NF-SS: A Normal Form for Semistructured Schema

Xiaoying Wu Tok Wang Ling Sin Yeung Lee Mong Li Lee
School of Computing, National University of Singapore, Singapore
{wuxiaoy1, lingtw, jlee, leeml }@comp.nus.edu.sg

Gillian Dobbie
Dept of Computer Science, University of Auckland, New Zealand
gill@cs.auckland.ac.nz

Abstract. Semistructured data is becoming increasingly important for web applications with the development of XML and related technologies. Designing a “good” semistructured database is crucial to prevent data redundancy, inconsistency and undesirable updating anomalies. However, unlike relational databases, there is no normalization theory to facilitate the design of good semistructured databases. In this paper, we introduce the notion of a semistructured schema and identify the various anomalies that may occur in such a schema. A Normal Form for Semistructured Schemata, NF-SS, is proposed. A semistructured schema in NF-SS guarantees minimal redundancy and hence no undesirable updating anomalies for the associated semistructured databases. Furthermore, a semistructured schema in NF-SS gives a more reasonable representation of real world semantics. We develop an iterative algorithm based on a set of heuristic rules to restructure a semistructured schema into a normal form. These design methods also provide insights into the normalization task for semistructured databases.

1. Introduction

Semistructured data plays a crucial role in the new Internet applications ranging from electronic commerce to web site management to digital government. The emergence of XML (eXtended Markup Language) [2] as the likely standard for representing and exchanging data on the web has confirmed the central role of semistructured data. Many information providers have published their databases on the web as semistructured data, and others are developing repositories for new applications. As with traditional databases, data redundancy and inconsistency may occur in a semistructured database if its schema is not designed properly. Thus, it is important to provide guidelines for designing “good” semistructured databases.

Unlike the relational model where normalization theory is used to decide whether a set of relations is a good design for a given database, there is no normalization theory defined for semistructured data to determine whether a semistructured database has been well-designed. This is a major problem in the field of semistructured data research.

Normal forms defined for relational databases, either 3NF, BCNF, 4NF and 5NF for the flat relational model, or nested relations like NNF [8] and NF-NR [7], are not directly applicable to semistructured data for the following reasons. First, the semistructured data model is richer and more complex than the flat relational data model. For example, XML incorporates cardinality constraints that are not found in the relational data model. Second, a semistructured data instance whose structure is embedded together with the data, is required to *conform to* its schema. Hence there is no regular structure in semistructured data instances. Third, unlike comparing values of atomic types, it is a nontrivial task to directly compare values of hierarchically structured data. The notion of value equality for semistructured data has to be defined. Fourth, dependency constraints (such as functional and multi-valued dependencies) used in traditional design approaches are not directly applicable to the semistructured data model.

In this paper, we propose NF-SS, a normal form for semistructured schema. We use XML as a data model to represent semistructured data and enrich the model with schema and integrity constraints such as dependency and key constraints. We will show that a semistructured schema in NF-SS not only reduces redundancy and undesirable updating anomalies for the complying semistructured databases, but also captures the set of semantic connections among objects and attributes that exist in the real world. We will propose a set of heuristic restructuring rules and use them in an algorithm to obtain a NF-SS schema.

The rest of paper is organized as follows. Section 2 gives a motivating example and identifies the anomalies that may occur in semistructured databases. Section 3 defines NF-SS, a normal form for semistructured schema. Section 4 presents a set of heuristic rules to restructure a semistructured schema. An algorithm that iteratively transforms a semistructured schema into NF-SS is also given. Section 5 discusses some related works and we conclude in Section 6 with directions for future work.

2. Motivating Example

In this section, we will give an example to demonstrate that without some guide to design schemas, it is easy to produce semistructured schemas that contain redundancy and unnatural representation of real word semantics.

Example 2.1 Consider the XML DTD in Figure 2.1. A graphical representation of the DTD is shown in Figure 2.2(a). A database instance that conforms to this DTD is given in Figure 2.3. Note that this database is not well designed because it contains data redundancy: the *name* and *age* of a *student* will be repeated for each additional *course* he/she takes. Similar to traditional databases, we can identify three kinds of update anomalies in a badly designed semistructured database: *insertion anomaly*, *rewriting anomaly* and *deletion anomaly* (see [9] for more details). This redundancy can be avoided if the database is designed according to the schema shown in Figure 2.2(b), where *student's* information, including name and age, is referenced rather than nested under *course*.

```
<!ELEMENT   department    (course+)
    <!ATTLIST     department
                     name            ID        #REQUIRED>
<!ELEMENT   course              (students*)>
    <!ATTLIST     course
                    cid              ID          #REQUIRED
                    title       CDATA          #IMPLIED>
<!ELEMENT   student             (grade?)>
    <!ATTLIST     student
                    sid        ID     #REQUIRED
                  name    CDATA         #REQUIRED
                  age       CDATA         #IMPLIED>
<!ELEMENT    grade          (#PCDATA)>
```

Fig. 2.1: Example of a DTD.

