Dealing with redundancies and dependencies in normalization of XML data

Tadeusz Pankowski
Institute of Control and Information Engineering
Poznań University of Technology
Pl. M.S.-Curie 5, 60-965 Poznań, Poland
Email: tadeusz.pankowski@put.poznan.pl

Tomasz Piłka
Faculty of Mathematics and Computer Science
Adam Mickiewicz University
ul. Umultowska 87, 61-614 Poznań, Poland
Email: tomasz pilka@amu.edu.pl

Abstract—In this paper we discuss the problem of redundancies and data dependencies in XML data while an XML schema is to be normalized. Normalization is one of the main tasks in relational database design, where 3NF or BCNF, is to be reached. However, neither of them is ideal: 3NF preserves dependencies but may not always eliminate redundancies, BCNF on the contrary—always eliminates redundancies but may not preserve constraints. We discuss the possibility of achieving both redundancy-free and dependency preserving form of XML schema. We show how the XML normal form can be obtained for a class of XML schemas and a class of XML functional dependencies.

I. INTRODUCTION

As XML becomes popular as the standard data model for storing and interchanging data on the Web and more companies adopt XML as the primary data model for storing information, XML schema design has become an increasingly important issue. Central objectives of good schema design is to avoid data redundancies and to preserve dependencies enforced by the application domain. Existence of redundancy can lead not only to a higher data storage cost but also to increased costs for data transfer and data manipulation. It can also lead to update anomalies.

One strategy to avoid data redundancies is to design redundancy-free schema. One can start from an intuitively correct XML schema and a given set of functional dependencies reflecting some rules existing in application domain. Then the schema is normalized, i.e. restructured, in such a way that the newly obtained schema has no redundancy, preserves all data (is a lossless decomposition) and preserves all dependencies. In general, obtaining all of these three objectives is not always possible, as was shown for relational schemas [1]. However, in the case of XML schema, especially thanks to its hierarchical structure, this goal can be more often achieved [2].

The normalization process for relational schemas was proposed in the early 70s by Codd [3], [4]. Then a number of different normal forms was proposed, where the most important of them such as 2NF, 3NF [3], BCNF [4], and 4NF [5] are discussed today in every database textbook [1], [6], [7]. These normal forms together with normalization algorithms, aim to deal with the design of relational database taking into account different types of data dependencies [8], [7]. The result of the normalization should be a well-designed database [9]. The process of normalization of an XML schema is similar: we have to choose an appropriate XML schema for a given DTD and a set of data dependencies.

Recently, research on normalization of XML data was reported in papers by Arenas and Libkin [10], [11], Kolahi and Libkin [2], [12], [13], Yu and Jagadish [14].

In this paper we discuss a method for normalizing a class of XML schemas into an XML normal form that is redundancy-free and preserves all XML functional dependencies. To this order we apply the theory proposed in [12] and [11]. The novelty of the paper is the following:

• We use a new language (preliminarily proposed in [15]) for expressing schemas in a form of tree-pattern formulas, and for specifying XML functional dependencies.

• We show how the proposed formalism can be used for normalizing XML schemas into normal form similar to that of BCNF with eliminating redundancies but preserving all functional dependencies.

The structure of the paper is the following. In Section 2 we introduce an running example and motivate the research. In Section 3 a relational form of the running example is considered and some problems with its normalization are discussed. Basic notations relevant to the discussed issue from the XML perspective, are introduced in Section 4. We define a notation for defining tree-pattern formulas and for specifying data dependencies: XML functional dependencies and keys. Next, in Section 5, we discussed different normalization alternatives—we show advantages and drawbacks of some schema choices. Finally, in Section 6, an XML normal form (XNF) is defined (according to [11]) and we show how this form can be reached for our running example. We discuss the problem from theoretical and practical points of view. Section 7 concludes the paper.

