KEYS GRAPH - BASED RELATIONAL TO XML TRANSLATION ALGORITHM

Wilmodes Manzi de Arantes Júnior\(^{(1, 2)}\), Christine Verdier\(^{(1)}\)

\(^{(1)}\) Lyon Research Center for Images and Intelligent Information Systems (LIRIS), INSA Lyon, 20 avenue Albert Einstein, Villeurbanne, France

\(^{(2)}\) Calystène Informatique Santé, 32 avenue du Vercors, Meylan, France

Keywords: Relational to XML translation, Keys graph, Functional dependencies

Abstract: The authors propose two algorithms for generating a DTD and an XML document respectively from the metadata and the content of a relational database without any intermediary language or user intervention. Such algorithms always generate semantically correct XML output by respecting database functional dependencies represented in a graph structure they take as input. Finally, different XML representations (or views) meeting expectations of different kind of users can be obtained from the same data according to the data entity chosen as translation pivot.

1 INTRODUCTION

In the last years, much have been said about XML and its applications. However, the majority of the business data are stored in relational databases and needs to be translated. In this paper, we present two separated algorithms for translating the structure and the content of a relational database respectively in a DTD and an XML document. The algorithms use a keys graph (Flory & Kouloumdjian, 1978) (automatic generation in (Manzi, Verdier & Flory, 2002)) to represent all functional dependencies in the database for ensuring that translation results reflect accurately semantic relationships between data entities. The algorithms can also generate XML output reflecting data from the point of view of a particular data entity (from a database containing professors and courses, we can create, for example, a professor-centered and a course-centered document for different purposes). Finally, no intermediary mapping languages nor user intervention are required.

2 EXAMPLE DATABASE

The example database we will use throughout this paper is the following:

![Diagram](image)

Figure 1: example database.

<table>
<thead>
<tr>
<th>course table</th>
<th>student table</th>
</tr>
</thead>
<tbody>
<tr>
<td>course-id</td>
<td>course-name</td>
</tr>
<tr>
<td>C1</td>
<td>French</td>
</tr>
<tr>
<td>C2</td>
<td>sport</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prof table</th>
<th>univ table</th>
</tr>
</thead>
<tbody>
<tr>
<td>prof-id</td>
<td>prof-name</td>
</tr>
<tr>
<td>P1</td>
<td>John</td>
</tr>
<tr>
<td>P2</td>
<td>Paul</td>
</tr>
<tr>
<td>P3</td>
<td>Carl</td>
</tr>
<tr>
<td>P4</td>
<td>Phil</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>univ-id</th>
<th>univ-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>INSA</td>
</tr>
<tr>
<td>U2</td>
<td>Lyon1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>scores table</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>prof-id</td>
<td>course-id</td>
<td>student-id</td>
</tr>
<tr>
<td>P1</td>
<td>C1</td>
<td>S1</td>
</tr>
<tr>
<td>P1</td>
<td>C1</td>
<td>S2</td>
</tr>
<tr>
<td>P1</td>
<td>C2</td>
<td>S1</td>
</tr>
<tr>
<td>P2</td>
<td>C1</td>
<td>S2</td>
</tr>
</tbody>
</table>
2.1 Table/entity-centered translations

Some algorithms transform relational data in such a way that all tags and attributes in the resulting XML document represent database tables, rows, columns, data types, field lengths, default values and so on. We call this kind of transformation table-centered. In this paper, we follow an entity-centered approach in which the XML document we generate contains only high-level concepts present in the database Entity-Relationship model: data entities, associations (represented by element nestings) and attributes.

3 DTD GENERATION

This algorithm is executed according to a data entity called pivot node which determines the meaning of the resulting XML representation since all database content is rearranged in order to present data from its point of view. The steps of the algorithm are:

3.1 Choosing the pivot node

As the translation always begins with a data entity, the pivot node must be intermediary. Suppose we have chosen prof-id:

3.2 Traversing the sub-graph below it

In this phase, the algorithm visits the sub-graph 1. The first node to be analyzed is the pivot node itself, which is an intermediary one. Then we:

(A) create a composite DTD element having the same name as the node table (prof) and whose children list is initially empty;
(B) create a new PCDATA element having the same name as the node attribute (prof-id);
(C) add the name of the DTD element created in B to the children list of the element created in A.

Next step consists in traversing all non-visited edges starting at the pivot node. Next node is prof-name, which is a leaf one. Then we:
(D) create a new PCDATA element having the same name as the attribute of the node (prof-name).

