, F)^+$ with $S_1 \cup S_2 \subseteq paths(X')$.
2. Each FD over $q, p_i, p_i.@l_i$ ($i \in [1, n]$) and $p.@l$ is transferred to τ and its children. That is, if $S_1 \cup S_2 \subseteq \{q, p_1, \ldots, p_n, p_1.@l_1, \ldots, p_n.@l_n, p.@l\}$ and $S_1 \rightarrow S_2 \in (X, F)^+$, then we include an FD obtained from $S_1 \rightarrow S_2$ by changing p_i to $q.\tau.\tau_i$, $p_i.@l_i$ to $q.\tau.\tau_i.@l_i$, and $p.@l$ to $q.\tau.@l$.
3. $\{q, q.\tau.\tau_1.@l_1, \ldots, q.\tau.\tau_n.@l_n\} \rightarrow q.\tau$, and $\{q.\tau, q.\tau.\tau_i.@l_i\} \rightarrow q.\tau.\tau_i$ for $i \in [1, n]$.

2) *Moving Attributes*

Let $X = (E, A, M, P, r, \sum)$ be XML Schema and F a set of FDs over X. Assume that (X, F) contains an anomalous FD $q \rightarrow p.@l$, where $q \in EPaths(X)$. To eliminate the anomalous FD, we move the attribute $@l$ from the set of attributes of the last element of p to the set of attributes of the last element of q, as shown in Figure 5.

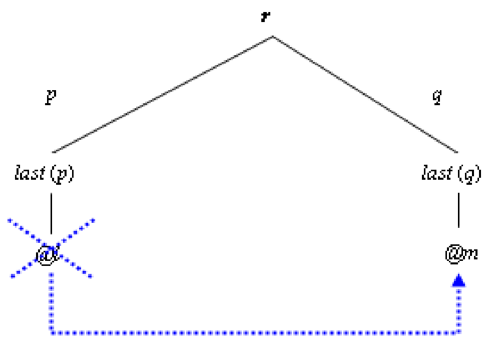


Figure 5. Moving Attributes

Formally, to eliminate the anomalous functional dependency, we consider the new XML Schema, $X[p.@l := q.@m]$, where $@m$ is an attribute, is defined to be $(E, A', M, P', r, \sum)$, where $A' = A \cup \{@m\}$, $P'(last(q)) = P'(last(q)) \cup \{@m\}$, $P'(last(p)) = P(last(p)) - \{@l\}$ and $P'(\tau') = P(\tau')$ for each $\tau' \in E - \{last(q), last(p)\}$.

After transforming X into a new XML Schema $X[p.@l := q.@m]$, a new set of functional dependencies is generated. Formally, the set of FDs $F[p.@l := q.@m]$ over $X[p.@l := q.@m]$ consists of all FDs $S_1 \rightarrow S_2 \in (X, F)^+$ with $S_1 \cup S_2 \subseteq paths(X[p.@l := q.@m])$.

3) *The Algorithm*

The algorithm applies the two transformations introduced in the previous sections until the schema is in X-BCNF, as shown in Figure 6.

The algorithm shows in Figure 6, involves FD implication, that is, testing membership in $(X, F)^+$ (and consequently testing X-BCNF and (X, F)-minimality). Since each step reduces the number of anomalous paths, then we obtain:

Step 1 : If (X, F) is in X-BCNF then return (X, F) otherwise go to step 2.
Step 2 : *Moving attributes:-*
If there is an anomalous FD $S \rightarrow p.@l$ and $q \in EPaths(X)$ such that $q \in S$, and $q \rightarrow S \in (X, F)^+$, then
Step 2.1: Choose a new attribute @m
Step 2.2: $X := X[p.@l := q.@m]$
Step 2.3: $F := [p.@l := q.@m]$
Step 2.4: *Go to Step 1.*
Step 3: Creating new element types:-
Choose a (X, F)-minimal anomalous FD $S \rightarrow p.@l$, where $S = \{q, p_1.@l_1, \ldots, p_n.@l_n\}$
Step 3.1: Create new element types $\tau, \tau_1, \ldots, \tau_n$
Step 3.2: $X := X[p.@l := q.\tau\,[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l\,]]$,
Step 3.3: $F := F[p.@l := q.\tau[\tau_1.@l_1, \ldots, \tau_n.@l_n, @l\,]]$
Step 3.4: Go to *Step 1*

Figure 6. X-BCNF decomposition algorithm.