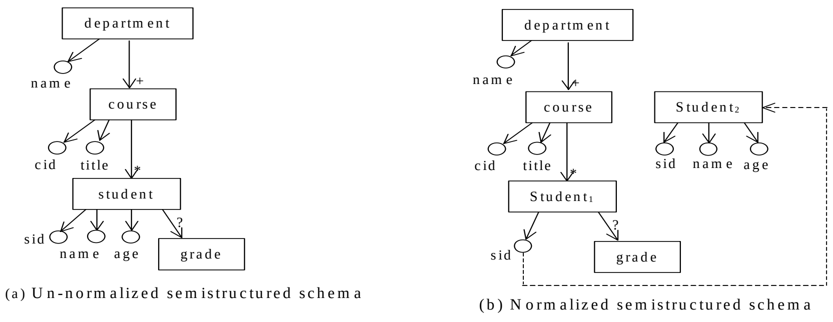

Fig. 2.2: Graphical representations for the DTD in Figure 2.1.

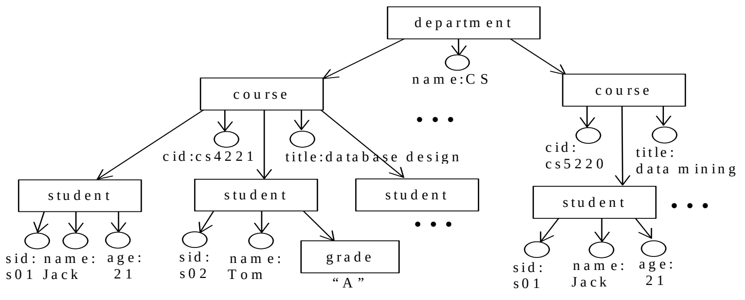

Fig. 2.3: A semistructured database instance that conforms to the schema in Figure 2.2(a).

3. A Normal Form for Semistructured Schema (NF-SS)

In this section we will give the formal definition of a semistructured schema as well as the concept of data tree that conforms to its schema. We will also introduce the concept of Extended Functional Dependency (EFD) and key constraints defined over semistructured schema. Finally, we will present the definition of NF-SS.

3.1 Semistructured Schema (*D*) and Data Tree (*T*)

Definition 3.1: A *semistructured schema D* = (E, A, B, P, R, r) where

- E is a finite set of *object types* in *D*.
- A is a finite set of *attributes*, disjoint from E.
- B is a set of basic domain type like *string, integer, Boolean* etc.
- P is a function from E to *object type definition*, which is an expression $o_1^{w_1},...,o_k^{w_k}$, where O_i is either a distinct object type in E or a basic domain type in B. Each W_i is a symbol in {*, +, ? ,1} called *multiplicity*.
- R is a function from E to the power set of A. If a $\in$R (o), then a is defined for o.
- r $\in$ E and is called the *object type of the root*.

We denote an object type *o* in an schema *D* with attributes $a_1, \ldots, a_m$ and sequence of sub-object type $o_1,\ldots o_n$ by $o(a_1,\ldots,a_m;\ o_1,\ldots o_n)$. *D* can be represented graphically as a tree with labeled rectangles and circles denoting object types and attributes respectively. Any multiplicity is indicated on the edges between object types.

Example 3.1 The schema in Figure 2.2(a) can be represented as *D*=(E, B, A, P, R, r) where E= {department, course, student, grade} A = {cid, title, sid, name, age}

P(department) = course$^+$	R(department) = {name}
P (course) = student*	R(course) = {cid, title}
P (student) = grade	R(student) = {sid, name, age }
P (grade) = *string*	r = department

A semistructured database instance such as an XML document is usually modeled as a labeled tree. We define such a data tree as an image of a given semistructured schema as follows.

Definition 3.2: A *data tree T* w.r.t. a semistructured schema *D* = (E, A, B, P, R, r) is defined to be a tree *T* = (V, *lab*, *obj*, *att*, *val*, *root*) where

- V is a finite set of nodes.
- *lab* is a labeling function that mapping each node in *V* to E$\cup$A$\cup$B.
- *obj* is a partial function from V to a sequence of nodes in V such that for any v$\in$V, *obj*(v) is defined iff *lab*(v)=o, and o$\in$E; moreover, if *obj*(v)=<v_1, …, v_n>, then <*lab*(v_1), …, *lab*(v_n)> must be in $<o_1^{w_1},...,o_n^{w_n}>$ defined by P(o); and for $o_i^{w_i}$ in P(o), the number of children of v that is labeled O_i as is restricted as follows:
  w_i=1: exactly one child is labeled O_i; w_i =?: at most one child is labeled O_i;
  w_i =+: at least one child is labeled O_i; w_i =*: no restriction;
- *att* is a partial function from V$\times$A to V such that for any v$\in$V and a$\in$A, *att*(v,a)=v’ is defined iff *lab*(v)=o, *lab* (v’)=a, o$\in$E , v’$\in$V and a $\in$R(o).
- *val* is a partial function from V to atomic values such that for any node v$\in$V, *val*(v) is an atomic value iff *lab*(v)=s, s$\in$B or s$\in$A.
- *root* is a distinguished node in V, *lab*(*root*)=r , and is called the root of *T*.

