# Dynamic XML View Creation And Update Propogation Using Relational Database

Sanjay K Madria and Janarthanan Eindhal
Department of Computer Science
Missouri University of Science and Technology, Rolla, MO 65401
madrias@mst.edu

eXtensible Mark Language(XML) has already become the industry standard for exchange and representation of data. The ability of XML to provide semantics to the data it holds finds its purpose in many applications over diverse industries from petroleum to biology. While most of the existing data is stored and maintained in traditional databases there is a need for transformation of data from the database to represent them as XML views. The issues which are important are (a) how to convert data from a flat relational database into hierarchical and semi-structured XML data and vice-versa (b) how to reflect updates on the XML view in the databases and vice-versa. Hence, the problem of creating XML views from the Relational model and updating relational databases through XML views has become a primal issue and the center of attention for the research community. The aim of this paper is complete automation to create XML views of the relational database using the XML schema and the Database schema as input. A one to one mapping is defined between the data in the database and the XML; the relation between data is maintained. The paper provides a novel approach for updating the relational database based on the changes in the XML document with the help of XML change detection tools. The changes in XML provided by the change detection tool are converted into SQL queries to update the database.

Keywords: XML-Database mapping, Dynamic XML view management, Relational database

## 1. INTRODUCTION

Low cost data conversion, software independence, platform independence, data reusability, portability, accessibility and flexibility are some of the advantages that make eXtensible Markup Language(XML) [W3C] as the standard for data exchange. Most of the vendors of data management are fully involved in harnessing the capabilities that XML offers. XML by itself is self-describing in nature i.e. it not only provides data but also information about the data, in short called metadata. XML has been used as a data model for representing semi-structured (tree-structure) data [McHugh et al.1997].

Even though XML is an exiting area to explore in terms of data management, relational databases super cede them in terms of reliability, scalability, management tools and overall performance [Shanmugasundaram et al.2000]. While most of the companies store their information in traditional databases, they have a need to represent their data in XML format. For example consider a scenario where a company selling its products has to provide data to its customers and vendors. They have to present their data in a way the customers understand and thus arises the problem of how to convert the data from the flat database into semi-structured XML document. Also, when a vendor receives the XML data from the company there might be changes the vendor might impose like the price of an item quoted, adding new items to the list, removing items out of stock etc directly on the XML document and thus the problem of updating the database based on the modified XML arises. To sum up, an effective automatic translation technique is necessary in between the relational database and XML to represent data and also to capture the updates made on the XML view to reflect them in the database.

The problem of XML view maintenance over relational databases has been studied over a long period. It involves several issues like,

✦ ***Publishing relational data as XML views:*** The main issue here is the nature of the two storage models, namely the relational database and XML. While the relational database is flat

XML is hierarchical, tagged and tree-structured. There is a need for a one to one mapping between the data in the relations to the data in the XML. A simple and straight forward approach is to transform the data in each tuple of a table into XML with the column names as tags.

✦ ***Maintaining the view:*** After publishing the XML document if there is a change in the database then how to maintain the view is an important problem. Since relational database systems could be huge, the creation of view every time an update is made is time consuming and not an efficient method of maintenance. The changes have to be made incrementally in which the changes made to the database are integrated with the view without having to create the whole document from scratch.

✦ ***Updating the relational data:*** As mentioned in the scenario above, companies would like to update the XML views directly than maintaining them in the back end (relational database). The main issues faced here is how to detect the changes in the XML, what are the modifications done and how does it affect the database? Some translation methodology should exist which can convert the changes in the XML into basic SQL queries in order to update the relations. Also, updating relational database can cause ambiguity issues. Suppose an element was deleted in the XML document and the corresponding data in the relational table is deleted, it is quite possible that the data deleted might be related to some other table. Issues like these can incur data losses. In the following subsections the approach proposed in this paper to solve the issues of publishing relational data and updating the relational data are discussed.

## 1.1 DB2XML Conversion

The first problem addressed is to create the XML document from the relational database in short represented as DB2XML. The user provides the target XML specification as XML Schema along with the database schema, more on XML Schema can be found in Section 2.2. The system combines the information in the XML Schema and the database schema to generate SQL queries which when applied to the underlying database produces the XML document. The application is based on Nested Relational Algebra to generate the SQL queries. The benefit of this application is that it can be used for any well structured XML schema and the user can obtain the desired XML documents from the relational data. The user interference is avoided very much since the whole process is carried out automatically. Figure 1 shows the graphical representation of the basic model in creating XML views.

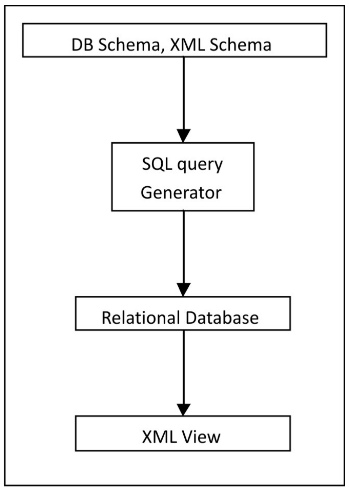


Figure. 1: Graphical representation of the DB2XML application model

### 1.2 Updating Relational Database Through XML Views

The second area focuses on capturing the updates made to the XML and modifying the relations based on the changes. There are two ways in which this could be obtained, one is using the virtual approach which insists on creating the view starting from scratch and the other one is materialized approach in which the updates are made as and when they occur and the cost of recreating the whole view is eliminated. The second approach is also called Incremental Recomputation. The use of Incremental Recomputation techniques [Qian et al.1991, Gupta et al.1999] are considered much better in performance than the default virtual approach starting from scratch.