II. MOTIVATION—REDUNDANCIES IN XML DATA

In XML data, like in relational ones, redundancies are caused by bad design of schemas. There are two kinds of design problems [11]: first of them is caused by non-key functional dependencies and is typical for relational schema design, while the other is more closely related to the hierarchical structure of XML documents.
As we mentioned above, in the case of XML schemas some redundancy problems may also occur because of bad design of hierarchical structure of XML document.

\[
S_1: \quad \text{db} \quad \text{part}^* \quad \text{pld} \quad \text{supplier}^* \quad \text{offer}^+ \\
\text{old} \quad \text{sId} \quad \text{price} \quad \text{delivTime} \quad \text{delivPlace}
\]

Fig. 1. Sample XML schema tree

\[
db \rightarrow \text{part}^* \\
\text{part} \rightarrow \text{pld}, \text{supplier}^* \\
\text{pld} \rightarrow \epsilon \\
\text{supplier} \rightarrow \text{offer}^+ \\
\text{offer} \rightarrow \text{old}, \text{sId}, \text{price}, \text{delivTime}, \text{delivPlace} \\
\text{old} \rightarrow \epsilon \\
\text{sId} \rightarrow \epsilon \\
\text{price} \rightarrow \epsilon \\
\text{delivTime} \rightarrow \epsilon \\
\text{delivPlace} \rightarrow \epsilon
\]

Fig. 2. A DTD describing the XML schema in Fig. 1. The symbol \( \epsilon \) denotes the empty string.

\[
I_1: \quad \text{db} \quad \text{part} \quad \text{pld} \quad \text{supplier} \quad \text{offer}^+ \\
\text{old} \quad \text{sId} \quad \text{price} \quad \text{delivTime} \quad \text{delivPlace} \\
\text{pld} \rightarrow \text{p1}, \text{p2} \\
\text{supplier} \rightarrow \text{supplier} \\
\text{offer} \rightarrow \text{offer}^+ \\
\text{old} \rightarrow \text{o1}, \text{o2}, \text{o3}, \text{o4} \\
\text{sId} \rightarrow \text{s1}, \text{s1}, \text{s2}, \text{s1} \\
\text{price} \rightarrow \text{x1}, \text{x2}, \text{x3}, \text{x4} \\
\text{delivTime} \rightarrow \text{t1}, \text{t2}, \text{t3}, \text{t4} \\
\text{delivPlace} \rightarrow \text{d1}, \text{d1}, \text{d2}, \text{d1}
\]

Fig. 3. Sample instance of schema \( S_1 \)

**Example 2.1:** Let us consider the XML schema tree in Fig. 1 that describes a fragment of a database for storing data about parts and suppliers offering these parts. Its DTD specification is given in Fig. 2, and an instance of this schema is depicted in Fig. 3.

Each `part` element has identifier `pld`. For each part may be zero or more suppliers offering this part. Offers are stored in `offer` elements. Each offer has: offer identifier `old`, supplier identifier `sId`, price, delivery time `delivTime`, and delivery place `delivPlace`.

We assume that the following constraints must be satisfied by any instance of this schema:

- all `offer` children of the same `supplier` must have the same values of `sId`; this is similar to relational functional dependencies, but now we refer to both the values (text value of `sId`), and to structure (children of the same `supplier`).
- `delivPlace` functionally depends on part (`pld`) and supplier (`sId`), i.e. when a supplier has two different offers for the same part (possibly with different `delivTime` and/or `price`) the `delivPlace` is the same - see offers `o1` and `o2` in Fig. 3.
- `delivPlace` functionally determines supplier (`sId`). It means that having a delivery place (`delivPlace`) we know exactly which supplier is associated to this place; although one supplier can own many delivery places. For example, in Fig. 3 `d1` is delivery place uniquely associated to the supplier `s1`.

It is easy to show that schema in Fig. 1 leads to redundancy: `sid` (an also all other data describing suppliers such as e.g.: name and address) and `delivPlace` are stored multiple times for a supplier.

Further on we will show that a special caution should be paid to such kind of dependencies as these in which participates `delivPlace`. In this case we have to do with "cyclic" dependencies, i.e. `delivPlace` depends on `pld` and `sId` (`pld, sId -> delivPlace`) and `sId` depends on `delivPlace` (`delivPlace -> sId`).

First, we will discuss difficulties caused by such "cyclic" dependencies in relational databases, and next, we will show how this problem can be solved in the case of XML data.

### III. Dealing with Redundancies and Dependencies in Relational Databases

#### A. Relational schemas and functional dependencies

In relational data model, a relational schema is understood as a pair

\[ R = (U, F) \]

where \( U \) is a finite set of attributes, and \( F \) is a set of functional dependencies over \( F \). A functional dependence (FD) as an expression of the form

\[ X \rightarrow Y \]

where \( X, Y \subseteq U \) are subsets of \( U \). If \( Y \subseteq X \), then \( X \rightarrow Y \) is a trivial FD. By \( F^+ \) we denote all dependencies which can be inferred from \( F \) by means of Armstrong’s axioms [16], [7].

