Normalization Algorithms from Relational to XML Databases

Hosam F. El-Sofany¹, Fayed F. M. Ghaleb², Sameh S. Daoud², Ahmad M. Hasna¹, Jihad M AlJa'am¹, Quatibah Malluhi¹, and Samir A. El-Seoud³

¹Department of Computer Science and Engineering, College of Engineering, Qatar University.
²Department of Mathematics, Faculty of Science, Ain Shams University.
³Computer Science Department, Princess Sumaya University for Technology.

Key words: XML, Database Design, Functional Dependencies, Redundancy, Normal Forms, Normalization, Algorithms.

Abstract

With the present of XML and its use as a database, schema design and normal form theory has attracted novel research interest. In this paper we address 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 defined the functional dependencies and normal forms for XML Schema, we present the decomposition algorithm for converting any XML Schema into normalized one, that satisfies X-BCNF.

1 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.

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.

**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.

```xml
<?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>
```
An example of an XML document 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*

This constraint (which looks very much like a functional dependency), causes the document to store redundant information: for example, the name Deere for student st1 is stored twice. And just 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. Therefore, the updated XML Schema will take the following form:

```
<?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:complexType>
  </xs:element>
</xs:schema>
```
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 reminds us of the bad relational design caused by nonkey functional dependencies, and how the database designer solve this problem by modifying the schema. Our goal is to show how to detect anomalies of those kinds, and to transform documents in a lossless fashion into ones that do not suffer from those problems.

2 Primarily Definitions

Assume that we have the following disjoint sets:

• $\mathcal{E}$: set of element names,
• $\mathcal{A}$: 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 $\emptyset$ and S represent element type declarations EMPTY (null) and #PCDATA, respectively.

We first describe the formal definitions of XML Schema (XSchema) and the conforming of XML tree to XSchema. We define the XSchema based on regular tree grammar theory that introduced in [14].
Definition 1 (XSschema): An XSschema is denoted by 6-tuple: \( X = (E, A, M, P, r, \Sigma) \), where:
- \( E \) is a finite set of element names in \( \hat{E} \),
- \( A \) is a function from an element name \( e \in E \) to a set of attribute names \( a \in \hat{A} \),
- \( M \) is a function from an element name \( e \in E \) to its element type definition:
  \[ \{ \text{i.e.}, \; M(e) = \alpha, \; \text{where} \; \alpha \; \text{is a regular expression:} \]
  \[ \alpha ::= e \mid t \mid \alpha + \alpha \mid \alpha \mid \alpha^* \mid \alpha^+ \]
  \[ \text{where} \; \varepsilon \; \text{denotes the empty element}, \; t \in DT, \; "\cdot" \; \text{for the union}, \; \alpha^* \; \text{for the Kleene closure,} \; \alpha^+ \; \text{for} \; (\alpha + \varepsilon) \; \text{and} \; \alpha^* \; \text{for} \; (\alpha, \alpha^*) \]
- \( P \) is a function from an attribute name \( a \) to its attribute type definition:
  \[ \{ \text{i.e.}, \; P(a) = \beta, \; \text{where} \; \beta \; \text{is a 4-tuple} \; (t, n, d, f), \; \text{where:} \]
  - \( t \in DT \),
  - \( n \) is either "\varepsilon" (nullable) or "\varepsilon?" (not nullable),
  - \( d \) is a finite set of valid domain values of \( a \) or \( \varepsilon \) if not known, and
  - \( f \) is a default value of \( a \) or \( \varepsilon \) if not known.
- \( r \subseteq E \) is a finite set of root elements,
- \( \Sigma \) 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 XSschema): Given an XSschema \( X = (E, A, M, P, r, \Sigma) \), 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 XSschema 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 \( V \times Att \rightarrow Str \). For each \( v \in V \), the set \( \{ @l \in Att \mid att(v, @l) \} \) is defined) is required to be finite.
- \( root \subseteq V \) is called the root of \( T \).

The parent-child edge relation on \( V \), \( \{(v_1, v_2) \mid v_2 \text{ occurs in ele}(v_1)\} \), 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:
\[ p_1 \ldots p_n \text{ with } p_1, \ldots, p_{n-1} \in \hat{E} \text{ and } p_n \in \hat{E} \cup \hat{A} \cup \{S\} \]
is a path in \( T \) if there are vertices \( v_1 \ldots v_{n-1} \in V \) s.t.:
- \( v_1 = root, v_{i+1} \text{ is a child of } v_i (1 \leq i \leq n - 2), lab(v_i) = p_i (1 \leq i \leq n - 1). \)
- If \( p_n \in E \), 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 \( T \).
We next give a definition of a tree conforming to the XML schema \( T \vdash X \), and a tree compatible with \( X \).