Example 3.2 The data tree in Figure 2.3 is an instance of the schema in Figure 2.2(a). The labeled rectangles and circles denote object nodes and attribute nodes respectively. In the department object node, *obj* defines its sub-objects such as course "cs4221" and teacher "t1234" etc., while *att* defines its attribute nodes labeled with name, which is assigned value "cs" by *val*.

For any node n in a tree, either a schema tree *D* or a data tree *T*, there exists a unique path starting from the root to *n*, denoted as $Path_D$(n) or $Path_T$(n) accordingly. Their formal definitions are provided in [10], and their notation makes use of XPath expression [4].

Since data tree T is an image of D, given a node n∈E∪A in D, there maybe many nodes in T whose path in T is equal to n's path in D, we call such set of nodes as target set of n in *T*. Formally, it is defined as follows:

Definition 3.5: Let *T*=(V, *lab*, *obj*, *att*, *val*, *root*) be a data tree of schema *D*= (E, A, B, P, R, r). n∈E∪A. The *target set* of n in *T*, denoted as *T*[n], is {v: v∈V, Path_T(v)= Path_D(n)}

Example 3.3: Refer to the schema in Figure 2.2(a), the path for object type *student* is /*department/course/student*. For its data tree *T* in Figure 2.3, the path for student "*s01*" is */department/course/student*; the target set *T*[*student*] includes nodes of *students* with *sid* "*s01*" and "*s02*".

3.2 Extended Functional Dependency (EFD) and Key Constraints

Extended Functional Dependency (EFD)

Before we give the definition for EFD, we first define some terminology that is fundamental to the semantics of EFD.

First, we provide the notion of equality of two nodes in data trees. Intuitively, two nodes n_1, n_2 are *value equal* denoted by $n_1 =_v n_2$, if they have the same label and either they have the same atomic value (when they are value or attribute nodes) or their children are pair-wise *value equal* (when they are object nodes).

Example 3.4 In Figure 2.3, the leftmost *student* node (*sid* is "*s01*") and the rightmost *student* node (*sid* is "*s01"*) are value equal because they have the same label tag (*student*) and all their children <*sid*, *name*, *age*> are pairwise value equal.

In relational databases, when two tuples agree on a set of attributes X, this implies that their projections on X are equal. Similarly, two data trees T_1 and T_2 that agree on an object type or attribute *X* implies that there exists nodes in their target sets of *X* in T_1 and T_2 respectively satisfying value equality.

Definition 3.6: Let T_1 and T_2 be two data trees that are images of schema *D* = (E, A, B, P, R, r). Let *X*∈E ∪ A. T_1 and T_2 *agree on X*, denoted as $T_1 =_X T_2$ iff the following condition is hold: $\exists t_1 \in T_1[X], t_2 \in T_2[X]$, such that $(t_1 =_v t_2)$

Note that we do not require that the above two target sets to satisfy set equality, since it is possible that there are no such nodes in a data tree.

Definition 3.7: Let D = (E, A, B, P, R, r) be a semistructured schema, let X ⊆ E∪A and Y ∈ E∪A. Y is extended functionally dependent on X, is denoted as X⇒Y. Let S denotes a set of data trees that are images of D, S *satisfies* $X \Rightarrow Y$, iff for any data trees T_1, T_2 in S, if they agree on every component in X, then they will agree on Y. That is,

$$\forall T_1, T_2 \in S((\forall x \in X, T_1 =_x T_2) \mapsto T_1 =_y T_2).$$ [1]

In the above definition, we write the EFD as $X \rightarrow Y$ if Y is an attribute of an object type or a single valued object type. If there exists an $X' \subset X$ such that $X' \Rightarrow Y$, then the EFD $X \Rightarrow Y$ is called *partial* EFD; otherwise we say that $X \Rightarrow Y$ is a *full* EFD. If $Y \subseteq X$, then $X \Rightarrow Y$ is a *trivial* EFD. $X \Rightarrow Y$ is said to be *coherent* iff /X/Y is a path in D; otherwise it is called an incoherent EFD. If there exists $Z \in E \cup A$, such that $X \Rightarrow Y$ and $Y \Rightarrow Z$ and $Y \nRightarrow X$, then Z is transitively extended functionally dependent on X via Z. From the definition of partial EFD and transitive EFD as well as inference rules for EFD, it is easy to conclude that partial EFD is a special case of transitive EFD. We use the following notation for an EFD:

$$O_1[@X_1], \ldots, O_i[@X_i], \ldots, O_{n-1}[@X_{n-1}] \Rightarrow O_n[@X_n]$$

where O_i is an object type in D, X_i is a set of O_i's attributes that participate in the dependency. Note that the notation makes use of Xpath[4] expressions.