A relation of type \( U \) is a finite set of tuples of type \( U \). Let \( U = \{A_1, \ldots, A_n\} \) and \( \text{dom}(A) \) be the domain of attribute \( A \in U \). Then a tuple \([A_1 : a_1, \ldots, A_n : a_n]\), where \( a_i \in \text{dom}(A_i) \), is a tuple of type \( U \).

A relation \( R \) conforms to schema \( R = (U, F) \) (is an instance of this schema) if \( R \) is of type \( U \), and all dependencies
from \( F^+ \) are satisfied by \( R \). An FD \( X \rightarrow Y \) is satisfied by \( R \), denoted \( R \models X \rightarrow Y \), if for all tuples \( r_1, r_2 \in R \) holds
\[
\pi_X(r_1) = \pi_X(r_2) \Rightarrow \pi_Y(r_1) = \pi_Y(r_2),
\]
where \( \pi_X(r) \) is the restriction (projection) of tuple \( r \) on the set \( X \) of attributes.

A key in \( R = (U,F) \) is such a minimal set \( K \) of attributes that \( K \rightarrow U \) is in \( F^+ \). Then each \( A \in K \) is a prime attribute.

B. Normalization of relational schemas

The main task in relational schema normalization is producing such a set of schemas that posses the required form, usually 3NF or BCNF. The normalization process consists in decomposition of a given input schema. The other approach consists in synthesizing 3NF from functional dependencies [8].

Ideally, a decomposition of a schema should be lossless, i.e. should preserve data and dependencies. Let \( \mathcal{R} = (U,F) \), \( U_1, U_2 \subseteq U \), and \( U_1 \cup U_2 = U \), then schemas \( \mathcal{R}_1 = (U_1,F_1) \) and \( \mathcal{R}_2 = (U_2,F_2) \) are a lossless decomposition of \( \mathcal{R} = (U,F) \), iff:
- The decomposition preserves data, i.e. for each instance \( R \) of \( \mathcal{R} \) the natural join of projections of \( R \) on \( U_1 \) and \( U_2 \) produces the relation equal to \( R \), i.e.
\[
R = \pi_{U_1}(R) \Join \pi_{U_2}(R).
\]
- The decomposition preserves dependencies, i.e.
\[
F^+ = (F_1 \cup F_2)^+,
\]
where \( F_1 = \{ X \rightarrow Y \mid X \rightarrow Y \in F \land X \subseteq U_1 \} \), and similarly for \( F_2 \).

The decomposition \( ((U_1,F_1),(U_2,F_2)) \) of \( (U,F) \) preserves data, if \( U_1 \cap U_2 \rightarrow U_1 \in F^+ \) (or, symmetrically, \( U_1 \cap U_2 \rightarrow U_2 \in F^+ \) ) [9], [7]. Then we say that the decomposition is determined by the functional dependence \( U_1 \cap U_2 \rightarrow U_1 \in F^+ \). A schema \( \mathcal{R} = (U,F) \) is in 3NF if for every FD \( X \rightarrow A \in F^+ \) holds:
- \( X \) is a superkey, i.e. a key is a part of \( X \), or
- \( A \) is prime.

The second condition says that only prime attributes may be functionally dependent on a set of attributes which is not a key. A schema is in BCNF if only the first condition of the two above is allowed. It means, that if whenever a set \( X \) determines functionally an attribute \( A \), then \( X \) is a superkey, i.e. determines the whole set \( U \).

The aim of a normalization process is to develop normal forms by analyzing functional dependencies and successive decomposition of the input relational schema into its projections. This way a well-designed schema can be obtained, where unnecessary redundancies and update anomalies had been eliminated. In practice, 3NF is accepted as the most desirable form of relational schemas. It does not eliminate all redundancies but guarantees dependency preservation. On contrast, BCNF eliminates all redundancies but does not preserve all dependencies. In [13] it was shown that 3NF has the least amount of redundancy among all dependency preserving normal forms. The research adopts a recently proposed information-theoretic framework for reasoning about database designs [10].

