RIAL: Redundancy Reducing Inlining Algorithm to Map XML DTD to Relations

Amir Jahangard Rafsanjani  
Sharif University of Technology  
Jahangard@ce.sharif.edu

Seyed-Hassan Mirian-Hosseinabadi  
Sharif University of Technology  
hmirian@sina.sharif.edu

Abstract

XML has emerged as a common standard for data exchange over the World Wide Web. One way to manage XML data is to use the power of relational databases for storing and querying them. So the hierarchical XML data should be mapped into flat relational structure. In this paper we propose an algorithm which maps DTD to relational schema and as well as content and structure it preserves the functional dependencies during the mapping process in order to produce relations with less redundancy. This is done by categorizing functional dependencies and introducing four rules to be applied to the relations created by the hybrid inlining algorithm according to each category. These rules will reduce redundancies by moving attributes, creating relations, introducing keys and preserving functional dependencies.

1. Introduction

XML[1] has become the de facto standard for data exchange on the World Wide Web. With the increase of XML Data the need for efficient ways to query over these data also increases. There are multiple approaches for querying XML data[13]. One is to use the relational database techniques i.e. storing and querying XML data using relational database management systems. The main challenge in this approach is that XML uses hierarchical data model while relational data model is flat, so storing XML data in relational databases is not a straight forward task.

Multiple algorithms have been proposed for mapping XML data to relations [2,3,4]. These algorithms are focused on structure and content. But in many cases semantic constraints are as important as structure and content for users. The users need to map XML into relations which preserve semantic constraints of XML data.

In order to introduce semantic constraints (e.g. keys, functional dependencies, etc) to XML, different approaches have been proposed [5,6,7,8,12]. Among those XML functional dependencies have special importance in representing semantic constraints [5,9]. In relational model functional dependencies (FDs) are used for finding keys, normalizing relations and avoiding update anomalies. We can use FDs in the XML to relational mapping process to produce better relational schemas.

There are some works in the context of porting XML semantic constraints to relational data [10,11]. In [11] an algorithm based on the reference hybrid algorithm [2] for mapping XML to relations has been proposed which preserves the constraints defined by Keys and KeyRefs. [10] is also based on hybrid algorithm. This algorithm extracts constraints from DTD (Document Type Definition)[14] and applies them to relations but it does not consider functional dependencies. In [17] a method is introduced to map XML DTD to relational schema which preserves FDs by creating a relation for each.

In this paper a method is proposed for mapping XML DTD to relational schema which considers XML functional dependencies. This paper is
organized in five sections. Section 2 introduces a few preliminary definitions are given. Our mapping algorithm is proposed in section 3. A case study is presented in section 4 and the paper is concluded in section 5.

2. Preliminary definitions

XML is a textual view of hierarchical data model. Before presenting the proposed algorithm we need to have a formal definition of functional dependency for XML data, we use the definitions introduced in [5] as the basis of our approach:

Assume that we have the following disjoint sets: El of element names, Att of attribute names, Str of possible values of string-valued attributes, and Vert of node identifiers. All attribute names start with the symbol @, and these are the only ones starting with this symbol. We let S and ⊥ (null) be reserved symbols not in any of those sets.

A DTD D is defined to be $D = (E, A, P, R, r)$, where $E \subseteq El$ is a finite set of element types, $A \subseteq Att$ is a finite set of attributes, $P$ is a set of rules $\tau \rightarrow P_i$ for each $\tau \in E$, where $P_i$ is a regular expression over $E - \{r\}$, $R$ assigns a subset of $A$ to each element $\tau \in E$, and $r \in E$ is the root.

An XML tree is a finite rooted directed tree $T = (N, G)$, where $N$ is the set of nodes, and $G$ is the set of edges, together with a labeling function $\lambda : N \rightarrow El$ and an attribute function $\rho@a : N \rightarrow Str$ for each $@a \in Att$. We say tree $T$ conforms to DTD $D = (E, A, P, R, r)$, written as $T \models D$, if the root of $T$ is labeled $r$, and for every $x \in N$ with $\lambda(x) = a$, the word $\lambda(x_1) \ldots \lambda(x_n)$ is in the language defined by $P_a$ where $x_1, \ldots, x_n$ are children of $x$ in order, $@l \in R(a)$ iff the function $\rho@l$ is defined on $x$.

Given a DTD $D = (E, A, P, R, r)$, an element path $q$ is a word in the language $E^*$, and an attribute path is a word of the form $q.@l$, where $q \in E^*$ and $@l \in A$. An element path $q$ is consistent with $D$ if there is a tree $T \models D$ that contains a node reachable by $q$; if the nodes reachable by $q$ have attribute $@l$, then the attribute path $q.@l$ is consistent with $D$. The set of all paths consistent with a DTD $D$ is denoted by $paths(D)$. The last element type that occurs on a path $q$ is called last($q$). The set of all paths consistent with a DTD $D$ that end with an element type is called $Epaths(D)$.