**Definition 5:** Given an XML schema \( X = (E, A, M, P, r, \Sigma) \) 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 \( (T \vdash X) \) 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 \( @l \in P(lab(v)) \).
- \( \text{lab}(\text{root}) = r \).
- We say that \( T \) is compatible with \( X \) (written \( T \preceq X \)) iff \( \text{paths}(T) \subseteq \text{paths}(X) \).
- Clearly, \( T \vdash X \) implies \( T \) is compatible with \( X \).

**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 \( T_1 \leq T_2 \) if:

- \( V_1 \subseteq V_2 \).
- \( \text{root}_1 = \text{root}_2 \).
- \( lab_2|_{V_1} = lab_1 \).
- \( att_2|_{V_1 \times A_{str}} = att_1 \).
- \( \forall v \in V_1 \), \( ele_1(v) \) is a sub-list of a permutation of \( ele_2(v) \).

**Definition 7:** The relation \( T_1 \leq T_2 \) is pre-order, which gives rise to an equivalence relation: \( T_1 \equiv T_2 \) iff \( T_1 \leq T_2 \) and \( T_2 \leq T_1 \). That is, \( T_1 \equiv T_2 \) iff \( T_1 \) and \( T_2 \) are equal as unordered trees.

- We define \([T]\) to be the \( =\)-equivalence class of \( T \).
- We write: \([T] \vdash X\) if \( T_1 \vdash X \) for some \( T_1 \in [T] \).
- It is easy to see that for any \( T_1 \equiv T_2 \), \( \text{paths}(T_1) = \text{paths}(T_2) \), hence, \( T_1 \preceq X \) iff \( T_2 \preceq X \).
- We shall also write \( T_1 < T_2 \) when \( T_1 \leq T_2 \) and \( T_2 \not< T_1 \).

In the following definitions, we extend the notion of tuple for relational databases to the XML model. In addition, we extend the introduced notion by \cite{9}, that based on XML DTD, to cover the XML Schema instead.

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 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 \( X = (E, A, M, P, r, \Sigma) \), a tree tuple \( t \in X \) is a function, \( t: \text{paths}(X) \rightarrow Vert \cup Str \cup \{\varphi\} \) such that:

- For \( p \in E\text{Paths}(X) \), \( t(p) \in Vert \cup \{\varphi\} \), and \( t(r) \neq \varphi \)
- For \( p \in \text{paths}(X) \setminus E\text{Paths}(X) \), \( t(p) \in Str \cup \{\varphi\} \)
- 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 \( t(p_2) = \varphi \)
- \( \{p \in \text{paths}(X) | t(p) \neq \varphi \} \) is finite

\( \Pi(X) \) 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 $\text{paths}(X)$ such as:

$t(\text{courses}) = v_0$
t($\text{courses}.\text{course}$) = $v_1$
t($\text{courses}.\text{course}.@cno$) = csc200
t($\text{courses}.\text{course}.\text{title}$) = $v_2$
t($\text{courses}.\text{course}.\text{taken by}$) = $v_3$
t($\text{courses}.\text{course}.\text{taken by}.\text{student}$) = $v_4$
t($\text{courses}.\text{course}.\text{taken by}.\text{student}.@sno$) = st1
t($\text{courses}.\text{course}.\text{taken by}.\text{student}.\text{name}$) = $v_5$
t($\text{courses}.\text{course}.\text{taken by}.\text{student}.\text{name}.S$) = Deere
t($\text{courses}.\text{course}.\text{taken by}.\text{student}.\text{grade}$) = $v_6$
t($\text{courses}.\text{course}.\text{taken by}.\text{student}.\text{grade}.S$) = A+

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

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

We note that, in this definition, the lexicographic order is arbitrary, and it is chosen simply because an XML tree must be ordered.

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:

```
\text{csc200}
\text{Automata Theory}
\text{Deere}
\text{st1}
\text{A+}
```

We would like to describe XML trees in terms of the tuples they contain [9]. For this, we need to select tuples containing the maximal amount of information. This is done via the usual notion of ordering on tuples (and relations) with nulls [10-12].