C. Relational analysis of XML schema

Let us consider the relational representation of data structure presented in Fig. 1. Then we have the following relational schema:

\[
\mathcal{R} = (U,F), \quad U = \{ old, sId, pId, price, delivTime, delivPlace \}, \quad F = \{ old \rightarrow sId, pId, price, delivTime, delivPlace, \quad sId, pId \rightarrow delivPlace, \quad delivPlace \rightarrow sId \}.
\]

In \( R \) there is only one key. The key consists of one attribute \( old \) since all attributes in \( U \) functionally depends on \( old \). Thus, \( R \) is in 2NF and \( old \) is the only prime (key) attribute in \( \mathcal{R} \). Additionally, we assume that a given supplier delivers a given part exactly to one place (\( pId, sId \rightarrow delivPlace \)). Moreover, delivery place \( delivPlace \) is connected with only one supplier (\( delivPlace \rightarrow sId \)).

\( R \) is not in 3NF because for the functional dependency \( sId, pId \rightarrow delivPlace \):
- \( sId, pId \) is not a superkey, and
- \( delivPlace \) is not a prime attribute in \( U \).

Similarly for \( delivPlace \rightarrow sId \).

The lack of 3NF is the source of redundancies and update anomalies. Indeed, for example, the value of \( delivPlace \) will be repeated as many times as many different tuples with the same value of the pair \( sId, pId \) exist in the instance of \( \mathcal{R} \). To eliminate this drawback, we can decompose \( \mathcal{R} \) into two relational schemas, \( \mathcal{R}_1 \) and \( \mathcal{R}_2 \), which are in 3NF. The decomposition must be based on the dependency \( sId, pId \rightarrow delivPlace \) which guarantees that the decomposition preserves data. As a result, we obtain:

\[
\mathcal{R}_1 = \{ U_1, F_1 \}, \quad U_1 = \{ old, sId, pId, price, delivTime \}, \quad F_1 = \{ old \rightarrow sId, pId, price, delivTime \}.
\]

\[
\mathcal{R}_2 = \{ U_2, F_2 \}, \quad U_2 = \{ sId, pId, delivPlace \}, \quad F_2 = \{ sId, pId \rightarrow delivPlace, \quad delivPlace \rightarrow sId \}.
\]

The discussed decomposition is both data and dependencies preserving, since:

\[
R(U) = \pi_{U_1}(R) \Join \pi_{U_2}(R),
\]
for every instance \( R \) of schema \( \mathcal{R} \), and
\[
F = (F_1 \cup F_2)^+.
\]

However, we see that \( \mathcal{R}_2 \) is not in BCNF, since \( delivPlace \) is not a superkey in \( \mathcal{R}_2 \).

The lack of BCNF in \( \mathcal{R}_2 \) is the reason of redundancies. For example, in table \( \mathcal{R}_2 \) we have as many duplicates of \( sId \) as
many tuples with the same value of delivPlace exist in this table.

We can further decompose $R_2$ into BCNF schemas $R_{21}$ and $R_{22}$, taking delivPlace $\rightarrow$ sid as the base for the decomposition. Then we obtain:

\[
R_{21} = (U_{21}, F_{21}), \text{ where } U_{21} = \{\text{delivPlace, sid}\}, \quad F_{21} = \{\text{delivPlace} \rightarrow \text{sid}\}.
\]

\[
R_{22} = (U_{22}, F_{22}), \text{ where } U_{22} = \{\text{pId, delivPlace}\}, \quad F_{22} = \emptyset.
\]

After applying this decomposition to $R_2$ we obtain tables $R_{21}$ and $R_{22}$:

<table>
<thead>
<tr>
<th>$R_{21}$</th>
<th>$R_{22}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sId$</td>
<td>$pId$</td>
</tr>
<tr>
<td>s1</td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
</tr>
</tbody>
</table>

This decomposition is information preserving, i.e.

\[ R_2 = R_{21} \bowtie R_{22}, \]

but does not preserve functional dependencies, i.e.

\[ F_2 \neq (F_{21} \cup F_{22})^+ = F_{21}. \]

We can observe some negative consequences of the loss of functional dependencies in the result decomposition.

Assume that we insert the tuple $(p1, d2)$ into $R_{22}$. The tuple will be inserted because it does not violate any constraint imposed on $R_{22}$. However, taking into account table $R_{21}$ we see that supplier $s1$ (determined by $d2$ in force of delivPlace $\rightarrow$ sid) offers part $p1$ in the place $d1$. Thus, the considered insertion violates functional dependency $sid, pId \rightarrow \text{delivPlace}$ defined in $R_2$.

The considered example shows that in the case of relational databases we are not able to completely eliminate redundancies and also preserve all functional dependencies. It turns out ([13]) that the best form for relation schema is 3NF, although some redundancies in tables having this form can still remain.