Given an XML tree $T = (N, G)$ such that $T \models D$, a tree tuple [5] is a subtree of $T$ rooted at $r$ containing at most one occurrence of every path. Intuitively, the set of all tree tuples in $T$ forms a relational representation of $T$. Formally, a tree tuple is a mapping $t : paths(D) \rightarrow N \cup Str \cup \{\bot\}$ such that if for an element path $q$ whose last letter is $a$, we have $t(q) \neq \bot$, then $t(q) \in N$ and $\lambda(t(q)) = a$; if $q$ is a prefix of $q$, then $t(q) \neq \bot$ and $t(q)$ lies on the path from the root to $t(q)$ in $T$; if $@l$ is defined for $t(q)$ and its value is in $Str$, then $t(q.@l) = v$.

A functional dependency over a DTD $D$ [5] is an expression of the form $\{q_1, \ldots, q_n\} \rightarrow q$, where $q, q_1, \ldots, q_n \in paths(D)$. A tree $T$ satisfies an $FD \{q_1, \ldots, q_n\} \rightarrow q$ if for any two tree tuples $t_1, t_2$ in $T$, whenever $t_1(q_i) = t_2(q_i) \neq \bot$ for all $i \in [1..n]$, then $t_1(q) = t_2(q)$.

Given a DTD $D$ and a set $\Sigma$ of FDs over $D$, $(D, \Sigma)^+$ is the set of all FDs implied by $(D, \Sigma)$. An FD is called trivial if it belongs to $(D, \emptyset)^+$.

```xml
<!DOCTYPE db [
  <!ELEMENT db (conf*)>]
<!ELEMENT conf (title, issue*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT issue (#PCDATA)>
<!ELEMENT inproceedings (author*, title)>
<!ATTLIST inproceedings
  key ID #REQUIRED
  pages CDATA #REQUIRED
  year CDATA #REQUIRED>
<!ELEMENT author EMPTY>
<!ATTLIST author
  name CDATA #REQUIRED
  email CDATA #REQUIRED
  institute CDATA #IMPLIED>
]

Figure 1. DTD Example

Example 1. (From [5] with some modifications) Figure 1 shows an example of a DTD. This DTD expresses that every conference has a title and one or more issue. Papers are stored in inproceedings elements. As it's common in regular expressions * shows zero or more repetitions. Two keywords #PCDATA and CDATA are used as string data type for elements and attributes.
The formal representation of the DTD of Example 1 will be as follows (\$ means #PCDATA and \(\varepsilon\) means EMPTY):

- \(E = \{db, conf, title, issue, inproceedings, author\}\)
- \(A = \{@key, @pages, @year, @name, @email, @institute\}\)
- \(P(db) = conf^*\)
- \(P(conf) = title, issue^*\)
- \(P(title) = S\)
- \(P(issue) = inproceedings^*\)
- \(P(inproceedings) = author^*\)
- \(R(db) = R(conf) = R(issue) = R(title) = \{\}\)
- \(R(inproceedings) = \{@key, @pages, @year\}\)
- \(R(author) = \{@name, @email, @institute\}\)
- \(r = db\)

The set of all paths (paths(d)) for DTD of Figure 1 is shown in Figure 2.

![Figure 2. Set of all paths in DTD of Example 1](image)

Now we can specify functional dependencies:

- **FD1**: \(db.conf.title.S \rightarrow db.conf\)
- **FD2**: \(db.conf.issue \rightarrow db.conf.issue.inproceedings.@year\)
- **FD3**: \(\{db.conf.issue, db.conf.issue.inproceedings.title.S\} \rightarrow db.conf.issue.inproceedings\)
- **FD4**: \(db.conf.issue.inproceedings.@key \rightarrow db.conf.issue.inproceedings\)
- **FD5**: \(\{db, db.conf.issue.inproceedings.author.@email\} \rightarrow db.conf.issue.inproceedings.author.@name.S\)

Constraint FD1 enforces that two distinct conferences have distinct titles. Given that an issue of a conference represents a particular year of the conference, constraint FD2 enforces that two articles of the same issue must have the same value in the attribute year. Constraint FD3 enforces that for a given issue of a conference, two distinct articles must have different titles. Constraint FD4 enforces that key is an identifier for each article in the database. Finally constraint FD5 enforces that two distinct author with the same email address must have the same name.

### 3. Redundancy Reducing algorithm

Just like relational data, XML semantic constraints can be shown by functional dependencies. In this section we propose an algorithm to map XML to relations that reduces redundancy and preserves functional dependencies. It means that if the FDs are known for XML data, these constraints as well as content and structure are preserved during the mapping process.

The proposed method is based on the hybrid inlining algorithm [2]. In this algorithm the DTD will be simplified in a way that it does not affect the relational schema generated from the DTD. The transformations are of three types:

a. **Flattening** transformations which convert a nested definition into a flat representation (i.e., one in which the binary operators "," and "|" do not appear inside any operator)

\[(e_1, e_2) \rightarrow e_1^*, e_2^*\]