Example 3.5 Consider the schema in Figure 2.2(a). We have the following two EFDs: (1)course[@cid],student[@sid]⇒student[@name] (2)student[@sid]⇒student[@name] where (1) is a partial EFD while (2) is a full EFD since a *student*'s name is fully determined by *sid*.

An incoherent EFD introduces a path that is not expressed in the schema's structure. We say that the existence of incoherent EFD leads to *path anomaly* in schema that indicates an unintuitive grouping of objects and attributes in schema's structure. In such cases, the schema does not adequately reflect the real world semantic relationships and will lead to data redundancy and data retrieval overheads.

Example 3.6 Consider **t**he schema D shown in Figure 3.1(a) that intends to be a schedule for teachers lecturing courses. Assume the specified EFD is teacher[@tid], time [@day, @hour]→subject[@cid]. It is an incoherent EFD, since /*teacher*/*time*/*subject* is not a path in the schema. Therefore, there exists a path anomaly. Such anomaly can be avoided if we promote *time* to be child of *teacher*, and move *ClassRoom* and *subject* under *time*, as shown in Figure 3.1(b).

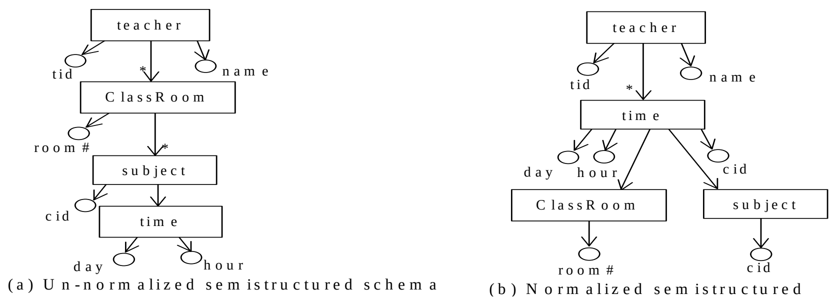

Fig. 3.1: Schema for Example 3.6 illustrating path anomaly

[1] The inference rules for EFD is given in [10]

Example 3.7 In Figure 2.2(a), *age* is transitively dependent on *course* via *student* given the constraints: (1) course[@cid]$\Rightarrow$student[@sid]
(2) student[@sid]$\rightarrow$student[@age] and
student[@sid] $\not\Rightarrow$course[@cid]

Theorem 3.1: Let D = (E, A, B, P, R, r) be a semistructured schema, X, Y, Z $\subseteq$ E $\cup$A. If Z is transitively dependent on X via Y, then there exists a data tree of D in which a *rewriting anomaly* arises when the values of Z are updated.

We will illustrate the above theorem with the following example. A detailed proof is given in [10].

Example 3.8 Refer to Figure 2.3. As aforementioned, *age* is transitively dependent on *course* via *student*. The age information for student with *sid* equal "*s01*" is repeated for two courses in which he is enrolled. If we update his age information under the course *"cs4221"*, then it can cause inconsistency unless the update is also propagated to his age information under the course *"cs5220"*.

Key Constraints

Key plays an important part in database design [1,9]. Unlike previous proposal that define key constraints for semistructured data in the absence of schema [3], we want to define the notion of key based on semistructured schema and EFD. This allows us to derive key constraints more easily.

Definition 3.8: Let $O(a_1,\ldots,a_m\,;o_1,\ldots,o_n)$ be a complex object type in semistructured schema, and K is a non-empty subset of $\{a_1,\ldots,a_m,\ o_p,\ldots,o_q\}$, where $\{o_p,\ldots,o_q\} \subseteq \{o_1,\ldots,o_n\}$, and are single-valued atomic sub-object types. The *key* of O is defined as:

Case 1: O is the root of D. K is a *candidate key* of O iff $K\rightarrow O$ and there is no proper subset K_1 of K, such that $K_1 \rightarrow O$.

Case 2: O is at some level in D. Let $Path_D(O)$=/O_0/…/O_{v-1}/O, v>0. Let $Pk \subseteq K_{ov-1}$, where K_{ov-1} is the key of O's parent O_{v-1}. $K_o = Pk \cup K$. K_o is a *candidate key* of O iff ($Pk, K\rightarrow O$) and there is no proper subset S of $\{P_k, K\}$ such that $S \rightarrow O$.

We use the following notation for the key of an object type O in a semistructured schema D: K_o = O_1[@X_1]/…/O_i[@X_i]/…/O_n[@X_n]/O[@X], where O_i is an object type in D, X_i is a set of O_i's components, and O_1/…/O, is a path in D (n>0). If n equals one, then K_o is called an *absolute key*. Otherwise it is called a *relative key*.

Example 3.9 Consider a semistructured schema describing books, which has path /book[@isbn]/chapter[@number]/chapter[@number]. We have

- K_{book}= *book*[@*isbn*]. K_{book} is an absolute key indicating that *isbn* uniquely identifies a book.
- $K_{chapter}$ = *book*[@*isbn*]/*chapter*[@*number*]. $K_{chapter}$ is a relative key indicating that a chapter number uniquely identifies a chapter contained in a book.