In next section we will show that the hierarchical structure of XML documents can be used to overcome some of the limitations of relational normal forms [11]. As it was shown in [12], there are decompositions of XML schemas that are both information and dependency preserving. In particular, we can obtain a form of XML schema that is equivalent to BCNF, i.e. eliminates all redundancies, and additionally preserves all XML functional dependencies.

IV. XML SCHEMAS AND INSTANCES

Schemas for XML data are usually specified by DTD or XSD [17]. In this section we will define XML schema by means of tree-pattern formulas (TPF) [18], [15]. Furthermore, we do not consider attributes in XML trees since they can always be represented by elements. Schemas will be used to specify structures of XML trees. Some other properties of XML trees are defined as schema constraints.

Definition 4.1: Let $L$ be a set of labels, and $x$ be a vector of variables. A schema TPF over $L$ and $x$ is an expression conforming to the syntax:

\[ S ::= /l[E] \]

\[ E ::= l = x | l[E] | E \land \ldots \land E, \]

where $l \in L$, and $x \in x$. In order to indicate the set and ordering of variables in $S$ we will write $S(x)$.

Example 4.1: For the schema tree from Fig. 1, the schema TPF has the following form:

\[ S_1 = /db/part/pId = x_1 \land \text{supplier}[	ext{offer}[	ext{oid} = x_2 \land \text{sid} = x_3 \land \text{price} = x_4 \land \text{delivTime} = x_5 \land \text{delivTime} = x_6]][]. \]

We see that a schema TPF has the following properties:

- reflects the tree structure of XML data,
- binds variables to paths in the schema tree,
- is a well-formed XPath predicate according to [19].

Definition 4.2: Let $S$ be a schema TPF over $x$ and let an atom $l = x$ occur in $S$. Then the path $p$ starting in the root and ending in $l$ is called the type of the variable $x$, denoted $\text{types}(x) = p$.

The type of $x_1$ in $S_1$ is: $\text{types}(x_1) = /db/part/pId$.

An XML database consists of a set of XML data. We define XML data as an unordered rooted node-labeled tree (XML tree) over a set $L$ of labels, and a set $Str \cup \{\bot\}$ of strings and the distinguished null value $\bot$ (both strings and the null value, $\bot$, are used as values of text nodes).

Definition 4.3: An XML tree $I$ is a tuple $(r, N^e, N^t, \text{child}, \lambda, \nu)$, where:

- $r$ is a distinguished root node. $N^e$ is a finite set of element nodes, and $N^t$ is a finite set of text nodes;
- $\text{child} \subseteq \{\{r\} \cup N^e\} \times (N^e \cup N^t) – a relation introducing tree structure into the set $\{r\} \cup N^e \cup N^t$, where $r$ is the root, each element node has at least one child (which is an element or text node), text nodes are leaves;
- $\lambda : N^e \rightarrow L – a function labeling element nodes with names (labels);
- $\nu : N^t \rightarrow Str \cup \{\bot\} – a function labeling text nodes with text values from $Str$ or with the null value $\bot$.

□
It will be useful to perceive an XML tree $I$ with schema $S$ over tuple of variables $x$, as a pair $(S, \Omega)$ (called a description), where $S$ is the schema TPF, and $\Omega$ is a set of valuations of variables in $x$. A valuation $\omega \in \Omega$ is a function assigning values from $\text{Str} \cup \{\bot\}$ to variables in $x$, i.e. $\omega : x \to \text{Str} \cup \{\bot\}$.

Example 4.2: The instance $I_1$ in Figure 3 can be represented by the following description:

\[
I_1 := (S_1(x_1, x_2, x_3, x_4, x_5, x_6), \{(p_1, o_1, x_1, x_1, t_1, d_1), (p_1, o_2, x_1, x_2, t_2, d_1), (p_1, o_3, x_2, t_3, x_3, d_2), (p_2, o_4, x_1, x_4, t_4, d_1))\}
\]

An XML tree $I$ satisfies a description $(S, \Omega)$, denoted $I \models (S, \Omega)$, if $I$ satisfies $(S, \omega)$ for every $\omega \in \Omega$, where this satisfaction is defined as follows:

Definition 4.4: Let $S$ be a schema TPF over $x$, and $\omega$ be a valuation for variables in $x$. An XML tree $I$ satisfies $S$ by valuation $\omega$, denoted $I \models (S, \omega)$, if the root $r$ of $I$ satisfies $S$ by valuation $\omega$, denoted $(I, r) \models (S, \omega)$, where:

1) $(I, r) \models \langle \langle [l]E \rangle, \omega \rangle$, if $\exists n \in \text{N}^*$ child($r, n) \land (I, n) \models \langle [l]E \rangle, \omega \rangle$;

2) $(I, n) \models \langle [l][E_1 \land \ldots \land E_k], \omega \rangle$, iff $\lambda(n) = l$ and $\exists n_1, \ldots , n_k \in \text{N}^*$ child($n_i, n_i) \land (I, n_i) \models (E_i, \omega)$ for $1 \leq i \leq k$;

3) $(I, n) \models \langle l = x, \omega \rangle$, if $\lambda(n) = l$ and $\exists n' \in \text{N}^*$ child($n, n') \land \nu(n') = \omega(x)$.

A description $(S, \Omega)$ represents a class of $S$ instances with the same set of values (the same $\Omega$), since elements in instance trees can be grouped and nested in different ways.

For example, the XML tree in Fig. 4 satisfies two descriptions $(S_1, \Omega_1)$, and $(S_2, \Omega_2)$ where:

\[
S_1 = \langle A \rangle B = x_1 \land C = x_2, \\
\Omega_1 = \{(b, c_1), (b, c_2)\};
\]

\[
S_2 = \langle A \rangle B = x_1 \land C = x_2 \land D = x_3, \\
\Omega_2 = \{(b, c_1, d_1), (b, c_2, d_1)\}.
\]

Fig. 4. A simple XML tree

V. XML FUNCTIONAL DEPENDENCIES AND KEYS

Over a schema TPF we can define some constraints, which specify functional dependencies between values and/or nodes in instances of the schema. These constraints are called XML functional dependencies (XFD).

Definition 5.1: An XML functional dependency (XFD) over a set $L$ of labels and a set $x$ of variables is an expression with the syntax:

\[
fd ::= /P[C]/.../P[C], \\
P ::= l \mid P/l, \\
C ::= \text{TRUE} \mid P \lor C \land \ldots \land C,
\]

where $l \in L$, and $x$ is a variable in $x$. If variable names are significant, we will write $fd(x)$.

An XFD is an XPath expression that for a given valuation $\omega$ of its variables returns a sequence of objects (nodes or text values). An XML tree $I = (S, \Omega)$ satisfies an XFD $f(x_1, \ldots , x_k)$ if for each valuation $\omega \in \Omega$ of its variables, $f(x_1, \ldots , x_k)$ returns a singleton.

Definition 5.2: Let $I$ be an instance of schema TPF $S(x)$ and $f$ be an XFD defined over $S$. The instance $I$ satisfies $f$, denoted $I \models f$, if for every valuation $\omega$ of variables in $x$, the implication holds

\[
I \models (S, \omega) \Rightarrow \text{count}(\{f(\omega)\}) \leq 1,
\]

where $\{f(\omega)\}$ is the result of $f(\omega)$ computed by the valuation $\omega$.

In the following example we discuss some XFDs over $S_1$.

Example 5.1: Over $S_1$ the following XFDs can be defined:

\[
f_1(x) = /db/part[\text{supplier}/of\text{fer}/oId = x], \\
f_2(x) = /db/part[\text{supplier}/of\text{fer}/oId = x]/pId, \\
f_3(x_1, x_2) = /db/part[pId = x_1]/\text{supplier}/of\text{fer}[ \\
\quad \text{delivPlace} = x_2]/sId, \\
f_4(x) = /db/part/\text{supplier}/of\text{fer}[ \\
\quad \text{delivPlace} = x]/sId, \\
f_5(x_1, x_2) = /db/part[pId = x_1]/\text{supplier}/of\text{fer}[ \\
\quad \text{of\text{fer}/sId} = x_2].
\]

According to XPath semantics [19] the expression $f_1(x)(\omega)$ is evaluated against the instance $I_1$ as follows: (1) first, a set of nodes of type /db/part is chosen; (2) next, for each chosen node the predicate [supplier/of\text{fer}/oId = x] is tested, this predicate is true in a node $n$, if there exists a path of type supplier/of\text{fer}/oId in $I_1$ leading from $n$ to a text node with the value $\omega(x)$. We see that $\text{count}(\{f_1(x)(\omega)\})$ equals 1 for all four valuations satisfied by $I_1$, i.e. for $\omega_i = [x \mapsto o_i], 1 \leq i \leq 4$.