The method proposed in this paper for automatically updating the database from XML document seeks the help of change detection tools, and is believed to be the first to address the issue through change detection. There are a number of change detection tools commercially available in the market. Here the change detection algorithm XREL_Change_SQL [Sundaram.2005] is used for detecting changes in two XML documents, one is the original XML document and the other is the one on which the changes were imposed. Even though the algorithm provides the changes in XML documents its not enough since the database has to be updated as per the changes in the XML. In order to achieve this, a series of rules are proposed which receives the changes from XREL_Change_SQL and converts those changes into SQL queries. These SQL queries are then applied to the database to update them. There are different changes possible on the database like Insert, Delete, Update, Move etc. Chapter 4 provides in-depth analysis of change detection technique and the rules to generate SQL queries.

The following section provides some previous work done in this area followed by some preliminaries to understand the research in detail. Section 3 provides the system implementation and algorithms used for creating XML views. Section 4 provides the rules for generating SQL queries to update the database based in changes in the XML view. Finally, Section 5 provides the conclusion and ideas for future work.

## 2. RELATED WORK AND BACKGROUND

A lot of work has been going on in the area of generating XML documents from relational databases [Carey et al.2000] [Fernandez et al.2000] [Microsoft Corporation] [Banerjee et al.2000]. XPERANTO [Carey et al.2000] uses XML-QL [Deutsch et al. 1999] and creates XQGM (XML Query Graph Model) - a language-neutral intermediate representation for XML queries which is later converted into SQL queries. SilkRoute [Fernandez et al.2000] uses SQL in combination with XML-QL; also it uses RXL a declarative query language to construct XML documents. Microsoft uses annotated XML Schema called XDR Schemas (XML-Data Reduced) [Microsoft Corporation]. One thing that is common in all theses approaches is that they all make use of some specific query translation language. The user is forced to learn and understand those languages. Unlike the methodologies mentioned above, the approach specified in this paper is to naturally extend SQL. XML/SQL a language proposed in [Vittori et al.2001] is purely focused on SQL; the method involves generating SQL queries similar to Nest-Project-Join expressions. The XML is generated based on queries and constructs, the query holds the data from SQL joins and the construct provides the XML specification for tagging. [Badia.] uses Non-First Normal Form (N$F^2$) relations [Ablteboul et al.1984] which insists on breaking the rule of the First-Normal-Form (1NF); also it is the only paper that motivates all the work to be done automatically. The (N$F^2$) relations are essential for generating the XML documents once the SQL queries are executed. The concept of (N$F^2$) is used for the implementation in [Badia.], but unlike (N$F^2$) systems [Scholl et al.1987] [Dadam et al.1986] the focus in this paper is not to store the tuples and query them later, instead an efficient run-time publishing and tagging of XML documents is implemented.

On the other hand updating relational database through XML views has not received much attention. [Carey et al.2000] [Bohannon et al.2002] [Shanmugasundaram et al.2001] explore the idea of building XML views over relational databases and use existing XML query language to query those views. The goal here is not query the view instead update the view. Commercial database like DB2 which allows the updates requires the user to issue the updates directly to the database and not through XML views. Other commercial XML databases like XIndice [Apache.2002] and Tamino [Software AG. 2002.] also are related to updating XML views but the source is not a relational database. In [Tatarinov et al.2001] XML is updated using XQuery ignoring the use of relational database i.e. its updates on pure XML, the approach in this paper is to publish XML from relational database and maintain the updates. [Dayal et al.1982] proposes theorems to translate XQuery to SQL, to map the updates on XML to the updates made on the relational database. [Masunaga.1984] provides the design for a view update translator and a set of translation rules to convert the updates on XML onto relational databases. The design is in the form of a tree where the root is the view and the leaves are relational data. The update propagates through the leaves from the root. But there are ambiguity issues which are not taken care of by them. The most recent work published [Braganholo et al.2006] addresses the problem with a framework called PATAXO; they use UXQuery [Braganholo et al.2003] as a XML view definition language. Unlike all these approaches mentioned above the approach in this paper is believed to be the first in addressing the issue of updating relational data through XML views using XML change detection tools.

This next section provides some background knowledge about XML, XML Schema, Nested Relational Algebra etc required to understand things further.

### 2.1 XML

XML is a subset of Standard Generalized Markup Language (SGML) it is a data model to represent and store semi-structured data on the web and for data exchange. Since semi-structured data (or tree-based or nested data) is complex and can have many levels of nesting there is a need to come up with a data model that could store such data; XML's hierarchical nature made it suitable for this purpose. XML represents data by providing start and end tags which provide some semantics to the data it holds in between those tags. An XML document comprises of a root under which several elements are listed as a tree. Each element has a start and end tag and could be either simple or complex. Elements inside elements should be properly nested. A simple element does not have any child i.e. it ends the path from the root to that element, whereas a complex element can hold other simple elements as well as complex elements. Unlike HTML whose tags are predefined XML is pretty flexible and allows the user to specify own structure by providing user-defined tags. A simple example of a XML document containing books and their prices is given in Figure 1.

### 2.2 XML Schema

The XML Schema is the actual specification or the framework which sets the rules as to how the XML can be structured. Each XML document has its own schema and an XML document is said to be well formed if and only if it conforms to the specifications of its XML Schema. Two most general models of specifying the XML schema are the DTD (Document Type Definition) and XSD (XML Schema Definition). In this paper we make use of the XSD due to some of its benefits over DTD. For example in XSD one can specify the element type whether it's a string, integer etc and also its possible to provide names for complex types which can be reused anywhere in the schema. Declarations in XSD can have more complex internal structures than DTD; also one can add extra information in XSD when required. The data types in XSD are more data-oriented than document-oriented as in DTD. In addition to elements there are attributes which define certain constraints on the data. The elements and attributes can have their own values, the attributes are declared within the start tag while the element values occur in between the start and end tags.

