Converting relational database into XML documents with DOM

J. Fonga,*, H.K. Wonga, Z. Chengb

aDepartment of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong, People’s Republic of China
bDepartment of Electronic Engineering, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong, People’s Republic of China

Received 14 October 2002; revised 4 January 2003; accepted 4 February 2003

Abstract

The revolution of XML is recognized as the trend of technology on the Internet to researchers as well as practitioners. Companies need to adopt XML technology. With investment in the current relational database systems, they want to develop new XML documents while running existing relational databases on production. They need to reengineer the relational databases into XML documents with constraints preservation. In the process, schema translation must be done before data conversion. Since the existing relational databases are usually normalized, they have to be reconstructed into XML document tree structures. This can be accomplished through denormalization by joining the normalized relations into tables according to their data dependencies constraints. The joined tables are mapped into DOMs, which are then integrated into XML document trees. The user specifies an XML document root with its relevant nodes to form a partitioned XML document tree to meet their requirements. The selected XML document tree is mapped into an XML schema in the form of DTD. We then load joined tables into DOMs, integrate them into a DOM, and transform it into an XML document.

Keywords: XML document; Denormalization; Relational database; Schema translation; Data conversion; Data dependencies; Document object model; Document type definition

1. Introduction

XML databases use XML documents as fundamental units, define a model such as elements, attributes, PCDATA, etc. for an XML instance, and store data as either binary code or text file. XML documents have been widely used on the Internet for business in both B2B and B2C. We expect a strong need to migrate relational databases into XML documents.

To make relational schema compatible with the XML schema, we denormalize the relational schema into joined tables which are transformed into DOMs according to their data dependency constrains. These DOMs are integrated into a DOM which is translated into an XML document tree. Upon user selection, we then translate the partitioned XML document tree into an XML schema. We load each tuple of the joined tables into the object instances of elements in the XML documents according to the translated XML schemas and their data dependencies.

The data dependencies constraints in the denormalized relational schema are mapped into XML document trees in elements and sub-elements. In the process, the partial functional dependencies are mapped into elements and attributes. The transitive data dependencies are mapped into element, sub-element, and sub-sub-element in the XML documents. The multi-valued dependencies are mapped into multiple sub-elements under one element. The join dependencies are mapped into a group element. As a result, the data semantics in the relational schema are translated and preserved in the XML document tree.

An XML document tree is represented as a linked list—one element follows another. We employ DOM for implementation (Fig. 1). Each joined table with data dependencies is translated into a DOM. These DOMs will be integrated into a DOM, and then translated into an XML document. When a DOM parser reads in XML document, it creates a document object first, and the whole XML document is traversed from this point. For example, below is an XML document with implementation using DOM:

* Corresponding author. Tel.: +852-2788-8580; fax: +852-2788-8614.
E-mail addresses: csjfong@cityu.edu.hk (J. Fong), hkwong@cs.cityu.edu.hk (H.K. Wong).

0950-5849/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0950-5849(03)00026-0
The relational schema is in table format accessed by Structured Query Language (SQL) matching data value between tables. The XML schema is in tree format accessed by XML Query Language (XQL) navigating through an XML document tree. To make relational tables compatible with XML documents, we denormalize the relational database into joined tables, translate them into DOMs, integrate them into a DOM and map it into an XML schema. We load each tuple of the joined tables into object instance in DOMs and then transform it into an XML document according to their data dependencies as shown in Fig. 2.

**Notations.**

**Functional dependency.** A functional dependency is a statement of the form \( X \rightarrow Y \), where \( X \) and \( Y \) are sets of attributes. The FD: \( X \rightarrow Y \) holds for relation \( R \) if whenever \( s \) and \( t \) are tuples of \( R \) where \( s[X] = t[X] \), then \( s[Y] = t[Y] \).

**Multi-valued dependency.** Let \( R \) be a relation, and let \( X, Y, \) and \( Z \) be attributes of \( R \). Then \( Y \) is multi-dependent on \( X \) in MVD: \( X \rightarrow Y[Z] \) if and only if the set of \( Y \)-values matching a given \((X\text{-value}, Z\text{-value})\) pair in \( R \) depends only on the \( X \)-value and is independent of the \( Z \)-value.