- $K_{section}$= *book*[@*isbn*]/*chapter*[@*number*]/*section*[@*number*]. $K_{section}$ is a relative key indicating that a section number uniquely identifies a section within a chapter of a book.

[10] defines a foreign key constraint for semistructured data which is similar to IDREF in XML. Consider the schema in Figure 2.2(b). *sid* in $student_1$ is a foreign key and refers to $student_2$ (referenced object type). This reference semantics is represented graphically as a dashed edge (*reference edge*) from the foreign key to the referenced object type.

Definition 3.9: Let *D* be a semistructured schema and *O* be its root object type. The set of *basic dependencies* of *D*, denoted as BD(*D*), is defined as follows:

- Let X, Y be children of *O*, non-trivial extended functional dependencies of the form X⇒Y where X is a key of *O* or Y is part of a key of *O*, are in BD(*D*).
- For each sub-object type O_i of *O*, extended functional dependency $K_O \Rightarrow O_i$ is in BD (*D*), where K_O is *O*'s key.
- Let O_1 be a complex sub-object type of *O* and D_1 be a schema tree that is rooted at O_1 and add K_O as attribute(s) of O_1, then BD(D_1) $\subseteq$ BD(*D*).
- No other non-trivial dependencies that are not generated from above is in BD(*D*)

3.3 NF-SS: Normal Form for Semistructured Schemata

We now give the definition of NF-SS.

Definition 3.9 Let *D* be a semistructured schema and *O* be its root object type. *D* is in *Normal Form for Semistructured Schemata* (*NF-SS*), iff

1. *O* has at least one key.
2. For any non-trivial EFD of the form X⇒Y satisfied by *O*, where X and Y are attributes or atomic sub-object types of *O*, then either X is a key or Y is part of the key of *O*.
3. For any complex sub-object type O_1 of *O*
   (a) If adding K_O to O_1 as its components with other remains, a schema tree D_1 rooted at O_1 will be in NF-SS.
   (b) $K_O \cap K_{O1}=\phi$ or $K_O \subset K_{O1}$, where K_O and K_{O1} are *O* and O_1's key respectively.
   (c) O_1 is not transitively dependent on K_O
4. Any non-trivial EFD in *D* can be derived from BD(*D*) by using the inference rules for EFDs.

4. Designing Semistructured Schema in NF-SS

We adopt the restructuring approach to design semistructured schema. We propose a set of restructuring rules to transform a semistructured schema into NF-SS. This restructuring involves the decomposition of object types, creation of new object types and regrouping of components in a semistructured schema. The objective is to remove transitive or partial EFD and incoherent EFD. This is accomplished by identifying violations of the conditions of NF-SS from the given dependency and key constraints.

4.1 Normalization Rules

Rule 1. (**Remove Transitive Dependencies by Decomposition**) Given an object type *O* in a semistructured schema *D*. Suppose there exists some non-prime component(s) Y of *O* that is transitively dependent on some key K_O of *O*, i.e., $K_O \Rightarrow X$, $X \Rightarrow Y$ and $X \not\Rightarrow K_O$, and $X \cap K_O = \phi$. Then D can be restructured as follows:

1. Duplicate X to form a new node(s) Z.
2. Move Y and all the descendants of Y and their corresponding edges under Z.
3. Make X as foreign key of *O*, and add a reference edge from the original node X to Z.

Element duplication is actually relation decomposition or splitting when the element is an object type. Rule 1 can be used to remove undesirable transitive dependency. Note that splitting may happen in many ways and choosing a correct way to decompose is nontrivial since certain splitting will cause the loss of EFDs.

Example 4.1 Consider the schema in Figure 2.2(a) with the following dependency constraints:

(1)department[@name]$\Rightarrow$course[@cid] (2) course[@cid]$\rightarrow$department
(3)course[@cid]$\rightarrow$course[@title] (4)course[@cid]$\Rightarrow$student[@sid]
(5)course[@cid],student[@sid]$\rightarrow$grade (6)student[@sid]$\rightarrow$student[@name, @age]

From the EFDs (4) and (6), *D* is not in NF-SS since *name* and *age* are transitively dependent on *course* via *student.* Furthermore, student[@sid] $\not\Rightarrow$ course. Since *course*[*@cid*]$\cap$*student*[*@sid*]=ϕ, we can use Rule 1 to decompose *student* into two object types: student1(sid; grade) and student2(sid, name, age). The former remains as a sub-object type of course while the latter becomes the root of a new schema tree. A reference edge is created from student_1's sid (foreign key) to student_2. The schema is now in NF-SS, as shown in Figure 2.2(b).

Rule 2. (Remove Path Anomaly by Path Splitting) Given a semistructured schema D. Suppose there exists an incoherent EFD: $O_1[@X_1],\ldots,O_n[@X_n] \mapsto Y$, where $\mapsto$ denotes either $\rightarrow$or $\Rightarrow$, Y is either an object type or an attribute, and there exists a path P that contains $\{O_1,\ldots,O_n,Y\}$. Path P can be split into two sub-paths P_1 and P_2, where P_1 only contains $\{O_1,\ldots,O_n\}$ and Y, while P_2 contains $\{O_1,\ldots,O_n\}$ and (P-Y). If we have $\mapsto$ is $\rightarrow$, then the cardinality of Y (when *Y* is an object type) is assumed to be "?" after restructuring; Otherwise, if we have $\mapsto$ is $\Rightarrow$, then the cardinality of Y is "*".