An important aspect to know is the cardinality. Regular Expressions like ?, *, +, | can be expressed in XSD. The following table explains the regular expressions and the corresponding XSD representation, | represents choice which enables to choose either of two elements defined.

```
<Books>
  <Book>
    <Title> Introduction to XML </Title>
    <Author>
          <Firstname> Damian </Firstname>
          <Lastname> Martyn </Lastname>
    </Author>
    <Price> 150 </Price>
  </Book>
  <Book>
    <Title> Theory of Relational Databases </Title>
    <Author>
          <Firstname> Dimitry </Firstname>
          <Lastname> Ivanowsky </Lastname>
    </Author>
    <Price> 120 </Price>
  </Book>
</Books>
```

Figure. 2: A simple XML document

Table I: Regular expressions and their equivalent XSD representation

| Regular Expression | Definition | XSD Representation |
|---|---|---|
| $*$ | 0 or more occurrences | Minoccurs = 0, Maxoccurs = any value > 0 |
| $+$ | 1 or more occurrences | Minoccurs = 1, Maxoccurs = any value > 0 |
| ? | 0 or 1 occurrence | Minoccurs = 0, Maxoccurs = 1 |

The default value for minoccurs and maxoccurs is 1. Apart from these element types can have arbitrary atomic domains which helps in specifying keys for an element. For example an ID attribute can be used to give a unique value to that element. Similarly foreign keys are also possible, an IDREF referring another unique ID, list and union types can be mentioned as well. The XSD for the XML document is given in Figure 3.

## 2.3 XML Views From Relational Database

The XML view over relations is nothing but a snap shot of the data present in the database whose structure is specified by the XML document. Consider a sample relational database of Items and their price along with their quantity available. The relations are described as follows:

**Items**

| Id | Item |
|---|---|
| 1 | Keyboard |
| 2 | Mouse |

**Item_Details**

| Price | Quantity | ItemId |
|---|---|---|
| 50 | 50 | 1 |
| 15 | 80 | 2 |

Given a target XML schema, the data can be represented in one way in XML view as:

```
<xsd: element name="book" type="booktype"/>
          <xsd:complexType name="booktype">
                    <xsd:sequence>
                    <xsd:element name="Title" type="xsd:string"/>
                    <xsd:element name="author" type="authorname"/>
                              <xsd:complexType name = "authorname">
                              <xsd:sequence>
                                        <xsd:element name="Firstname" type="xsd:string"/>
                                        <xsd:element name="Lastname" type=""xsd:string"/>
                              </xsd:sequence>
                              </xsd:complexType>
                    </xsd:element>
                    <xsd:element name = "Price" type = "xsd:float" minoccurs = '0'
                                   maxoccurs = '300'/>
                    </xsd:sequence>
          <\xsd:complexType>
</xsd:element>
```

Figure. 3: XSD for the XML document in Figure 2

```
<Items>
          <Item> Keyboard </Item>
                    <Source>
                              <Price> 50 </Price>
                              <Quantity> 50 </Quantity>
                    </Source>
          <Item> Mouse </Item>
                    <Source>
                              <Price> 15 </Price>
                              <Quantity> 80 </Quantity>
                    </Source>
</Items>
```

Figure. 4: XML view for relations in Table 2

## 2.4 Nested Relational Algebra

A Nested Relational Algebra violates the $1^{st}$ normal form (1NF) where an attribute can have multiple values. Those attributes are called nested attributes or in other words multi-valued composite attributes. The multi-valued attributes helps to form a hierarchical structure similar to XML. This property is the key to obtain XML documents from relational databases. In order to explain the concept of nested relational algebra consider the following relation: Department (IdDept, Dname, Courses) Courses (Coursename, Profname) Suppose these two relations were to be nested based on the attribute IdDept the nest operator ($\nu$) can be expressed as:

Department$_{Nested}$ $\leftarrow (\nu)_{IdDept,Dname}$ (Department)

The resulting Department$_{Nested}$ table would form a multi-valued attribute of Courses based on the IdDept and Dname attributes. All the attributes which are not mentioned in the nest operator are converted into multi-valued attributes. Department$_{Nested}$ with some data might look like:

Table II: Nested Relation

| **IdDept** | **DName** | **Courses** | |
|---|---|---|---|
| 1 | Mathematics | **CouseName** | **Profname** |
| | | Algebra | John Reed |
| | | Logarithms | Ronald hopkins |
| 2 | Physics | **CouseName** | **Profname** |
| | | Mechanics | Jessica Adre |
| | | Thermodynamics | Julius Mclean |

In this paper Nest-Project-Join (NPJ) operations are used, which is obtained by first joining tables using a Join operator (X) followed by Project operator($\Pi$) which selects certain attributes from the result of the join operation and then applying the Nest operator to form a multi-valued composite structure like shown in Table 3.

## 3. XML DOCUMENT CONSTRUCTION

In order to publish the XML document from relational database an effective translation technique is required to transform the data in the Relational database to XML. In this Chapter Algorithms to achieve this goal is described. The formation of XML document is done through various steps and each step by itself is an individual module which leads to the final view.

The basic assumption in creating XML views is that the schema of the XML document and the underlying database schema are provided. The database is the source and the target is an XML view. In addition to some of the XML Schema attributes described in Section 2.2, the attribute called source is introduced in the schema which is used in this algorithm. The source is nothing but the name of the column in the database to which the element is mapped. The following are the XML and the Database schema of an example involving the Department, the Courses offered there and the Professors who offer them.