**Join dependency.** Let \( R \) be a relation, and let \( A, B, \ldots, Z \) be arbitrary subsets of the set of attributes of \( R \). Then JD holds for relation \( R \) if \( R = \prod[R[A]...R[Z]] \). That is, JD holds if and only if \( R \) is equal to the join of its projections on \( A,\ldots,Z \).

**Transitive dependency.** A functional dependency \( X \rightarrow Y \) in a relation schema \( R \) is a transitive dependency if there is a set of attribute \( Z \) that is neither a candidate key nor a subset of any key of \( R \), and both \( X \rightarrow Z \) and \( Z \rightarrow Y \) hold.

**Partial dependency.** A functional dependency \( X \rightarrow Y \) is a partial dependency if some attribute \( A \in X \) can be removed from \( X \) and the dependency still hold.

2. Related works

2.1. On XML transformation language

VXMLR [1] presents a visual based XML document management system. First an XML document is parsed into a DOM tree and the DTD of the document is extracted. Then the document tree is mapped into a relational table which is stored in a database. For processing XML queries, the path expressions queries are transformed into SQL statements submitted to the underlying RDBMS. SilkRoute [2] describes a general framework for mapping relational databases to XML virtual views using a declarative query language, RXL (Relational to XML Transformation Language). The resultant view is formulated by application using XML-QL to extract XML data. Using XRel [3], an XML document is decomposed into a set of nodes that are
stored in several tables along with encoded path information from the root to each node. XRel stores XML documents using a fixed relational schema without any information about DTDs and utilizes indices such as the B1-tree supported by DBMS. To process XML queries, it presents an algorithm for translating a core subset of XPath expressions into SQL queries. Shanmugasundaram et al. [4] proposes an SQL language extension, an XML constructor, for constructing complex XML documents directly in the relational engine. They explore different execution plans for generating the content of an XML document. The result showed that constructing XML documents inside the relation engine could have significant performance benefit. The mediator [16] converts data from relational databases into XML format and integrates data from different sources into a common view. It contains a query rewriting mechanism for both data conversion and data integration. For example, users can pose queries on the XML view and rewrite them to queries over the relational schema.

2.2. On XML middleware

XPERANTO [5] translates XML-QL into SQL requests using XML Query Graph Model (XQGM), receives tabular query result, tags the XPERANTO middleware layer to produce resultant XML documents, and returns the requested document to the users. Varlamis and Vazirgiannis [6] develop the X-Database system acting as an interface between the application and database. The base of the system is an XML-Schema describing the logical model of interchanged information. First, the system analyzes the syntax of the XML-Schema file and generates the relational schema. Then it stores XML documents’ information in the database and creates XML documents from the database contents. Finally, the system offers a mechanism for updating and querying databases contents using exclusively XML documents. Shanmugasundaram et al. [7] explore conservative approach of using relational database engines for processing XML documents conforming to DTD. It processes DTDs to generate relational schemas, parses XML documents, loads them into tuples of relational tables in database, translates XML queries into SQL statement, and converts the results back to XML documents. Khan and Rao [8] propose an automatic mapping technique from an XML document to RDBMS which preserves the nested structure and stores information into the relation in an encoded form. The mapping technique relies on both the content and the DTD of the document, which handles query effectively by reducing the number of join operations. For processing XML queries, they propose a mechanism for translating and XPath expression into SQL statement. They suggest two alternative techniques for addressing the matter of the restructuring aspect of the mapping.
2.3. On XML technology

WebReader [9] is a middleware between the browser and the web for automating the search and collecting information from the web. By facilitating metadata specification in XML and manipulation in XSL, WebReader provides users with a centralized, structured, and categorized means to specify and query web information. An experimental prototype based on XML, XSL and Java was developed to show the feasibility and practicality of the authors’ approach through a real life application example. Fong et al. [10] design a method to translate XQL into SQL in an XML gateway. The translation process adopts symbolic transformation of node navigation in XQL query graph to the relation join table navigation in SQL query graph. Fong and Dillon [11] compare the performance analysis between XML-Enabled Database and Native XML database and recommends Native XML database for very complex structure system. Fong et al. [15] outline steps for converting relational database into XML document. These steps show how to translate relational schema into XML schema, and mapping relational data to an XML document.