Rule 2 can be used to remove path anomalies and partial dependencies. This in turn helps to avoid *over-nesting*. Intuitively, components (whether they be attributes or object types) should be kept as close to the *owner* object type as possible. This is achieved though path splitting. The *promotion* of a component may happen more than once, each time moving the component closer to its rightful owner.

Example 4.2 Consider **t**he schema *D* shown in Figure 3.1(a). Assume the set of specified EFDs follows:

(1) teacher[@tid],time$\rightarrow$ClassRoom (2)teacher[@tid], time$\rightarrow$subject

D is not in NF-SS since (1) is an incoherent dependencies. Applying Rule 2, *D* is normalized by splitting path */teacher/ classroom/subject/time* into two sub-paths: */teacher/ time/ClassRoom* and */teacher/time/subject*, as shown in Figure 3.1(b).

Rule 3. (**Removing Partial Dependency by Creating New Object type**) Given an object type *O* in a semistructured schema D. Let X be a set of prime attributes of *O,* and Y be the set of *O's* attributes. Let O_1 be a sub-object type of *O*. If (K_O -X) $\Rightarrow$ O_1 and no proper superset of X satisfy this property, then D can be restructured as follows:

1.($K_O \cap$Y –X) becomes the only attribute(s) of *O* while O_1 remains to be its sub-object type.
2.Create a new object type O_2 that is a direct component of *O.*
3.Move rest of the components of *O* and all their descendants and corresponding edges under O_2.

Example 4.3 Consider the schema *D* in Figure 4.1(a). Suppose *O* satisfies the *EFDs* {*O*[@A,@B]$\rightarrow$D, *O*[@A,@B]$\Rightarrow O_2$, *O*[@A]$\Rightarrow$ O_1, *O*[@A] $\rightarrow$E} and the key of *O* is {A,B}.

D is not in NF-SS because *O*[@A,@B]$\Rightarrow O_1$ is not a full dependency. Applying Rule 3, a new object O_3 is created which has O_2 and {B, D, E} as its children (*O* is renamed as O'). The restructured schema given in Figure 4.1(b) is still not in NF-SS since there exists an incoherent dependency O'[@A]$\rightarrow$E. We apply Rule 2 to promote E as a child of O'' (O' is renamed as O''). The schema obtained in Figure 4.1(c) is now in NF-SS.

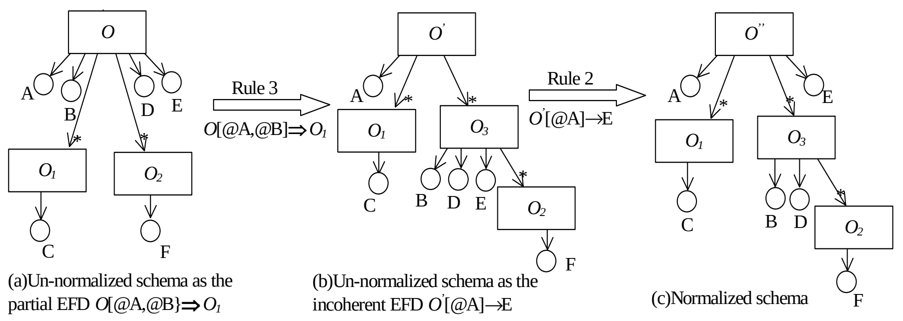
O
A
B
D
E
O1
O2
C
F
Rule 3
O[@A,@B]⇒O1
O'
O1
O3
B
D
E
O2
Rule 2
O'[@A]→E
O''
O3
(a)Un-normalized schema as the partial EFD O[@A,@B}⇒O1
(b)Un-normalized schema as the incoherent EFD O'[@A]→E
(c)Normalized schema

Fig. 4.1: Using Rules 2 and 3 to restructure a schema.

Rule 4. (**Restructure to satisfy Condition 3(b) of NF-SS Definition**) Given an object type *O* in a semistructured schema *D*. Let X be a set of *O*'s attributes and single-valued atomic sub-object types, O_1 be a complex sub-object type of *O*. O_1 has relative key K_{O1}, but $K_O \not\subset K_{O1}$ and $K_{O1} \not\Rightarrow K_O$. Let Y be $K_O \cap K_{O1} \cap$ X, and $Y \neq \phi$. *D* can be restructured as follows:

1. O_1 remains to be a sub-object type of *O*.
2. Make Y as components of *O*.
3. Create a new object type O_2 to be a child of *O* and the rest components of *O* (excluding Y) become children of O_2.