The source attribute provided in the schema is the actual name of the columns specified in the tables in the underlying database whose schema can be represented in Figure 6. A sample XML document for the Schema in Figure 5 is given in Figure 7.

To describe the process of creating a view, three different XML schemas were taken to produce three different XML views while the database schema remains the same. The beauty of the application is that the schema can be varied to produce the desired XML view. Each step below demonstrates how the three schemas are handled.

### 3.1 System Architecture

In this section, a detailed architecture of the implementation is described. It describes how the model proposed is actually carried out by the system. The following is a diagrammatic representation of the overall system architecture. In the diagram, the contents inside the dotted rectangle represent the system which is not visible to the user. The only inputs fed to the system are the XML Schema and the Database Schema. The XML Schema is first parsed by the module TL Generator to produce the Target Lists which is then provided to the module NPJ Generator. The TL Generator module comprises of the algorithm described in Figure 9 of Section 3.1.1. The Database Schema on the other hand is passed to the Database directly to get additional information about the source and the column mapping. Also the database provides the relational information of different tables like the primary key, foreign key and attributes which are important

```
<xsd:element name="Departments" source="null" type="Deptstype">
 <xsd:complexType name="Deptstype">
  <xsd:sequence>
   <xsd:element name="Department" type="depttype">
    <xsd:complexType name="depttype">
     <xsd:sequence>
      <xsd:element name="IdDept" source="IdDept" minOccurs="0" />
      <xsd:element name="DeptName" source="DeptName" minOccurs="0" />
      <xsd:element name="Courses" type="coursetype" />
       <xsd:complexType name="coursetype">
        <xsd:sequence>
         <xsd:element name="Course" source="CourseName" minOccurs="0" />
         <xsd:element name = "Professors" type = "proftype" />
          <xsd:complexType name = "proftype">
           <xsd:sequence>
              <xsd:element name = "Professor" source = "Profname" minoccurs="0" />
           </xsd:sequence>
          </xsd:complexType>
        </xsd:sequence>
       </xsd:complexType>
     </xsd:sequence>
    </xsd:complexType>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

Figure. 5: Sample XML Schema

to generate the SQL query. Once the table details are passed to the NPJ Generator it creates the SQL queries which are in accordance with the NPJ Expressions.

The queries are directly applied on the database and the resulting Non-First Normal Form nested view is given as input to the XML Document Constructor which is nothing but the implementation of the algorithm in Figure 13 of Section 3.1.4. The output is a view of the database expressed as XML.

The database used in this implementation is Oracle 9i, all the modules where developed as individual Java classes and the user interface was developed using Applets. The following sections explain the algorithms implemented in the system.

3.1.1 ***Create Target List.*** The system environment is assumed to be a middle scale MANET in which a few dozens of mobile nodes retrieve data items held by themselves by using a top-k query. The detail of the system model is as follows:

A Target List can be simply defined as a collection of (source, level, label). The source corresponds to the column name of the XML element stored in the database and level indicates the depth of the element in the XML document. Level can be understood by imagining the whole XML Schema as a tree structure; the distance is calculated from the topmost parent, incrementing by 1 every time the path ends in an element. The three different schemas differ in levels; they are of level 1, level 2 and level 3. Finally the label suggests whether the element is 'required' or 'optional'. The following is the algorithm for generating Target Lists. In order to create the Tar-

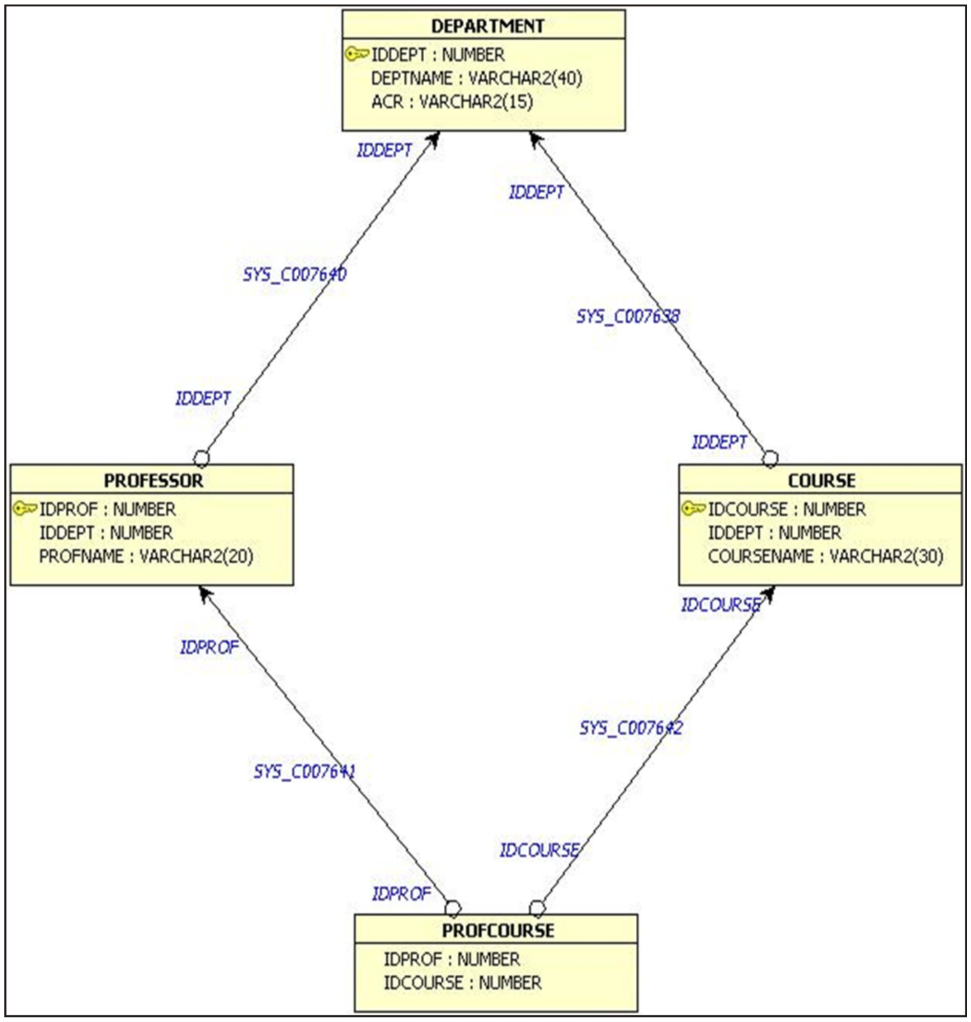


Figure. 6: Database Schema

get List the whole schema is parsed and the elements are captured along with their depth in the XML tree. The minimum occurrence of each element obtained can be used to determine whether an element has to be present or excluded in the final document generated. Sample Target Lists along with their XML schema for level 1, level 2 and level 3 can be shown in Figure 10.

TL3. (IdDept 1 required) (DeptName 1 required) (CourseName 2 required) (ProfName 3 optional)

3.1.2 ***Create NPJ Expression.*** The Target List generated above needs to be converted into a form which the relational database can understand. Here the relational algebra is used to represent the Target List information as algebraic expressions which can be converted into SQL queries for querying the database. Depending on whether the element is required or optional the join operation could be Outer or Inner respectively.

The NPJ expression is created with the help of the Target List by joining the tables which correspond to the Target List of the deepest level with the next higher level Target List in an iterative manner. For example, if there is a Target List with the deepest level as 4 then the table which holds the source of the Target List is joined with level 3 and a temporary NPJ expression is formed. This temporary NPJ expression is again joined with level 2 and again a temporary expression is formed which is again joined with level 1. Since there are no more levels below 1