2.4. On data conversion

Ref. [12] offers a methodology for translating hierarchical and network schemas to relational schemas. The methodology applies reverse engineering to extract entities and relationships into an extended entity relationship model which is mapped into a relational schema. Fong and Bloor [13] propose a method for converting network database into relational database. The author claims that the universe of data to be modeled can be represented by one relation in a relational schema. Any relation in the database is a projection of a universe relation. Ref. [14] presents techniques for converting relational databases into object-oriented databases. The data conversion involves unloading tuples of relations into sequential files, and reloading them into object-oriented databases with constraints preservation. Currently most industry software tools can only convert a few relations into an XML document without constraints preservation. This paper translates a view of relational database into an XML document for mass information transmission on the Internet. The resultant novel findings is a methodology of implementing information highway for web-based systems, including the translation from relational databases to XML documents with preservation of data dependency constraints.

3. Methodology of converting relational database into XML document

By following the stepwise procedure in Fig. 2, we perform the following steps.

3.1. Step 1 Denormalize relational schema

We denormalize relational schema into joined tables. In the process, we classify each attribute in the relation for the recovery of their constraints. The relations are preprocessed by making any necessary candidate key substitutions as shown below:

<table>
<thead>
<tr>
<th>Description</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR1</td>
<td>Primary relation describes entities. Primary relation Type 1 is a relation whose primary key does not contain a key of another relation</td>
</tr>
<tr>
<td>PR2</td>
<td>Primary relation Type 2 is a relation whose primary key does contain a key of another relation</td>
</tr>
<tr>
<td>SR1</td>
<td>Second relation has primary key which is fully or partially formed by concatenation of primary keys of other relations. Secondary relation Type 1 has the key of the secondary relation formed fully by concatenation of primary keys of primary relations</td>
</tr>
<tr>
<td>SR2</td>
<td>Secondary relation Type 2 is a relation of secondary relations that are not of Type 1</td>
</tr>
<tr>
<td>KAP</td>
<td>Key Attribute Primary is an attribute in the primary key of a secondary relation that is also a key of some primary relation</td>
</tr>
<tr>
<td>KAG</td>
<td>Key Attribute General is all the other primary key attributes in a secondary relation that are not of the KAP type</td>
</tr>
<tr>
<td>FKA</td>
<td>Foreign Key Attribute is a non-primary key attribute of a primary relation that is a foreign key</td>
</tr>
<tr>
<td>NKA</td>
<td>Non-key Attributes are the rest of the non-primary key attributes</td>
</tr>
</tbody>
</table>

We apply classification table to recover data dependency in the relational database, which is translated into XML documents as shown below.
Algorithm

Begin

Map relational schema into a classification table of relations;
Recover data dependency of relations from the classification table;
Denormalize relational schema into joined tables according recovered constraints;

Case joined table consisting of:
(i) functional dependency: translate joined table into a single sub-element XML document;
(ii) multi-valued dependency: translate jointed table into multiple sub-element XML document;
(iii) join dependency: translate joined table into a group element XML document;
(iv) m:n cardinality: translate jointed table into a referred element XML document;

Case End;
End;

The methodology details in the algorithm are described in the following four cases.

Case (i) Transform recovered functional dependency into a single sub-element XML document. Given receiver’s relations R1(A1, A3) and R2(A2, A4, *A1) with FD (functional dependency): R2.A1 → R1.A1, they are classified and joined into a table R(A1, A2, A3, A4) which is then translated into a single sub-element XML document by mapping parent relation R1 into element E1, and child relation R2 into sub-element E2 as follows:

<table>
<thead>
<tr>
<th>Relation name</th>
<th>Relational type</th>
<th>Primary key</th>
<th>KAP</th>
<th>KAG</th>
<th>FKA</th>
<th>NKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>PR1</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td>A3</td>
</tr>
<tr>
<td>R2</td>
<td>PR1</td>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td>A4</td>
</tr>
</tbody>
</table>

As a result, the FD is preserved after translation as shown below:

Case (ii) Transform recovered multi-valued dependency into a multiple sub-element XML document. Given receiver’s relations Relation R1(*A1, A2) and R2(*A1, A3) with MVD (multiple value dependency): A1 → A2/A3, they are classified and joined into a table R(R1, R2, R3) which is then translated into a multiple sub-elements XML document by mapping KAP A1 into element E1. KAG A2 and A3 into sub-elements E2 and E3 as follows:

<table>
<thead>
<tr>
<th>Relation name</th>
<th>Relational type</th>
<th>Primary key</th>
<th>KAP</th>
<th>KAG</th>
<th>FKA</th>
<th>NKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>PR2</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td>A2</td>
</tr>
<tr>
<td>R2</td>
<td>PR2</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td>A3</td>
</tr>
</tbody>
</table>

As a result, the MVD is preserved after translation as shown below:

```
MVD: A1 → A2
and A1 → A3
```

Case (iii) Transform recovered join dependency into a group element XML document. Given receiver’s relations R1(*A1, A2), R2(*A2, A3) and R3(*A3, A1) with JD (join dependency): R = R1 ⨝ R2 ⨝ R3, they are classified and joined into a table R(A1, A2, A3) which is then translated into a group element XML document by mapping KAP A1, A2 and A3 into a group of elements E1, E2 and E3 as follows:

<table>
<thead>
<tr>
<th>Relation name</th>
<th>Relational type</th>
<th>Primary key</th>
<th>KAP</th>
<th>KAG</th>
<th>FKA</th>
<th>NKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>SR1</td>
<td>A1, A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>SR1</td>
<td>A2, A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>SR1</td>
<td>A3, A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As a result, the JD is preserved after translation as shown below:

<table>
<thead>
<tr>
<th>JD:</th>
<th>Translated XML document</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ R1(A1, A2) ⊓ R2(A2, A3) ⊓ R3(A3, A1)</td>
<td></td>
</tr>
</tbody>
</table>

**Case (iv) Transform recovered m:n cardinality into a referral element XML document.** Given receiver’s relations R1(A1, A3), R2(A2, A4) and R3(‘A1’, ‘A2’) with MVD R1(A1, A3) → R2(A2, A4). They are classified and joined into a table R(A1, A2, A3, A4) which is then translated into a referral element XML document by mapping KAP A1 into attribute ‘id’ of element E1, and KAP A2 into attribute ‘idref’ of element E2 as follows:

<table>
<thead>
<tr>
<th>Relation name</th>
<th>Relational type</th>
<th>Primary key</th>
<th>KAP</th>
<th>KAG</th>
<th>FKA</th>
<th>NKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>PR1</td>
<td>A1</td>
<td>A3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>PR1</td>
<td>A2</td>
<td>A4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>SR1</td>
<td>A1, A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>A1 A2 A3 A4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relation name</th>
<th>Relational type</th>
<th>Primary key</th>
<th>KAP</th>
<th>KAG</th>
<th>FKA</th>
<th>NKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>PR1</td>
<td>A1</td>
<td>A3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>PR1</td>
<td>A2</td>
<td>A4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>SR1</td>
<td>A1, A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As a result, the referral is preserved after translation as shown below:

<table>
<thead>
<tr>
<th>Relational database</th>
<th>Translated XML document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Referral: R1(A1, A3)</td>
<td>Referral: E1(A1, A3)</td>
</tr>
<tr>
<td>→ → R2(A2, A4)</td>
<td>→ → E2(A2, A4)</td>
</tr>
</tbody>
</table>

3.2. Step 2 Derive relevant classes

To convert a relational database into an XML document tree, we integrate the translated XML document trees into an XML document tree, select an element as root and put its relevant information into a document. We load relational database into object instances of XML documents. Each XML document focuses on a root class upon user requirements. The selection is driven by the business nature. Relevance concerns elements that are related to a root selected by the users among the integrated XML document tree (DOM tree). Fig. 3 shows an integrated XML document tree. The user can select root A1 with its relevant classes to form a partitioned XML document tree.

3.3. Step 3 Integration of the translated XML documents using DOMs

Each joined table will be translated into a DOM tree. These DOM trees will be integrated into a DOM tree, and then translated into an XML document tree as shown in Fig. 4. We can integrate the translated XML documents (DOMs) into an XML document as shown below.