Proposition 4. *The X-BCNF decomposition algorithm terminates, and outputs a specification (X, F) in X-BCNF.*

VI. CONCLUSION AND FUTURE WORKS

We address the problem of schema design and normalization in XML databases model. The main contribution of this paper are the proposed normal forms for XML Schema, and the decomposition algorithm that used to convert any XML Schema into normalized one, that satisfies X-BCNF.

The decomposition algorithm can be improved in various ways, and we plan to work on making it more efficient. We also would like to find a complete classification of the complexity of the FD implication problem for various classes of XML Schemas. We plan to work on extending XML Schema normal form to more powerful normal forms, in particular by taking into account multi-valued dependencies.

REFERENCES

[1] W3C 2001 XML Schema: http://www.w3.org/XML/Schema.
[2] Kanne, C.C. and Moerkotte, G. Efficient storage of XML data. In Proceedings of the 16th International Conference on Data Engineering, 2000.

[3] Tatarinov, I., Ives, Z., Halevy, A., and Weld, D. Updating XML. In Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM, New York, 413–424, 2001.

[4] Paredaens, J., DE Bra, P., Gyssens, M. & Van Gucht, d., The Structure of the Relational Database Model, Springer-Verlag, 1989.

[5] Thalheim, B., Dependencies in Relational Databases, Teubner-Verlag, 1991.

[6] Embley, D. and Mok,W. Y. Developing XML documents with guaranteed "good" properties. In Proceedings of the 20th International Conference on Conceptual Modeling. 426–441, 2001.

[7] Arenas, M. and Libkin, L. An information-theoretic approach to normal forms for relational, 2003.

[8] Lee, .L., Ling, T. W., and Low, W. L. Designing functional dependencies for XML. In Proceedings of the 8th International Conference on Extending Database Technology. 124–141, 2002.

[9] Marcelo Arenas and Leonid Libkin, "A Normal Form for XML Documents". ACM Transactions on Database Systems, Vol. 29, No. 1, Pages 195–232, March 2004. (doi:10.1145/974750.974757)

[10] Buneman, P., Jung, A., and Ohori, A. 1991. Using power domains to generalize relational databases. *Theoret. Comput. Sci. 91*, 1, 23–55. (doi:10.1016/0304-3975(91)90266-5)

[11] Grahne, G. 1991. *The Problem of Incomplete Information in Relational Databases*. Springer-Verlag, New York, Cambridge, Mass.

[12] Gunter, C. 1992. Semantics of Programming Languages: Structures and Techniques.MIT Press, Cambridge, Mass.

[13] Ramez Elmasri and Shamkant B. Navathe. Fundamentals of Database System. Addison-Wesley, third edition, 2000.

[14] Murali M., and Dongwon L. XML to Relational Conversion using Theory of Regular Tree Grammers. Proc. of the 28th VLDB Conference, Hong Kong, China, 2002.

AUTHORS

Hosam F. El-Sofany received his Ph.D. and M. Sc. degree in Computer Science from Ain Shams University, Cairo, Egypt. He is currently a Lecturer at the Department of Engineering and Computer Science, College of Engineering, Qatar University, Qatar. He have a strong technical background including: designing and implementing Web-based systems. He published many research papers related to the E-learning technology in various International Journals and conferences. His research is focused on E-Learning, M-Learning, XML Databases, Databases Systems, and Semantic Web Applications. (email: helsofany@qu.edu.qa)

Professor Samir Abou El-Seoud received his BSc degree in Physics, Electronics and Mathematics from Cairo University in 1967, his Higher Diplom in Computing from Technical University of Darmstadt (TUD) -Germany in 1975 and his Doctor of Science from the same University (TUD) in 1979. Professor El-Seoud helds different academic positions at TUD Germany. Letest Full-Professor in 1987. Outside Germany Professor El-Seoud spent different years as a Full-Professor of Computer Science at SQU – Oman and Qatar University and acted as a Head of Computer Science for many years. At industrial institutions, Professor El-Seoud worked as Scientific Advisor and Consultant for the GTZ in Germany and was responsible for establishing a postgraduate program leading to M.Sc. degree in Computations at Colombo University / Sri-Lanka (2001 – 2003). He also worked as Application Consultant at Automatic Data Processing Inc., Division Network Services in Frankfurt/Germany (1979 – 1980). Professor El-Seoud joined PSUT in 2004. Currently, he is the Chairman of the Computer Science Dept. at PSUT. (email: selseoud@psut.edu.jo)

Manuscript received 15 December 2008. Published as submitted by the authors.