```
<Departments>
        <Department>
                <Dept>1</Dept>
                <DeptName> Computer Science </DeptName>
                        <Courses>
                                <Course> Database Systems </Course>
                                        <Professors>
                                                <Professor> Mike Preston </Professor>
                                                <Professor> Randy Jackson </Professor>
                                        </Professors>
                                <Course> Automata Theory </Course>
                                        <Professors>
                                                <Professor> James Anderson </Professor>
                                        </Professors>
                        </Courses>
        </Department>
        <Department>
                <Dept>2</Dept>
                <DeptName> Electrical Engineering </DeptName>
                        <Courses>
                                <Course> Circuit Design </Course>
                                        <Professors>
                                                <Professor> Richard Antony </Professor>
                                        </Professors>
                        </Courses>
        </Department>
</Departments>
```

Figure. 7: Sample XML Document

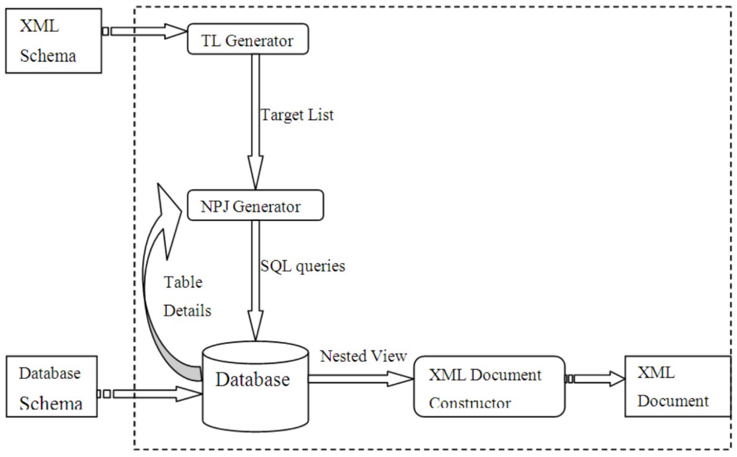


Figure. 8: System Architecture

the final NPJ expression is obtained. The attributes which are selected in each join are crucial since they might be needed for the next higher level join.

The problem of selecting the attributes in different level becomes easy since the XML schema already has the information on what attributes are needed to form the View. There is no need for joining tables in level 1 schema since there is only one table involved. The following shows the NPJ expression along with their equivalent SQL queries for level 2 and 3. The symbols used are from standard relational algebra.

***NPJ Expression***

Consider the level 2 Target List TL2, the first level attributes (IdDept and DeptName) are the

```
INPUT: XML Schema
OUTPUT: Target List TL
ASSUMPTION: XML schema is well formed, each element has a source name in the database.
TL = Empty;
LIFO = Empty;
Level = 0;
While XML Schema not Empty do
        if (complex type of element &T is found)
        level = level +1;
        add &T to LIFO;
        else if (simple type &T is found)
        label = if (MinOccurs = 0) ``required"; else ``optional";
        TL = add (source name of &T, level, label) to TL;
        add &T to LIFO;
else if (closing tag is found) pop corresponding matching tag &T
      off LIFO
End While
If (LIFO is empty) return TL
```

Figure. 9: Algorithm for generating Target List

```
<xsd:element name="Departments" type="Deptstype">
<xsd:complexType name="Deptstype">
<xsd:sequence>
<xsd:element name="Department" type="depttype">
<xsd:complexType name="depttype">
<xsd:sequence>
<xsd:element name="Dept" source="IdDept" minOccurs="0" />
<xsd:element name="DeptName" source="DeptName" minOccurs="0" />
<xsd:element name="Deptacr" source="acr" minOccurs="0" />
</xsd:sequence>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Figure. 10: Level 1 XML Schema