- 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 so is $t_2.p$, and $t_1.p \neq \varnothing$ implies $t_1.p = t_2.p$. 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:

\[ \forall t_1 \in Y \quad \exists t_2 \in Z \text{ such that, } t_1 \subseteq t_2. \]
Definition 10 (tuplesX): Given XML Schema X and an XML tree T such that T \sim X, tuplesX(T) is defined to be the set of maximal tree tuples t (with respect to \subseteq), such that \text{tree}_X(t) is subsumed by T, that is:

\[ \text{max}_{\subseteq} \{ t \in T(X) \mid \text{tree}_X(t) \leq T \} \]

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

Now, we'll 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 (treesX): Given XML Schema X and a set of tree tuples \( Y \subseteq \mathcal{T}(X) \), treesX(Y) is defined to be:

\[ \text{min}_{\leq} \{ T \mid T \sim X \text{ and } \forall t \in Y, \text{tree}_X(t) \leq T \} \]

Notice that, if \( T \in \text{trees}_X(Y) \) and \( T' \equiv T \), then \( T' \) is in treesX(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 tuplesX(T), up to equivalence =.

Finally, we will define the functional dependencies (FD), for XML model by using the tree tuples discussed above:

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 \( S_1, S_2 \) are finite nonempty subsets of paths(X),

- The set of all FDs over X is denoted by FD(X).
- For \( S \subseteq \text{paths}(X) \), and \( t, t' \in \mathcal{T}(X) \), \( t.S = t'.S \) means \( t.p = t'.p \ \forall p \in S \).
- Furthermore, \( t.S \neq \emptyset \) means \( t.p \neq \emptyset \ \forall p \in S \).

Definition 13: If \( S_1 \rightarrow S_2 \in \text{FD}(X) \) and T is an XML tree such that, \( T \sim X \) and \( S_1 \cup S_2 \subseteq \text{paths}(T) \), we say that T satisfies \( S_1 \rightarrow S_2 \) (written \( T \models S_1 \rightarrow S_2 \)) if: for every \( t_1, t_2 \in \text{tuples}_X(T), t_1.S_1 = t_2.S_1 \) and \( t_1.S_1 \neq \emptyset \) imply \( t_1.S_2 = t_2.S_2 \).

We observe 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 nonnull.

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 equivalence classes: i.e., for any FD \( f \), and \( T \equiv T' \), \( T \models f \iff T'.\models f \).
- We write \( T \models F \) for \( F \subseteq \text{FD}(X) \), if \( T \models f \) for each \( f \in F \), and we write \( T \models (X, F) \), if \( T \models X \) and \( T \models F \).

Example 4: Referring back to Example 1, we have the following FDs. Note that, cno is a key of course:

\[ \text{courses.course.cno} \rightarrow \text{courses.course} \quad \text{(FD1)} \]

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

\[ \text{courses.course.student.sno} \rightarrow \text{courses.course.student} \]
Finally, to say that two student elements with the same sno value must have the same name, we use
\[
\text{courses.course.taken_by.student.@sno} \rightarrow \text{courses.course.taken_by.student.name.S} \quad \text{(FD3)}
\]

**Definition 15**: Given XML Schema \(X\), a set \(F \subseteq FD(X)\) and \(f \in FD(X)\), we say that \((X, F)\) implies \(f\), written \((X, F) \models f\), if for any tree \(T\) with \(T \models X\) and \(T \models F\), it is the case that \(T \models f\).

The set of all FDs implied by \((X, F)\) will be denoted by \((X, F)^+\).

## 3 Primary and Foreign Keys in XML Schema

In this section, we’ll present the formal definition of the primary, foreign, and super keys of the XML Schema. We’ll use keys and FD concepts, mainly to introduce the concept of normal forms for XML Schema. Also, 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 16 (key, foreign key, and superkey)**: Let \(X = (E, A, M, P, r, \Sigma)\) be XML Schema, a constraint \(\Sigma\) over \(X\) has one of the following forms:

- **key**: \(e(\ell) \rightarrow e\), where \(e \in E\), and \(\ell\) is a set of attributes in \(P(e)\). It indicates that the set \(\ell\) of attributes is a key of elements of \(e\).
- **foreign key**: \(e_1(\ell_1) \subseteq e_2(\ell_2)\) and \(e_2(\ell_2) \rightarrow e_2\) where \(e_1, e_2 \in E\), and \(\ell_1, \ell_2\) are non-empty sequences of attributes in \(P(e_1), P(e_2)\), respectively, and moreover \(\ell_1\) and \(\ell_2\) have the same length. This constraint indicates that \(\ell_1\) is a foreign key of \(e_1\) elements referencing key \(\ell_2\) of \(e_2\) elements.
- A constraint of the form \(e_1(\ell_1) \subseteq e_2(\ell_2)\) is called an inclusion constraint.
- Observe that a foreign key is actually a pair of constraint, namely an inclusion constraint \(e_1(\ell_1) \subseteq e_2(\ell_2)\) and a key \(e_2(\ell_2) \rightarrow e_2\).
- **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.