\[(e_1^*, e_2) \rightarrow e_1^?, e_2^?\]
\[(e_1, e_2^?) \rightarrow e_1^?, e_2^?\]

b. **Simplification** transformations which reduce many unary operators to a single unary operator

\[e_1^* \rightarrow e_1^*\]
\[e_1^? \rightarrow e_1^?\]

(c) **Grouping** transformations that group sub-elements having the same name (for example, two \(e^*\) sub-elements are grouped into one \(e^*\) sub-element).

\[... , e^*, ... , e^* , ... \rightarrow e^* , ...\]
\[... , e^* , ... , e^? , ... \rightarrow e^* , ...\]
\[... , e^? , ... , e^* , ... \rightarrow e^* , ...\]
\[... , e^* , ... , e^? , ... \rightarrow e^* , ...\]
\[... , e^? , ... , e^* , ... \rightarrow e^* , ...\]

(d) In addition, all "+*" operators are transformed to "*" operators.

**Example 2.** Using the above simplification procedure, \(<!ELEMENT a ((b|c)|e)?\), ...
(e?([f?,(b,b)*]))*) can be transformed to
<e?((b*,c?,e*,f*))>

Figure 3 shows this algorithm. The model of
DTD used in[2] is not rooted so any element can be
considered as the root and one should create a DTD
graph for every element in the model and then
consider the union of the relations created by these
graphs as the result relation set. But in our
approach we consider a rooted DTD model so we
only need to create a DTD graph for the root
element.

1. Simplify the DTD using the transformations
above.
2. Construct a DTD Graph for the root element
which shows the structure of the DTD. The
graph nodes are elements, attributes or
operators in the DTD; each element appears in
the graph just once while attributes and
operators appear as many times as they
appear in DTD.
3. For unmarked edges in graph, if they construct
a loop, choose one of them (edges with “*”
have higher priority), break the loop, create a
relation for the end node and mark the edge.
4. For each unmarked edge in the DTD graph, if it
has “*” create a relation for the end element
and mark the edge.
5. Create a relation for the root.
6. For each unmarked element, Inline the end
element into start element and do this until
there exists a relation for the start element.
7. Preserve parent/child relationship by
including a “parentid” in the child relation
referencing “id” in parent relation

Figure 3. Hybrid inlining algorithm

Figure 4 shows the DTD graph constructed
from DTD of Example 1.

In order to reduce redundancies and preserve
the functional dependencies we categorize
functional dependencies into four categories and
introduce a rule for each category that will be
applied to the relations created by the inlining
algorithm, but before applying these rules we
introduce a rule which we can call it Rule0 and
applies to all type of FDs in the set:

- Rule0 : for each path p ∈ EPaths(D) if p is
on the left hand side (LHS) or on the right
hand side (RHS) of a FD then a relation is
created for the last(p) if it is not already
created by the inlining algorithm, i.e.
last(p) should not be inlined in other
relations.

Now we introduce the rules:

- Rule1 (Attribute Moving): For each FD of
the form q → p.@l where q,p ∈
EPaths(D), if q → p ∈ (D,Σ)* then inline
@l in the relation correspondent to
last(q).

- Rule2 (Relation Creating): For each FD of
the form \{q,p₁, @l₁,...,pₙ@lₙ\} → p.@l
where q ∈ EPaths(D) and n ≥ 1, create a
new relation with @l₁, @l₂,..., @lₙ as
attributes and set a foreign key referring to
the relation created for last(q) or the
relation in which last(q) is inlined and a
foreign key referring to the new created
relation is added to the relation created for
p. The key of this relation will be composition of $@l_1, ..., @l_n$.

- **Rule3 (Key Generating):**
  o For each FD of the form $p. @l_1, ..., p. @l_n \rightarrow p$ where $p \in \text{EPaths}(D)$ set $@l_1, ..., @l_n$ as the primary key of the relation. We can here remove the surrogate key introduced by the inlining algorithm.
  o For each FD of the form $q_1, S, ..., q_n, S \rightarrow p$ where $p \in \text{EPaths}(D)$ and $p$ is a proper prefix of $q_1, 1 \leq i \leq n$, set $\text{last}(q_1), ..., \text{last}(q_n)$ as the primary key of the relation. We are sure that $\text{last}(q_i)$ is inlined in $p$, because there cannot be a *-edge between $\text{last}(p)$ and $\text{last}(q_i)$.

- **Rule4 (FK/CX Generating):** For each FD of the form \{ $p_1, ..., p_n, @l \}$ $\rightarrow$ $p$ where $p, p_1, ..., p_n \in \text{EPaths}(D)$, consider $R, R_1, ..., R_n$ are relations for $\text{last}(p), \text{last}(p_1), ..., \text{last}(p_n)$ respectively. For each $R_i, 1 \leq i \leq n$, a foreign key $f_{k_i}$ is added to $R$ referring to $R_i$ if it already is not present. The composition these foreign keys with $@l$ will be a candidate key for $R$.