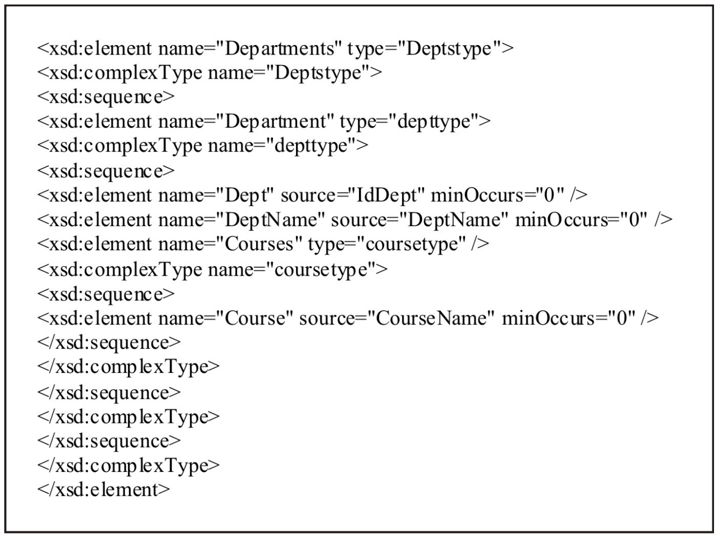
columns in the table Department and second level attribute (CourseName) is the column in the table Course. To obtain the NPJ expression for this Target List the two tables are first joined based on the key. In this case IdDept is the primary key in Department and foreign key in Course table. The attributes or columns which are necessary to form the XML are projected and finally nested based on the lower level (1) attributes (IdDept and DeptName). The NPJ expression for the Target List can be shown as:

TL2.$\nu_{IdDept,Deptname}(\Pi_{IdDept,DeptName,CourseName}(DepartmentX_{IdDept}Course))$

**Equivalent SQL statement:**

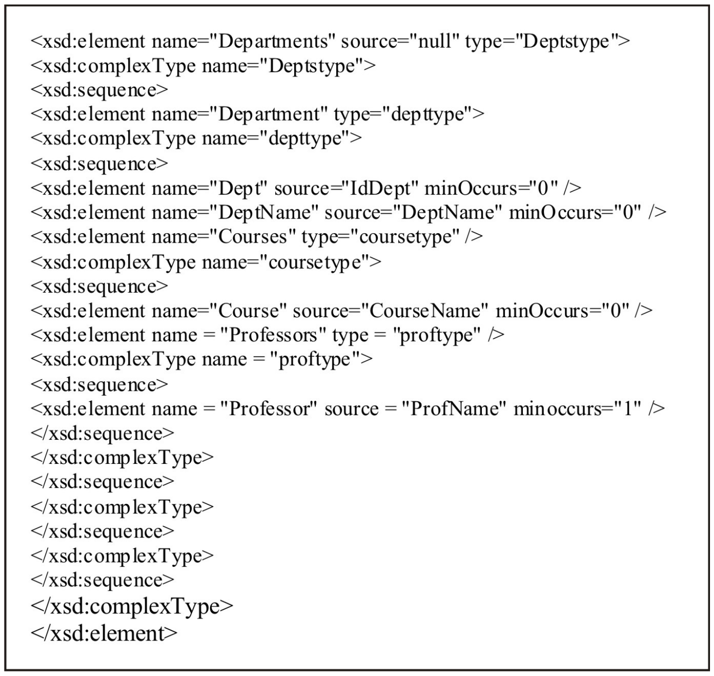
*Select Department.IdDept, Department.DeptName, Course.CourseName from Department, Course where Deprtmant.IdDept = Course.IdDept(+)*

Since the Target List1 indicates that the elements (Department) are required whether they



```
<xsd:element name="Departments" type="Deptstype">
<xsd:complexType name="Deptstype">
<xsd:sequence>
<xsd:element name="Department" type="depttype">
<xsd:complexType name="depttype">
<xsd:sequence>
<xsd:element name="Dept" source="IdDept" minOccurs="0" />
<xsd:element name="DeptName" source="DeptName" minOccurs="0" />
<xsd:element name="Courses" type="coursetype" />
<xsd:complexType name="coursetype">
<xsd:sequence>
<xsd:element name="Course" source="CourseName" minOccurs="0" />
</xsd:sequence>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Figure. 11: Level 2 XML Schema



```
<xsd:element name="Departments" source="null" type="Deptstype">
<xsd:complexType name="Deptstype">
<xsd:sequence>
<xsd:element name="Department" type="depttype">
<xsd:complexType name="depttype">
<xsd:sequence>
<xsd:element name="Dept" source="IdDept" minOccurs="0" />
<xsd:element name="DeptName" source="DeptName" minOccurs="0" />
<xsd:element name="Courses" type="coursetype" />
<xsd:complexType name="coursetype">
<xsd:sequence>
<xsd:element name="Course" source="CourseName" minOccurs="0" />
<xsd:element name = "Professors" type = "proftype" />
<xsd:complexType name = "proftype">
<xsd:sequence>
<xsd:element name = "Professor" source = "ProfName" minoccurs="1" />
</xsd:sequence>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Figure. 12: Level 3 XML Schema

have empty courses or not, Outer Join is used in this case so that all departments are listed even if they do not contain any course.