Now, we will represent in the DTD the edge linking prof-id and prof-name by creating a nesting between the DTD elements generated by these nodes. So, we:
(E) add the name of the DTD element created by the destination node in D to the children list of the DTD element created by the origin node in A/C:

Next two nodes we visit are univ-id and univ-name, which are treated according to the rules used in A, B and C. So we have three new elements:

Finally, we indicate there is an edge between prof-id and univ-id by creating a nesting between the DTD elements they created:

3.3 Traversing the sub-graph above it

Now, we will traverse the sub-graph 2. Next node is the head of the graph which, differently from leaf and intermediary ones, does not create any DTD element. As the order in which branches starting at a head node are visited determines the meaning of the translation result, they are sorted so that branches starting with key attributes (e.g. course-id) appear
before branches starting with relationship attributes (e.g. score).

Once graph branches are ordered, the algorithm traverses each non-visited one. Each time it finishes visiting a branch b, we indicate that b is linked to the graph head by creating a nesting between the DTD elements generated by the first node of b and by the first node of the branch visited immediately before b. For example, the first node of the branch starting with course-id create the following DTD element:

```xml
<ELEMENT course (course-id, course-name)>
```

Then, for indicating the link between this branch and the graph head, we add the name of this element to the children list of the element created by the first node of the last visited branch (starting at prof-id):

```xml
<ELEMENT course (course-id, course-name)>
```

The next branch we visit starts with student-id node and its relationship with the last visited one (starting with course-id) is indicated as follows:

```xml
<ELEMENT student (student-id, student-name)>
```

Finally, we reach the nodes representing relationship attributes, and all remaining nestings will be made between the PCDATA elements they create and the composite element created by the first node of the last branch starting with a key attribute (student-id):

```xml
<ELEMENT score (#PC)>
<ELEMENT result (#PC)>
```

In the next section we present an algorithm for predicting the cardinalities of all nestings we have created so far.

### 3.4 Determination of cardinalities

Each time we create a nesting between two elements E1 and E2, we predict the cardinality ω of E2 with relation to E1 (```<!ELEMENT E1(E2ω)>```) as follows:

(A) If we are analysing a key attribute contained in an intermediary node, the cardinality is 1..1 for sure.

```xml
<ELEMENT tab (att, ...)>
```

for example, the cardinality of the key attribute att in the children list of tab element is 1..1 for sure.

(B) If we are going down between two graph nodes, the cardinality is 1..1 for sure because upper attributes functionally determines lower ones.

For example, att2 and att3 have cardinalities 1..1 for sure in the children list of tab element:

```xml
<ELEMENT tab (att1, att2, att3, ...)> 
```

(C) If we are going up or at the same level in the graph, the destination node attribute is not functionally determined by the origin node one. Then, we query the database and the cardinality is predicted by composing the two rules below:

Rule 1: If at least one instance of the origin node attribute is linked to no instances of the destination node attribute THEN the minimum cardinality is 0 for sure, ELSE it can be 1;

Rule 2: If at least one instance of the origin node attribute is linked to several instances of the destination node attribute, THEN the maximum cardinality is N for sure, ELSE it can be 1. The composition table is:

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Result</th>
<th>Likelihood</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>apply</td>
<td>0..N (*)</td>
<td>sure</td>
</tr>
<tr>
<td>apply</td>
<td>not apply</td>
<td>0..1 (?)</td>
<td>not sure</td>
</tr>
<tr>
<td>not apply</td>
<td>apply</td>
<td>1..N (+)</td>
<td>not sure</td>
</tr>
<tr>
<td>not apply</td>
<td>not apply</td>
<td>1..1 (−)</td>
<td>not sure</td>
</tr>
</tbody>
</table>

For example, when going from prof-id to course-id nodes, we predict the cardinality of course element in the children list of prof element by applying these rules to the scores table. Rule 1 applies as at least one value of the origin node is linked to no value of the destination node (P3 has no entries in the table). Rule 2 applies as at least one value of the origin node is linked to several values of the destination node (P1 is linked to C1 and C2). So, the first line of the composition table states that the cardinality of course element is 0..N ("*" symbol) for sure.

```xml
<ELEMENT prof (prof-id, prof-name, univ, course)*> 
```

The final DTD the algorithm generates is (PCDATA elements are note included for space reasons):

```xml
<ELEMENT prof (prof-id, prof-name, univ, course)> 
```

```xml
<ELEMENT course (course-id, course-name, student)> 
```

```xml
<ELEMENT student (student-id, student-name, score, result)> 
```

The complete algorithm for generating a DTD from a relational database is presented in figure 3.
4 XML DOCUMENT GENERATION

Our second algorithm generates an XML document from a relational database. In this document, tags reflect database structure (as described by its DTD) and contents are retrieved from database tables. The algorithm starts at a pivot node and visits all nodes below and above it generating XML tags and SQL queries. In our example, suppose we chose prof-id attribute as pivot, which is intermediary. Then we:

(A) create an empty XML tag (element) having the same name as the node table (prof):

```
<prof></prof>
```

because it is the pivot node, we create an SQL query for retrieving all values of its attribute (prof-id) from its table (prof). The query and the result are:

```
SELECT prof-id FROM prof
```

(B) visit, for each retrieved value, all subsequent graph nodes. The first value is P1. Then, we create an XML tag having the same name as the node attribute (prof-id) and whose value is P1, and we add this new tag into the tag created in A (which is initially empty):

```
<prof-id> P1 </prof-id>
```

Figure 3: algorithm for generating a DTD from a relational database.

(C) visit the sub-graph below the pivot node for the value P1. Next node, prof-name, is a leaf. Then, we create an SQL query for retrieving the value of this attribute as functionally determined by the actual value of the father node attribute (prof-id = P1). The query and the result are:

```
SELECT prof-name FROM prof
WHERE prof-id = P1
```

John

Next node, univ-id, is intermediary, so the process is the same as for prof-id. Then, we represent the edge linking univ-id to prof-id through a nesting between the XML tags representing them.

Now, the translation algorithm goes up in the graph and reaches its head. Again, it traverses all
non-visited graph branches from left to right creating nestings linking the actual branch either to the last-visited or to the last one starting with a key attribute. All branches must be ordered as stated before.

Next branch starts with course-id node. Then, we retrieve all values of its attribute as functionally determined by the combination of the values of the previous visited nodes starting with key attributes (prof-id = P1). In other words, we want to know all courses taught by professor P1:

Once again, the algorithm must visit all subsequent graph nodes for each retrieved value. For course-id = C1, an XML tag is created and added into the tag representing the last visited graph branch, prof-id:

| SELECT course-id FROM score WHERE prof-id = P1 GROUP BY course-id | C1 | C2 |

For the next branches, we must combine the values of all already visited key attributes (prof-id = P1 and course-id = C1). Next one starts with student-id:

When traversing a branch, the function 

```java
FUNCTION buildXML (GraphNode NBV, XMLElement ELEM, ANDClauses CLAUSES, Str tableName, int IND)
returns XMLElement
```

is called with the pivot node of the translation.

### Figure 4: algorithm for generating an XML from a relational database.

![Detailed algorithm for generating an XML from a relational database](image)

Then, for each retrieved value, we must traverse the branch starting with student-id and add the created tag into the tag created by course-id branch.

The last branches contain relationship attributes and must be linked to the last visited branch starting with a key attribute (student-id). Again, their values are functionally determined by the combination of

### function call

GraphNode PN = pivot node of translation
XMLElement rootElement = new XMLElement ("database","")
buildXML (PN, rootElement, [],"",0)
the values of the previous visited key nodes, prof-id = P1, course-id = C1 and student-id = S1:

| SELECT score FROM scores WHERE (prof-id = P1) AND (course-id = C1) AND (student-id = S1) |
| SELECT result FROM scores WHERE (prof-id = P1) AND (course-id = C1) AND (student-id = S1) |

As score and result are leaves, their tags are added to the tag created by student-id node:

XML tags
created by score
and result nodes

Although the graph traversal is finished at this point, the created XML document contains only data about professor P1, course C1 and student S1. Then, for translating available data about the other elements, we must revisit previous visited branches starting with key attributes from right to left in order to take into account all possible combinations of values of these three attributes in the database. According to the scores table, such combinations are:

- prof-id = P1, course-id = C1, student-id = S1
- prof-id = P1, course-id = C1, student-id = S2
- prof-id = P1, course-id = C2, student-id = S1
- prof-id = P2, course-id = C1, student-id = S3

Now we come back to the last visited branch starting with a key attribute, student-id, whose next value is S2 and we re-traverse all subsequent graph nodes. At this point, all values of student-id will be analyzed, then we come back to the prior branch, course-id. Its next value is C2. Again, all remaining branches are visited. The translation is complete when the graph is traversed for all of the combinations above.

The complete algorithm for generating an XML document from a relational database is presented in figure 4.

6 CONCLUSION

We have presented two algorithms for translating the structure and the content of a relational database respectively into a DTD and an XML document. They ensure the semantic correctness of the result by respecting database functional dependencies thanks to a directed graph indicating them. Additionally, these algorithms can create different entity-centered views of the same data. Finally, they require no user intervention, nor intermediary languages specifying mapping schemes. In the future, some improvements can be made in order to reduce the redundancy in the final XML document and the great number of SQL queries executed against the database.

REFERENCES