By mapping the DTD using the hybrid inlining algorithm the FDs like the one introduced in Rule1 are lost because the LHS and the RHS will be mapped into two different relations; so the Rule1 is to preserve these kinds of dependencies in the target relations by moving attributes. By applying Rule2 a new relation is created and by setting the correct key for this relation the FD is preserved. By applying Rule3, the relation does not need the surrogate key introduced by the inlining algorithm and the redundancy is reduced. Rule4 preserves another type of FD by introducing foreign key to the appropriate relations and making a composite key out of them. This approach will not reduce redundancy but it preserves the semantic constraints introduced by FDs.

Because the set of FDs is finite and we apply the rules on the relations created by the inlining algorithm it’s guaranteed that the algorithm will be terminated finally.

---

1 Foreign key / Candidate Key

**Theorem 1. (Complexity).** Our inlining algorithm can be performed in $O(n + f)$ where $n$ is the number of elements in the input DTD $D$, and $f$ is the number of FDs defined over $D$.

**Proof.** It’s obvious because in inlining algorithm each node of the DTD graph is visited at most once and for applying our rules we need to consider each FD once.

4. **Case Study**

Consider the DTD in Figure 1. The mapping will be as follows:

1. Construct the DTD graph as shown in Figure 4.
2. There is no loop in the graph so do nothing in the step.
3. For elements conf, issue, proceedings and author which have an incoming "*" edge a separate relation is created.
4. A relation is created for the root element (db).
5. Attributes $@key$, $@pages$, $@year$, $title$ are added to proceedings relation. Attributes $@name$, $@email$, $@institute$ are added to author relation. Attribute $title$ is added to conf relation.
6. A foreign key to relations conf, issue, proceedings and author in order to preserve parent/child relationships.

At the end the relations will be:

- db ($@id$
- conf ($@id, dbid, title$
- issue ($@id, confid$
- proceedings ($@id, issueid, @key, @pages, @year, title$
- author($@id, inprocid, @name, @email, @institute$

Note that the dependency between issue and @year is lost and the relation author is not in third normal form (3NF) because of the FD $@email \rightarrow @name$.

Now we apply our rules to the algorithm:

- Applying Rule1: Because of FD2 in Example 1, the attribute $@year$ is removed from proceedings relation and is added to issue relation.
- Applying Rule2: Because of FD5 in Example 1, a new relation with attributes $@name$, $@email$, $dbid$ is created (we name it
Another aspect of research is the reverse mapping considering these constraints in our future work. There are also other constraints like multi-valued dependencies and the foreign keys are updated.

Applying Rule 4: Because of FD3 in Example 1, (considering title as @l) two attributes issueid and title are chosen as a candidate key for inproceedings relation.

So after applying the rules the relations will be as follows:

- `db (#id)`
- `conf (dbid, title)`
- `issue (#id, confid, @year)`
- `inproceedings (issueid, @key, @pages, title)`
- `author(#id, inprocid, @institute, authorid)`
- `authorInfo(@email, @name, dbid)`

It is obvious that the redundancy is reduced and the FDs are preserved during the mapping process. And we can see that all the relations are in BCNF because the LHS of all FDs is the key of the relation.

5. Conclusion and future work

XML functional dependencies are similar to functional dependencies in relational model. In relational model the main application of FDs are decreasing redundancies and avoiding anomaly in updating data. In hierarchical data redundancy is an inevitable fact. With the introduction of functional dependencies for XML these redundancies can be specified. The relations which are made from these data should not have the same problems. In this paper we presented an algorithm that maps XML to relations and preserves these constraints as well as content and structure. There are some other XML schemas such as XML Schema that can be studied as a future work. There are also other constraints like multi-valued dependencies that should be considered in the mapping process. We plan to consider these constraints in our future work. Another aspect of research is the reverse mapping from relational schemas to XML schemas considering functional dependencies.

6. References