For the Target List of level 3 TL3 a recursive algorithm is applied to join the tables. Suppose the deepest level is denoted as ‘i’ the tables to which the attributes of level i belong are joined with the table of level (i-1) projected, nested as it is shown above for TL2 and the result is captured in a Temp view. Now the Temp view is joined with the next lower level (i-2) and again the results are stored as Temp view. The results propagate to level 1 by consecutive joins and the final NPJ Expression is obtained. For TL3 the NPJ can be shown as:

TL3. Temp = $\nu_{IdDept,CourseName}(\Pi_{IdDept,CourseName,ProfName}$
$(CourseX_{IdCourse}ProfCourseX_{IdProf}Professor))$

Nested Expression = $\nu_{IdDept,DeptName}(\Pi_{DeptName,Temp}(DepartmentX_{IdDept}Temp))$ Since the Course and the Professor table are linked via ProfCourse, the table ProfCourse is used to join the two tables.

**Equivalent SQL statement:**
*Select Department.IdDept, Department.DeptName, Course.CourseName, Professor.ProfName from Department, Course, Professor where Department.IdDept = Course.IdDept(+) and Course.IdCourse = ProfCourse.IdCourse and Professor.IdProf = ProfCourse.IdProf.*
From the above statement it is clear that the Outer join is used for the Department whereas the inner join/simple join is used for the Course since the ProfName is optional under the courses defined in the Target List 2.

3.1.3 ***Create Nested View.*** The Nested View is obtained by the execution of SQL query created from the NPJ expression. The query is applied directly to the database and the temporary view for query TL2 looks like the one in the following table.

Table III: Sample Nested Table/View

| **IdDept** | **DeptName** | **Id Course** | **CourseName** |
|---|---|---|---|
| 1 | Computer Science | 1 | Database |
| 1 | Computer Science | 2 | Compilers |
| 2 | Philosophy | 3 | Philosophy |

In the table above we can see that the First Normal form is not followed which is the key to obtain the nested view. For the ease of understanding the views created by the TL2 and TL3 NPJ expression can be represented graphically as:

Table IV: Nested View for Target Lists 2 & 3

**TL2.**

| **IdDept** | **DeptName** | **Course** | |
|---|---|---|---|
| 1 | Computer Science | **IdCouse** | **CourseName** |
| | | 1 | DataBase |
| | | 2 | Compilers |
| 2 | Philosophy | **IdCouse** | **CourseName** |
| | | 3 | Philosophy |

**TL3.**

| **IdDept** | **DeptName** | **Course** | | |
|---|---|---|---|---|
| 1 | Computer Science | **IdCouse** | **CourseName** | **IdProf** |
| | | 1 | Database | 123 |
| | | 2 | Compilers | 321 |

3.1.4 ***Create XML View.*** Once the view is created the XML document can be constructed from each tuple in the table. In the nested view there are simple and complex elements, in the Nested View1 the IdDept and the DeptName are simple whereas the IdCourse and CourseName is complex and the straight forward approach is to declare them as complex types in the XML document and they are in the deepest level. The IdDept and DeptName are at level 1 since they are simple attributes in the table. Thus the document is constructed by iteratively querying the database for level 1 to the deepest level with the help of the XML schema specification obtained in the Target List. Each tuple in the table forms a single XML document segment. Like wise all the tuples are retrieved to form the complete XML document.

The algorithm described above might be complicated as shown but in practice the code is quite understandable, reusable and flexible. The basic working of the algorithm is to start from the 1st level and display all the elements till it reaches the deepest level. The algorithm is recursive which can be used again and again for any level. The XML document is generated in such a way that first the element of level 1 is printed with opening and closing XML tags, the value for that element is obtained from the column in the table which is the source for the element. The source name is basically provided by the Target List. For example the List (DeptName 1 optional Department) (CourseName 2 Optional Courses) is displayed starting from the element DeptName, but since its also necessary to display the complex element's name under which the sub elements of that level are displayed an additional field in the Target List called ComplexElement name or simply the parent of a particular level or in Database terms the relation (table) to which the source of the sub elements belong. For level 1 the parent is Department (ComplexElement) under which the DeptNames could be displayed and for level 2 it is Courses.

```
INPUT: Temp Table/View, Target List.
OUTPUT: XML View.
ASSUMPTION: For level 1 the values of PCV (CT) and PCV (NT) are set to be different.
Level = 1;
Pointer = 0 (Tuple number);
LIFO = Empty;
Max = Max Level of XML Schema;
While (Table not Empty)
Create_XML (Level) {
        Print <&Relation Name> of the source in the Target List for current level;
        Push relation name in LIFO;
        Pointer = Pointer + 1;
                        While (PCV (CT) not equals PCV (NT))
Print E(C, V);
If (Level less than Max) {
Create_XML (Level + 1);
Pointer = Pointer – 1;}
Pointer = Pointer + 1;
End While
Print </&Relation Name> off from LIFO;
        If (Level not equals 1) {
                Level = level – 1;
                Create_XML (Level);}}
End While
```

Figure. 13: Algorithm for generating XML view.

The algorithm starts printing $< Department >$ under which the Element is printed with the name obtained from the XML Schema and the Value obtained from the corresponding table to which the source belongs in the database; in short E(C,V). The E(C, V) for DeptName looks like $< DeptName > \&Value < /DeptName >$. The &Value is retrieved from the table Department under the column of source name DeptName. Once it is printed the algorithm looks for level 2 which means it checks whether there are any courses under that DeptName, if yes then it is printed in a similar fashion as of level 1 under the DeptName tag starting with a Tag for the Parent which for level 2 is Courses. Since there is nothing under the courses as it is the deepest level of the Target List the courses for the particular DeptName are displayed one by one and the closing parent tag for Courses is printed finally. Since the first Dept and their courses are displayed the next DeptName is printed along with the courses and the whole process is repeated in an iterative and recursive way until there are no more tuples. For any level these basic steps are carried out and the same algorithm is used.