**Example 5**: consider the following XML Schema, that describe a fragment of the database called "company":

```xml
<xs:element name = "company">
  <xs:complexType>
    <xs:sequence>
      <xs:element name = "dept" type ="dept"
        minOccurs="0" maxOccurs ="unbounded" />
      <xs:element name = "emp" type ="emp"
        minOccurs="0" maxOccurs ="unbounded" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType>
    <xs:key name = "deptnoKey" >
      <xs:selector xpath = "dept" />
      <xs:field xpath = "deptno" />
    </xs:key>
    <xs:key name = "empnoKey" >
```
Note that:

- XML Schema and relational database both use keys as means of identifying records.
- XML Schema has a similar feature, the key. Like relational database keys it can be defined to include one or more fields.
- One important difference, is that XML Schema keys cannot be defined by the identified type. Instead, they must be defined at some enclosing scope: specifically, as part of an element that contains instances of the identified type.
- The XML Schema in example 5, is corresponding to the relational schema, illustrated in Figure 5.

4 Normal Forms for XML Schema

In this section, we introduce the normal forms for XML documents. The 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?

4.1 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 "record" 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 \texttt{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.

- The last feature is the most apparent when looking at an XML document. An XML record is a tree, not a table.
- The second feature affects the relational database normal forms. It may at first seem that it's all to the good that we can model multiple children (of simple or complex type) directly under a parent, without having to resort to multiple tables and foreign keys just to express a simple one-to-many relationship.

### 4.2 Second Normal Form for XML Schema (X-2NF)

X-2NF is based on the concept of \textit{full functional dependency}.

**Definition 17:** A functional dependency (FD) \( S_1 \rightarrow S_2 \), where \( S_1, S_2 \subseteq \text{paths}(X) \) is called \textit{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 \( p \in S_1, (S_1-\{p\}) \) does not functional determine \( S_2 \).

**Definition 18:** A FD \( S_1 \rightarrow S_2 \) is called \textit{partial dependency} if, for some \( p \in S_1, (S_1-\{p\}) \rightarrow S_2 \) is hold.

**Definition 19 (X-2NF):** An XML Schema \( X = (E, A, M, P, r, \sum) \) is in second normal form (X-2NF) if every elements \( e \in E \) and \( 1 \subseteq P(e) \) 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.

### 4.3 Third Normal Form for XML Schema (X-3NF)

X-3NF is based on the concept of \textit{transitive dependency}.

**Definition 20:** A functional dependency \( S_1 \rightarrow S_2 \), where \( S_1, S_2 \subseteq \text{paths}(X) \) is \textit{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.

**Definition 21 (X-3NF):** An XML Schema \( X = (E, A, M, P, r, \sum) \) is in third normal form (X-3NF) if it satisfies X-2NF and no (elements \( e \in E \) or \( 1 \subseteq P(e) \)) is transitivity dependent on the key elements of \( X \).

### 4.4 Boyce-Codd Normal Form for XML Schema (X-BCNF)

Boyce-Codd Normal form for 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 22 (X-BCNF): An XML Schema $X = (E, A, M, P, r, \Sigma)$ is in Boyce-Codd Normal Form (X-BCNF) if whenever a nontrivial functional dependency $S_1 \rightarrow S_2$ holds in $X$, where $S_1, S_2 \subseteq \text{paths}(X)$, then $S_1$ is a superkey of $X$.

We can also consider the following formal definition of X-BCNF [9]:

Definition 23: Given XML Schema $X$ and $F \subseteq \text{FD}(X)$, $(X, F)$ 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, $S \rightarrow p \in (X, F)^+$.

The intuition is as follows. Suppose that $S \rightarrow p.\@l$ is in $(X, F)^+$. If $T$ is an XML tree conforming to $X$ and satisfying $F$, then in $T$ for every set of values of the elements in $S$, we can find only one value of $p.\@l$. Thus, for every set of values of $S$, we need to store the value of $p.\@l$ only once, in other words, $S \rightarrow p$ must be implied by $(D, F)$.

In this definition, 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 again the examples introduced in Section 3, when only functional dependencies are provided.