Example 4.4 Consider the semistructured schema *D* in Figure 4.2(a). Suppose *O* satisfies the EFD: (1) O[@K, @A]$\Rightarrow O_1$ (2) O[@K, @B]$\Rightarrow O_2$ and the key of *O* K_O is {K, A, B}.

D is not in NF-SS since O_1 and O_2 are partially dependent on the key of *O*. We use Rule 3 to create a new object type O_3, rename *O* as $O^{'}$ and make it a child of $O^{'}$; After that, we move B and O_2 and all their descendants and corresponding edges under O_3. Figure 4.2(b) shows the schema obtained. This is still not in NF-SS because Condition 3(b) in the NF-SS definition remains violated: K_O=$O^{'}$[@K,@A],while K_{O3}= $O^{'}$[@K]/O_3[@B]. In addition, $K_{O3} \rightarrow K_O$ cannot be derived. Applying Rule 4, O_3 remains to be a sub-object type of $O^{''}$, and K become attribute of $O^{''}$ ($O^{'}$ is renamed as $O^{''}$). We create a new object type O_4 as a child of $O^{''}$ and move the rest components of $O^{''}$ and their corresponding edges under O_4. The schema shown in Figure 4.2(c) is now in NF-SS.

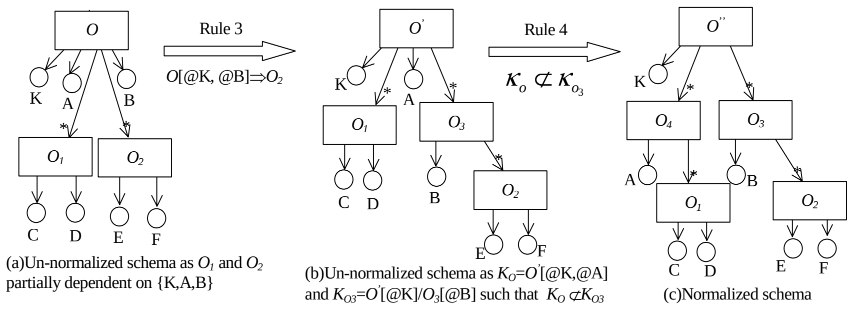
O
K
A
B
O1
O2
C
D
E
F
(a)Un-normalized schema as O1 and O2 partially dependent on {K,A,B}
Rule 3
O[@K, @B]⇒O2
O'
O3
(b)Un-normalized schema as KO=O'[@K,@A] and KO3=O'[@K]/O3[@B] such that KO ⊄KO3
Rule 4
Ko ⊄ Ko3
O''
O4
(c)Normalized schema

Fig. 4.2: Using Rules 3 and 4 to restructure a schema.

4.2 Restructuring Algorithm

In this section, we present an algorithm that uses the normalization rules presented in Section 4.1 to iteratively restructure a semistructured schema into NF-SS. The algorithm takes as input a set of semistructured schema and a set of dependency constraints for these schemas. It returns as output a set of semistructured schema in NF-SS.

Algorithm 4.1: Restructuring Algorithm

```
Input: A set S that contains semistructured schemas, and a set of EFDs for S.
Output:   A set of semistructured schemas that in NF-SS.
Begin
    1.  for each semistructured schema D in S do
            if D is not in NF-SS then repeat until no further change:
            (1) if $\exists$ transitive EFD: $K_O \Rightarrow X$, $X \Rightarrow Y$ and $X \not\Rightarrow K_O$ for an object type O in D,
                Case 1: $X \cap K_O = \phi$.
                        Apply Rule 1 to remove the transitive EFD.
                Case 2: $X \subset K_O$.
                        Apply Rule 3 to remove the transitive EFD.
                Case 3: $X \cap K_O \neq \phi$.
                        Apply Rule 4 to remove the transitive EFD.
              (2) if there exists incoherent EFD then apply Rule 2 to remove it.
        2. output S.
End
```

In the normalization process, object types may be created and components of the schema may be regrouped. Two problems are involved there. One is the cardinality of a new sub-object type. We assume "*" on the new object type at this design stage, and let designers/users elaborate later. The other is naming of the new object types. We believe it is generally preferable to have designers/users specify alternate names, which indicate the role played by the object type in the context of the application.

4.3 Discussion

We have presented a technique for restructuring semistructured schema to obtain NF-SS. We would like to highlight two pertinent issues for this restructuring approach. The first issue is the completeness of the restructuring rules. That is, given a schema, is it always possible to restructure it into a set of semistructured schema in NF-SS using heuristic rules such as Rules 1 to 4 ? This is a difficult question that also arises when the decomposition approach is used in relational databases. It is not always possible to get all the EFDs satisfied by a semistructured schema, that is, covering is not guaranteed. Furthermore, it is not always possible to preserve dependencies during transformation, that is, dependency preservation is not guaranteed, which is a problem that also happens to the decomposition method taken in relational database design. A formal investigation of the problem is beyond the scope of this paper. Nevertheless, we would like to point out that losing some EFD could actually prevent infinite loop for Algorithm 4.1 in some situations.