Similarly, execution of $f_2(x)(\omega)$ returns a text value of the path /db/part/pId, where from the set of nodes determined by /db/part are taken only nodes satisfying the predicate [supplier/of\text{fer}/oId = \omega(x)]. We see that also this XFD is satisfied by $I_1$.

However, none of the following XFDs is satisfied in $I_1$:

\[
g_1(x) = /db/part[\text{supplier}/of\text{fer}/sId = x], \\
g_2(x) = /db/part[pId = x]/\text{supplier}/of\text{fer}/sId \\
g_3(x) = /db/part[pId = x]/\text{supplier}/of\text{fer}[ \\
\quad \text{delivPlace} = x], \\
g_4(x) = /db/part/pId/\text{supplier}/of\text{fer}[ \\
\quad \text{delivPlace} = x].
\]
Evaluating the above XFDs against $I_1$, we obtain:

\[
\begin{align*}
\text{count}(\{g_1(x)| (x \mapsto s1)\}) &= 2, \\
\text{count}(\{g_2(x)| (x \mapsto p1)\}) &= 2, \\
\text{count}(\{g_3(x)| (x \mapsto p1)\}) &= 3, \\
\text{count}(\{g_4(x, y)| (x \mapsto p1, y \mapsto s1)\}) &= 2, \\
\text{count}(\{g_5(x)| (x \mapsto d1)\}) &= 3.
\end{align*}
\]

An XFD can determine functional relationship between a tuple of text values of a given tuple of paths and a path denoting either a text value (e.g. $f_2(x)$) or a subtree (a node being the root of the subtree) (e.g. $f_1(x)$).

There are also different notations to express functional notations. For example, according to the notation used in [11], the XFD $f_5(x_1, x_2)$ can be expressed as:

\[
db.part @ pId, db.part.supplier.offer @ sId \rightarrow \\
db.part.supplier.
\]

and using the language proposed in [20], as

\[
(db.part, \{pId\}), \\
(db.part, \{supplier, \{offer/sId\}\}),
\]

where the first expression is an absolute key saying that $db.part$ is absolutely determined by $pId$, and the second expression is a relative key saying that in the context of $db.part$ a tree $supplier$ is determined by the path $offer.sId$.

Further, on, by $(S, F)$ we will denote a schema TPF $S$ together with a set of XFDs defined over $S$. The closure of $(S, F)$, denoted by $(S, F)^+$, is the schema TPF $S$ and the set of all XFDs which can be derived from $(S, F)$.

An XFD must be implied by $f$, i.e.

\[
(S, F) \text{ is in XNF iff } f/l \in (S, F)^+ \Rightarrow f \in (S, F)^+.
\]

Intuitively, let $f(x_1, \ldots, x_k)/l$ be XFD and $f$ be an instance of $S$. $I$ satisfies $f(x_1, \ldots, x_k)/l$, if for any valuation $\omega$ of the tuple of variables $(x_1, \ldots, x_k)$, there is at most one text value of the type $type(f/l)$. Thus, to avoid redundancies, for every valuation of $(x_1, \ldots, x_k)$ we should store the value of $f/l$ only once, i.e. there must be only one subtree of type $type(f)$ denoted by the expression $f(x_1, \ldots, x_k)$. In other words, XFD $f$ must be implied by $(S, F)$ [11].

Let us consider schema $S_2$ in Fig. 5 and its instance $I_2$ in Fig. 6. Then

\[
f(x_1)/sId = /db/part/supplier/delivPlace = x_1)/sId
\]

is XFD over $S_2$. This dependency corresponds to relational functional dependency $delivPlace \rightarrow sId$ and says that delivery place determines the supplier ($sId$).

VI. NORMAL FORM FOR XML

To eliminate redundancies in XML documents, some normal forms for XML schemas have been proposed [21], [11], [2], [12]. We will define XNF (XML Normal Form) using tree-pattern formulas and functional dependencies defined in the previous section and adapting the approach proposed in [21].

\[
\text{Definition 6.1: Let } S \text{ be a schema TPF and } F \text{ be a set of functional dependencies over } S. (S, F) \text{ is in XML normal form (XNF) iff for every XFD } f/l \in (S, F)^+, \text{ also } f \in (S, F)^+ \text{, i.e.}
\]

\[
(S, F) \text{ is in XNF iff } f/l \in (S, F)^+ \Rightarrow f \in (S, F)^+.
\]

\[\square\]
However, $S_2$ is not in XNF, since its instance $I_2$ does not satisfy

$$f(x_1) = /db/part/supplier[delivPlace = x_1],$$

because for the valuation $\omega = [x_1 \mapsto "d1"]$ there are two different element nodes supplier with value $d1$ of delivPlace. It means that $I_2$ is not free of redundancy.