Example 6: Consider the XML Schema from Example 1 whose FDs are FD1, FD2, FD3, shown in Example 4. 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:

$$\text{courses.course.taken_by.student.@sno} \rightarrow \text{courses.course.taken_by.student.name}$$

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

$$\text{courses.info.number.@sno} \rightarrow \text{courses.info}$$

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

5. 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:

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

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 $\{q, q'\} \cup S \rightarrow p$ and replace it by $q'.@l \rightarrow q'$ and $\{q, q'.@l\} \cup S \rightarrow p$.

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 $p.@l$).
5.1 The Decomposition Algorithm

For introducing the decomposition algorithm, we make the following assumption: if \( S \rightarrow p.@l \) 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 \( S \rightarrow p.@l \) is called anomalous, over \((X, F)\), if it violates X-BCNF; that is, \( S \rightarrow p.@l \in (X, F)^+ \) but \( S \rightarrow p \notin (X, F)^+ \). 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 section we present an X-BCNF decomposition algorithm that combines two basic ideas: creating a new element type, and moving an attribute.

5.1.1 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 \( \{q, p_1.@l_1, \ldots, p_n.@l_n\} \rightarrow p.@l \), 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):

\[
\{\text{courses, courses.course.taken_by.student.@sno}\} \rightarrow \text{courses.course.taken_by.student.name.S.} \quad (1)
\]

To eliminate the anomalous FD, we create a new element type \( r \) 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 \( r \) 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 6.

![Figure 6: creating new element types](image)

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

13
Formally, if \( \tau, \tau_1, \ldots, \tau_n \) are element types that are not in \( E \), the new XML Schema, denoted by \( \chi[p.@l := q.r \left[ \tau_1.@l_1, \ldots, \tau_n.@l_n, @l \right] ] \), is \((E', A', M', P', r, \Sigma)\), where \( E' = E \cup \{ \tau, \tau_1, \ldots, \tau_n \} \) and

1. if \( M(\text{last}(q)) \) is a regular expression \( s \) and \( \tau \), that is \((s, \tau)\). Furthermore, \( M'(\tau) \) is defined as the concatenation of \( \tau_1^* \), \( \ldots, \tau_n^* \), \( M(\tau_i) = e \), for each \( i \in [1, n] \), and \( M'(\tau) = M(\tau) \), for each \( \tau' \in E - \{ \text{last}(q) \} \).
2. \( P(\tau) = \{ @l \} \), \( P(\tau_i) = \{ @l_i \} \), for each \( i \in [1, n] \), \( P(\text{last}(p)) = P(\text{last}(p)) - \{ @l \} \) and \( P'(\tau) = P(\tau) \) for each \( \tau' \in E - \{ \text{last}(p) \} \).

After transforming \( X \) into a new XML Schema \( X' = \chi[p.@l := q.r \left[ \tau_1.@l_1, \ldots, \tau_n.@l_n, @l \right] ] \), a new set of functional dependencies is generated. Formally, \( F[p.@l := q.r \left[ \tau_1.@l_1, \ldots, \tau_n.@l_n, @l \right] ] \) 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 \( S_1 \cup S_2 \subseteq \text{paths}(X) \).
2. Each FD over \( q, p_i . @l_i \) and \( p.@l \) is transferred to \( \tau \) and its children. That is, if \( S_1 \cup S_2 \subseteq \{ q, p_1, \ldots, p_n . @l_1, \ldots, p_n . @l_n, p.@l \} \) 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 \( \{ q. \tau . \tau, q. \tau . \tau_i . @l_i \} \rightarrow q. \tau . \tau_i \) for \( i \in [1, n] \).

### 5.1.2 Moving Attributes

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 \( q \rightarrow p.@l \), where \( q \in \text{EPaths}(X) \). 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 7.

![Figure 7: Moving Attributes](image)

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

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

The algorithm applies the two transformations introduced in the previous sections until the schema is in X-BCNF, as shown in Figure 8.

The algorithm shows in Figure 8, 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:

**Figure 8: X-BCNF decomposition algorithm.**

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

6. 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


Authors:

Quatibah Malluhi, Professor
Ahmad M. Hasna, Associate Professor
Jihad M. Al Ja’am, Associate Professor
Hosam, F. El-Sofany, Lecturer
Department of Engineering and Computer Science, College of Engineering, Qatar University, Doha – Qatar.

Fayed, F. M. Ghaleb, Associate Professor
Sameh S. Daoud, Associate Professor
Department of Mathematics, Faculty of Science, Ain Shams University, Cairo – EGYPT

Samir, Abou El-Seoud, Professor
Computer Science Department, Princess Sumaya University for Technology, Amman – Jordan