Consider the following two EFD (1) $A, B \Rightarrow C$ (2) $A, C \Rightarrow D$ for a schema *D* that has a path /A/B/C/D. There is a "conflict" between (1) and (2) in the sense that only one of them is expressible in *D*. Hence, applying these rules to an unnormalized schema results in infinite schema transformations and there may exist conflicts among the specified EFD constraints.

The second issue is the uniqueness of the solution. That is, does the process of restructuring give a unique solution? The answer is no. In the normalization of relational schema, it is well known that decomposition does not guarantee unique results as it depends on the order in which the dependencies are examined. Although the restructuring approach does not necessarily give unique results and guarantee dependency preservation, it does give practical heuristics and provides insights into the normalization task for semistructured databases.

5. Related Works

To the best of our knowledge, only [6] and [5] provide works that parallel our research efforts here. [6] defines a schema called S3-Graph. S3-Graph makes no distinction between element node and attribute node and does not specify cardinality on the schema. Therefore S3-Graph doesn't lend itself to XML definition. To identify redundancy in S3-Graph, [6] defines a dependency constraint called SS-Dependency. A S3-Graph is in S3-NF if there is no transitive SS-dependency. This limits the types of redundancies that can be resolved by S3-NF. S3-NF deals with SS-dependency constraints and does not handle key constraints, an essential feature in database design. Furthermore, S3-NF may not remove anomalies such as like partial dependency and path anomaly. In contrast, NF-SS is designed to handle more general situations and therefore, subsumes and extends that of [6].

[5] defines a normal form called XNF (XML Normal Form) is defined. The work in [5] focuses on how to translate a schema, that is represented in conceptual-model hypergraphs, to a scheme-tree forest in XNF. A scheme-tree forest *F* is in XNF if each scheme tree in *F* has no potential redundancy with respect to a specified set of (functional and multivalued) constraints *C* and *F* has as few, or fewer, scheme trees as any other schemes-tree forest corresponding to *M* in which each scheme tree has no potential redundancy with respect to *C*. CM hypergraph has no hierarchical structures, no key concepts; additionally, it has no concept of attributes resulting too many objects in a schema. Although an XNF-compliant DTD can ensure complying XML documents have as few hierarchies as possible, the presented algorithms for generating XNF scheme-tree forest suffers from efficiency. A large set of scheme-tree forests that in XNF is generated and this requires the user to select the best that satisfies their application requirements.

[7] studies the normalization for nested relational data model, and proposes a normal form called NF-NR(*Normal Form for Nested Relations*). Our NF-SS definition and normalization process is similar to that of [7] in concept, but is different in essence, which we have mentioned in the first section.

6. Conclusion

In this paper, we have shown the importance of designing good semi-structured databases. We defined a semistructured schema for semistructured databases, and incorporated it with integrity constraints such as dependency and key constraints. We identified various anomalies, including rewriting anomaly, insertion anomaly, deletion anomaly and path anomaly, that may arise when a semistructured database is not designed properly and contains redundancies. We proposed NF-SS, a normal form for semistructured schema. A semistructured schema in NF-SS does not have redundancy and hence no undesirable updating anomalies for the conforming semistructured databases. In addition, a semistructured schema in NF-SS also gives a more reasonable representation of real world semantics. We have presented a set of heuristic restructuring rules and developed an algorithm for iteratively restructuring a semistructured schema into NF-SS.

Future directions for research include investigating additional heuristic restructuring rules as well as extending existing rules to deal with various anomalies that may exist in semistructured schemata. We also intend to improve the restructuring Algorithm 4.1 by developing a conflict-detecting framework to check for the existence of conflicts within the specified dependency constraints for a schema.

References

1. S. Abiteboul, R. Hull and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995
2. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. 2nd Edition, Oct. 2000. http://www.w3.org/TR/REC-xml.
3. P. Buneman, S. Davidson, W. Fan, C. Hara and W. Tan. Keys for XML. *Proceedings of the 10th International World Wide Web Conference*, 2001.
4. J. Clark and S. DeRose. XML Path Language (XPath). W3C Working Darft, November 1999. http://www.w3.org/TR/xpath.
5. D.W.Embley and W.Y.Mok. Developing XML Documents with Guaranteed "Good" Properties. *Proceedings of the 20th International Conference on Conceptual Modeling (ER)*, 2001.
6. S. Y. Lee, M. L. Lee, T. W. Ling and L. A.. Kalinichenko. Designing Good Semi-structured Databases. *Proceedings of the 18th International Conference on Conceptual Modeling (ER)*, 1999.
7. T. W. Ling and L. L. Yan. NF-NR: A Practical Normal Form for Nested Relations. Journal of Systems Integration. Vol4, 1994, pp309-340
8. Z. M. Ozsoyoglu and L. Y. Yuan. A New Normal Form for Nested Relations. ACM Transaction on Database Systems. 12(1), (1987).
9. R. Ramakrishman and J.Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.
10. Xiaoying Wu. Designing Good Semistructured Databases. Master Thesis, School of Computing, National University of Singapore, 2002.