In the case of schema $S_3$ (Fig. 7) the corresponding XFD has the form:

$$f(x_1)/sId = /db/supplier/part/delivPlace = x_1/sId.$$ This dependency also and the XFD

$$f(x_1) = /db/supplier/part/delivPlace = x_1$$

are satisfied by $I_3$ in Fig. 8. We see that this time for the valuation $\omega = [x_1 \mapsto "d1"]$, there is only one element node of type supplier from which we can reach delivPlace with value $d1$. This means that schema $S_3$ not only captures the XML functional dependency under consideration, but also is free of redundancies which may be caused by capturing this dependencies that happens in the case of schema $S_2$.

The other dependency of interest is $sId, pId \rightarrow delivPlace$. Its specification with respect to $S_2$ and $S_3$ is as follows:

$$/db/part/pId = x_1]/supplier[sId = x_2]/delivPlace,$$

and

$$/db/supplier[sId = x_1]/part[pId = x_2]/delivPlace.$$ It is easy to show that if these XFDs are satisfied by valuations, respectively, $\omega$ and $\omega'$ in instances of $S_2$ and $S_3$, then also

$$/db/part[pId = x_1]/supplier[sId = x_2],$$

and

$$/db/supplier[sId = x_1]/part[pId = x_2]$$

are satisfied by these valuations and these instances.

However, neither $S_2$ nor $S_3$ is in XNF. We have already shown that there is redundancy in instances of $S_2$. Similarly, we see that also in instances of $S_3$ redundancies may occur. Indeed, since one part may be delivered by many suppliers then the description of one part may be multiplied under each supplier delivering this part, so such data as part name, type, manufacturer etc. will be stored many times.

In Fig. 9 there is the final schema, $S_4$, for schemas under considerations: $S_1$, $S_2$, and $S_3$; $S_4$ is in XNF. To make the example more illustrative, we added node name to part data. Also the instance in Fig. 10 was slightly extended as compared with instances $I_2$ and $I_3$.

XSD (XML Schema Definition) for $S_4$ in notation proposed in [17] is shown in Fig. 11.

Note that we cannot use DTD since there are two subtrees labeled part, where each of them has different type: the part subtree under supplier consists of pId and delivPlace, whereas the part subtree under parts consists of pId and price. Recall that in the case of DTD each nonterminal symbol (label) can have only one type (definition), i.e. can appear on the left-hand side of exactly one production rule [17].

A set of XFDs for $S_4$ is defined in Fig. 12. XFDs derived from them are listed in Fig. 13.
We see that the schema $S_4$ satisfies the condition of XNF. Thus, this schema is both redundant-free and dependency preserving.

VII. CONCLUSION

In this paper, we discussed how the concept of database normalization can be used in the case of XML data. Normalization is commonly used to develop a relational schema free of unnecessary redundancies and preserving all data dependencies existing in application domain. In order to apply this approach to design XML schemas, we introduced a language for expressing XML functional dependencies. In fact, this language is a class of XPath expressions, so its syntax and semantics are defined precisely. We define the notion of satisfaction of XML functional dependence by an XML tree. To define XNF we use the approach proposed in [11].

All considerations are illustrated by the running example. We discuss various issues connected with normalization and compare them with issues faced in the case of relational databases. We show how to develop redundancy-free and dependency preserving XML schema. It is worth mentioning that the relational version of the schema cannot be structured in redundancy-free and dependency preserving form. In this case, preservation of all dependencies requires 3NF but then some redundancy is present. Further normalization to BCNF eliminates redundancies but does not preserve dependencies. In the case of XML, thanks to its hierarchical nature, we can achieve both properties. However, it is not clear if this is true in all cases (see e.g. [12]).

In [15], [22], [23], XML functional dependencies (XFD) have been used in XML data integration settings, in particular for controlling query propagation in P2P environment and for reconciliation of inconsistent data.

Acknowledgement: This work was supported in part by the Polish Ministry of Science and Higher Education under grant 0014/R/2/T00/06/02.

REFERENCES