Algorithm

```
Begin
    Create a DOM tree for each XML document;
    For each DOM instance do
    Begin
        Search for the same DOM instance in another DOM;
        If found
            Then delete a duplicate DOM instance;
            Chain the DOM instances;
        End;
    Map the integrated DOM into an XML document;
End
```

When a DOM parser reads an XML document, it creates a document object first, and then the whole XML document is traversed from this point. During DOM’s merge, we evaluate every element node in each DOM against others, not only
the structure of parent and children relationship, but also their values as well. We defined a search algorithm matching element node within n DOMs with an algorithm as shown below.

**Algorithm**

/* the algorithm matches same element in doc, and return the vector object if end meets.
   Given getNode (name, value, doc)
   name: the Node name
   value: the Node value
   doc: DOM needs to search
   r: the set of same Node in doc with Node passed
   return: the set of found Node in doc
*/

Begin
   While name Node still has sibling Node
      Do begin
         Get child list of name Node
         For each node in child list
            If node.value equals to value
               Then add name Node to r
         Next
      End
   Return r
End

Since we put DOMs with different data dependencies in the same program, there are duplicate data within them. To integrate DOMs, we focus on one main DOM tree, and delete duplicate elements in the others after their children elements have been appended to the same element in the program. The reason is not only to avoid double checking every time the program call getNode( ) method, but also to avoid duplicate appending. Basically, the recursive algorithm is divided into search, deletion and inserting as follows:
Algorithm

/* Given reSort (node)
   node: every node in mainly focus DOM
   n: node in child list set
   p: the property of node
   s: return vector of getNode
   return: void
*/

Begin
   Get p of n
   If p is DOCUMENT type node
      The reSort (Document element of n);
   If p is ELEMENT type node
      Begin
         While node still has sibling node
            Do begin
               Get child list of node
               For each n in child list
                  reSort (n)
               Next
            End
         If p is TEXT type node
            Then s = getNode (n, n.value, DOM_1)
               If s not equals to null
                  Then Delete s elements in DOM_1
                  Append after n in main DOM
               s = getNode (n, n.value, DOM_2)
               If s not equals to null
                  Then Delete s elements in DOM_2
                  Append after n in main DOM
         Return void
      End
   End

The algorithm checks the property of Node first. According to different Node type, TEXT Node will be checked within the other two. When the function finishes its job, an integrated DOM is created. Write2File (Document doc) is a method in Write class, which serializes DOM to String and saves it to a file.

Algorithm

void write2file (Document node) // output the xml document into text file
Begin
   set output format;
   set string write;
   create a XML serializer;
   create a file;
   print to the file;
End

For example, given a group element XML document tree and a multiple sub-elements XML document tree, we can integrate them by matching element as follows:
4. Case study

In a Bank Loan application, a loan with an identity number belongs to a customer who has a customer identity number. Customers have loans secured by loan securities. Customers open accounts at different branches with maturity dates. Each loan is charged with interest of an index type interest rate as shown in the following normalized relations.

Relation Customer–Security

<table>
<thead>
<tr>
<th>Customer</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>LS001</td>
</tr>
<tr>
<td>Bob Kusik</td>
<td>LS002</td>
</tr>
<tr>
<td>Chris Bloor</td>
<td>LS003</td>
</tr>
</tbody>
</table>

Relation Customer–Branch

<table>
<thead>
<tr>
<th>Customer</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>Bayview</td>
</tr>
<tr>
<td>John Doe</td>
<td>Main Street</td>
</tr>
<tr>
<td>Bob Kusik</td>
<td>Market Street</td>
</tr>
</tbody>
</table>

Relation Security–Loan

<table>
<thead>
<tr>
<th>Security</th>
<th>Loan</th>
<th>Maturity_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS001</td>
<td>K6200104827</td>
<td>1/2/2004</td>
</tr>
<tr>
<td>LS002</td>
<td>K6030400610</td>
<td>1/10/2001</td>
</tr>
<tr>
<td>LS003</td>
<td>K6080800810</td>
<td>12/12/2001</td>
</tr>
</tbody>
</table>

Relation Loan–Index

<table>
<thead>
<tr>
<th>Loan</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>K62001040827</td>
<td>Prime</td>
</tr>
<tr>
<td>K60304000610</td>
<td>Mortgage</td>
</tr>
<tr>
<td>K60808000810</td>
<td>Car Loan</td>
</tr>
</tbody>
</table>

Relation Loan–Customer

<table>
<thead>
<tr>
<th>Loan</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>K62001040827</td>
<td>John Doe</td>
</tr>
<tr>
<td>K60304000610</td>
<td>Bob Kusik</td>
</tr>
<tr>
<td>K60808000810</td>
<td>Chris Bloor</td>
</tr>
</tbody>
</table>
4.1. Step 1 Denormalize relational schema into joined tables

After classifying each attribute in a classification table, we can derive their constraints as follows:

<table>
<thead>
<tr>
<th>Relation name</th>
<th>Relational type</th>
<th>Primary key</th>
<th>KAP</th>
<th>KAG</th>
<th>FKA</th>
<th>NKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer–Security</td>
<td>SR1</td>
<td>Customer, Security</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security–Loan</td>
<td>SR1</td>
<td>Security, Loan</td>
<td></td>
<td></td>
<td></td>
<td>Maturity_date</td>
</tr>
<tr>
<td>Loan–Customer</td>
<td>SR1</td>
<td>Loan, Customer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

JD: \( R_1(\text{Customer}, \text{Security}) \bowtie R_2(\text{Security}, \text{Loan}, \text{maturity}_date) \bowtie R_3(\text{Loan}, \text{Customer}) \rightarrow \) denormalized joined table \( R(\text{Customer}, \text{Security}, \text{Loan}, \text{maturity}_date) \)

<table>
<thead>
<tr>
<th>Relation name</th>
<th>Relational type</th>
<th>Primary key</th>
<th>KAP</th>
<th>KAG</th>
<th>FKA</th>
<th>NKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer–Account</td>
<td>PR2</td>
<td>Customer</td>
<td></td>
<td></td>
<td></td>
<td>Account</td>
</tr>
<tr>
<td>Customer–Branch</td>
<td>PR2</td>
<td>Customer</td>
<td></td>
<td></td>
<td></td>
<td>Branch</td>
</tr>
</tbody>
</table>

MVD: Customer \( \rightarrow \) Account|Branch such that \( R_1(\text{Customer}, \text{Account}) \bowtie R_2(\text{Customer}, \text{Branch}) \rightarrow \) denormalized joined table \( R(\text{Customer}, \text{Account}, \text{Branch}) \)

<table>
<thead>
<tr>
<th>Relation name</th>
<th>Relational type</th>
<th>Primary key</th>
<th>KAP</th>
<th>KAG</th>
<th>FKA</th>
<th>NKA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loan–Index</td>
<td>PR1</td>
<td>Loan</td>
<td></td>
<td></td>
<td></td>
<td>Index</td>
</tr>
<tr>
<td>Index–Rate</td>
<td>PR1</td>
<td>Index</td>
<td></td>
<td></td>
<td></td>
<td>Rate</td>
</tr>
</tbody>
</table>

TFD: Loan \( \rightarrow \) Index \( \rightarrow \) Rate such that \( R_1(\text{Loan}, \text{Index}) \bowtie R_2(\text{Index}, \text{Rate}) \rightarrow \) denormalized joined table \( R(\text{Loan}, \text{Index}, \text{Rate}) \).

4.2. Step 2 Translating the denormalized joined tables into XML document trees

First, we transform the JD: (Customer, Security, Loan) into a group element in an XML document tree as follows:

```
XML document tree

Group

Customer Loan Security

Underlined means occurrence required
```

Second, we transform the MVD: Customer \( \rightarrow \) Account|Branch into multiple sub-elements in an XML document tree as follows:

```
XML document tree

Group

Customer Account Branch

Underlined means occurrence required
```
Third, we transform the TFD: Loan → Index → Rate into elements and sub-elements in an XML document tree as follows:

Finally, we integrate the translated XML document trees into an XML document tree in Fig. 5.

### 4.3 Step 3 Select XML document root

In this case study, we select Bank as the root of the XML document for our application. We then map the integrated XML document tree into XML DTD as shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<:ELEMENT Bank { Group+ } >
  <:ELEMENT Group { Customer, Loan, Security } >
    <:ELEMENT Customer ( Account*, Branch* ) >
      <:ATTLIST Customer customer-id CDATA #REQUIRED >
        <:ELEMENT Account ( #PCDATA ) >
          <:ELEMENT Branch ( #PCDATA ) >
            <:ELEMENT Security EMPTY >
              <:ATTLIST Security security-id NMTOKEN #REQUIRED >
                <:ELEMENT Loan (Index) >
                  <:ATTLIST Loan loan-id NMTOKEN #REQUIRED >
                    <:ATTLIST Loan maturity_date CDATA #REQUIRED >
                      <:ELEMENT Index ( Rate ) >
                        <:ATTLIST index-type CDATA #REQUIRED >
                          <:ELEMENT Rate EMPTY >
                            <:ATTLIST interest-rate NMTOKEN #REQUIRED >
```

### 4.4 Step 4 Loan the denormalized table into DOMs

We can download a reorganized relation R(1) with the JD: (Customer, Security, Loan) into a group element in an XML document DOM (1) as follows:

Relation R(1)
We can download a reorganized relation $R(2)$ with the MVD: $Customer \rightarrow \rightarrow Account|Branch$ into an XML document $DOM(2)$ as follows:

Reorganized relation $R(2) = Relation(Customer \rightarrow Account) \Join Relation(Customer \rightarrow Branch)$

Relation $R(2)$

<table>
<thead>
<tr>
<th>Customer</th>
<th>Account</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Doe</td>
<td>025056</td>
<td>Bayview</td>
</tr>
<tr>
<td>John Doe</td>
<td>027300</td>
<td>Main Street</td>
</tr>
<tr>
<td>Bob Kusik</td>
<td>080801</td>
<td>Market Street</td>
</tr>
</tbody>
</table>

XML Document $DOM(2)$

We can download a reorganized relation $R(3)$ with the TFD: $Loan \rightarrow Index \rightarrow Rate$ into an XML document $DOM(3)$ as follows:
Reorganized relation $R(3) = \text{Relation } Loan \times \text{Relation } Index \times \text{Relation } Rate$

<table>
<thead>
<tr>
<th>Loan</th>
<th>Index</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>K62001040827</td>
<td>Prime</td>
<td>7</td>
</tr>
<tr>
<td>K60304000b10</td>
<td>Mortgage</td>
<td>6</td>
</tr>
<tr>
<td>K60808000810</td>
<td>Car-loan</td>
<td>8</td>
</tr>
</tbody>
</table>

XML Document DOM (3)

```xml
<Loan loan-id="k62001040827">
  <Index index-type = "Prime">
    <Rate interest-rate = "7"></Rate>
  </Index>
</Loan>

<Loan loan-id="k60304000610">
  <Index index-type = "Mortgage">
    <Rate interest-rate = "6"></Rate>
  </Index>
</Loan>

<Loan loan-id="k60808000810">
  <Index index-type = "Car-load">
    <Rate interest-rate = "8"></Rate>
  </Index>
</Loan>
```

Then we can then integrated XML Document DOM (1), XML Document DOM (2), XML Document DOM(3) into an XML document as follows:

```xml
<?xml version = "1.0"?>
<Bank>
  <Group>
    <Customer customer-id="John Doe">
      <Account>025056</Account>
      <Branch>Bayview</Branch>
      <Account>027300</Account>
      <Branch>Main Street</Branch>
    </Customer>
    <Security security-id="LS001"></Security>
    <Loan loan-id="k62001040827" maturity_date="1/2/2004">
      <Index index-type="Prime">
        <Rate interest-rate="7"></Rate>
      </Index>
    </Loan>
  </Group>
  <Group>
    <Customer customer-id="Bob Kusik">
      <Account>080801</Account>
      <Branch>Market Street</Branch>
    </Customer>
    <Security security-id="LS002"></Security>
    <Loan loan-id="k60304000610" maturity_date="1/10/2001">
      <Index index-type="Mortgage">
        <Rate interest-rate="6"></Rate>
      </Index>
    </Loan>
  </Group>
  <Group>
    <Customer customer-id="Chris Bloor"></Customer>
    <Loan loan-id="k60808000810" maturity_date="12/12/2001">
      <Index index-type="Car-Loan">
        <Rate interest-rate="8"></Rate>
      </Index>
    </Loan>
  </Group>
</Bank>
```
5. Prototype

We have implemented a prototype to construct a relational database into an XML document according to user requirements and selection. For example, since the relation Customer, relation Security, and relation Loan have join dependency, we denormalized them into a joined table. We then created views towards these scenarios, respectively, for query convenience on the Microsoft SQL Server 2000.

For the MVD, we denormalized Customer_Account and Customer_Branch because Customer $\rightarrow$ Account/Branch. Since the three attribute Customer, Security, and Loan have multi-valued dependency, we denormalized Customer_Security, Loan_Customer and Security_Loan into a joined table and created a SQL view accordingly.

For the TFD, we denormalized Loan_Index and Index_Rate because Loan $\rightarrow$ Index $\rightarrow$ Rate into a joined table and created a SQL view.

After we had created views towards these scenarios, respectively, for query convenience on the Microsoft SQL Server 2000, without loss of the generality, we created join tables based on their data dependencies as follows:

1) JD_Customer_Security_Loan
   Design view: Fig. 6
   View properties: Fig. 7

2) MVD_Customer_Account_Branch
   Design view: Fig. 8

![Fig. 6. Joint dependency of relations Customer_Security, Loan_Customer and Security_Loan.](image1)

![Fig. 7. Creative view of joined table with JD.](image2)
After the denormalization was done, we created an application based on Java Swing Technique. Query result was saved to resultSet object in memory. The layout and result of the three DOMs is given in Fig. 12.

By applying three DOMs to algorithm reSort, we get an integrated DOM tree as shown in Fig. 13.

6. Performance evaluation

To access the relative performance of the algorithm, we performed several experiments on an IBM compatible computer with a Mobile Intel Pentium III CPU clock rate of 750 MHz, 128 Megabytes of main memory, and running Windows 2000 Professional. The data resided in the FAT 32 file system and was stored on a 20 Gigabytes Ultra-ATA hard disk. The relational database is Microsoft SQL Server 2000. All programs are written in Java 2.
Fig. 11. Create view of joined table with TFD.

Fig. 12. DOM/Tree structure display.
We query these three set of data: 200 datasets, 500 datasets, and 1000 datasets. In each category of query, three parses time (generate time, merge time and write file time) were recorded. We queried four times in every category and took average of them.

<table>
<thead>
<tr>
<th></th>
<th>200 data set</th>
<th>500 data set</th>
<th>1000 data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM generate time (ms)</td>
<td>1236.75</td>
<td>1622.5</td>
<td>2020</td>
</tr>
<tr>
<td>DOM merge time (ms)</td>
<td>3214.5</td>
<td>19,678.25</td>
<td>78,643</td>
</tr>
<tr>
<td>Write file (ms)</td>
<td>325.5</td>
<td>676</td>
<td>1354.5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4767.75</strong></td>
<td><strong>21,976.75</strong></td>
<td><strong>82,017.5</strong></td>
</tr>
</tbody>
</table>

![Integrated DOM display](image1)

![DOM merge time](image2)

Fig. 13. Integrated DOM display.

Fig. 14. DOM merge time.
Fig. 14 was compared with the DOM merge average time from 200 to 1000 data query. The slope between 500 and 1000 was much larger than that between 200 and 500, while other average time was quite the same between these three queries. Fig. 15 is a chart comparing DOM generation time, DOM merge time, and file write time. The X and Y axis represent three parse and data sets, respectively. The Z axis is the second take into account during data conversion. From the figure above, the DOM creation and file saving time were quite the same either for 1000 data query or 200 data query. While DOM integration time was a bottleneck at 1000 data query, the slop from 500 data to 1000 data was much higher than that from 200 data to 500 data. In theory, merge algorithm could be further improved if we avoided using recursive function.

7. Conclusions

Much work have been done on translating relational database into XML documents without consideration of data constraints preservation. Very often, they tend to translate only a few relations into an XML document. This paper proposes using denormalization and DOM for the task with more flexibility and constraints transfer. Our approach denormalizes relational schema into joined tables which are transformed into DOMs, and then integrated into an XML document tree (DOM tree). The data dependencies constraints in the relational databases are represented in the relationship between Element and Sub-element in the XML documents. Such compatibility provides a means of database reengineering between relational databases and XML documents. We can reengineer a relational database into XML documents upon user specification. The user selects an XML document root for each partitioned XML document for their requirements. The process provides flexibility for user to convert a view of relational database into an XML document. The future development of this paper is to utilize the preserved data dependencies to recover the original relational database from the translated XML documents for data exchange on the information highway.

References