Table V: Sample Nested Table/View

| **IdDept** | **DeptName** | **CourseName** | **ProfName** |
|---|---|---|---|
| 1 | Computer Science | Database Systems | Mike Preston |
| 1 | Computer Science | Database Systems | Randy Jackson |
| 1 | Computer Science | Automata Theory | James Anderson |
| 2 | Electrical Engineering | Circuit Design | Richard Antony |

For example consider the sample Nested View given in Table 6. When the Algorithm in Figure 13 is applied to the view in Table 6, the resulting XML document will look like the one given in Figure 7.

## 3.2 Comparison Of Models

There are two models which are closely related to solving the problem of XML view maintenance with the help of relational database and most importantly Nested joins. The first model 1 [Badia.] considers solving the problem by creating Target Lists, using nested joins and creating a table which looks like the view in Table 5 from which the XML document is created. The second model 2 [Vittori et al.2001] which applies joins does not consider Target Lists and table creation; instead the approach is to have a query and a construct. The query is comprised of several join operations which provides the values from the database directly. The construct defines the layout or the skeleton of the XML document and gets the values from the join operation to provide the data to fill in the document.

These two models were tested for their performance with the help of the application developed. The tests were run for different number of tuples and for three different schemas varying in their levels as 1, 2 and 3. There were three tables involved; the total number of tuples is obtained as a result of joining those tables in the database. Since only one table is involved in level 1 Schema there was no significant difference between the two models in their joins, hence it is excluded. The database schema is similar to the one shown in Figure 6.

The database was populated by injecting values using automated code. The three schemas shown in Figure 10, 11 & 12 were used as input. The following results shown in Figures 15 and 16 were obtained for level 2 and level 3 Schema for both the models.

## 4. UPDATING RELATIONAL DATABASE THROUGH XML VIEWS

In this section, a method for propagating the changes from the XML view document to the relational database is proposed. In this scenario an XML document generated by the system in Section 3 is presented to the user who modifies by inserting, deleting, and updating some parts of the document. The document is sent back by the user to update the relations from which the XML was obtained. The architecture of the model is to execute the whole process automatically. In order to detect the changes made on the XML document the original document sent is compared with the returned document and the changes are captured with the help of XREL_Change_SQL a change detection technique [Sundaram et al.2005]. Once the changes are obtained from the change detection tool, certain rules are applied to the changes to generate the SQL queries which update the database. There are different kinds of modifications possible like insert, delete, update, move etc. which will be discussed in detail in the forthcoming subsections. First we present the outline of the model here.

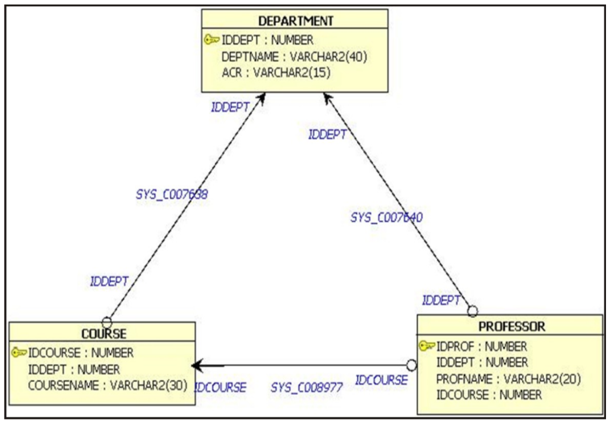


Figure. 14: Sample Database Schema

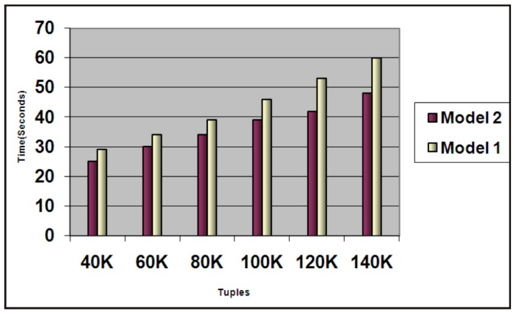


Figure. 15: Performance chart for level 2 XML Schema

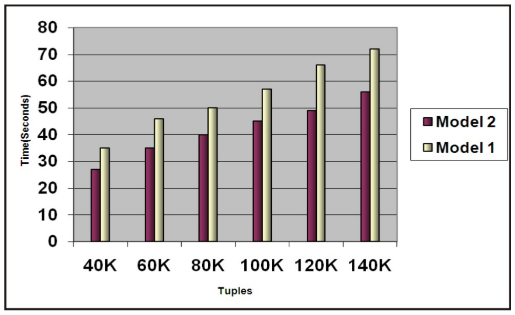


Figure. 16: Performance chart for level 3 XML Schema

## 4.1 XREL_CHANGE_SQL

In this section a brief introduction of the change detection tool XREL_Change_SQL is discussed which is necessary for updating the relational database. The tool uses a modified version of XREL [Yoshikawa et al.2001] a method of storing and retrieving XML documents using relational databases. Hence there is no need for any extensions to the existing database. In XREL_Change_SQL the whole XML document is considered as a tree structure and is split into various nodes (elements) based on the path of the node from the root. Each node is identified uniquely and is stored in relational database.

4.1.1 ***XREL_Change_SQL database schema.*** There are five tables described by the tool which store the XML documents:
*Element(docID, pathID, start, end, index, reindex, parentID, pstart)*

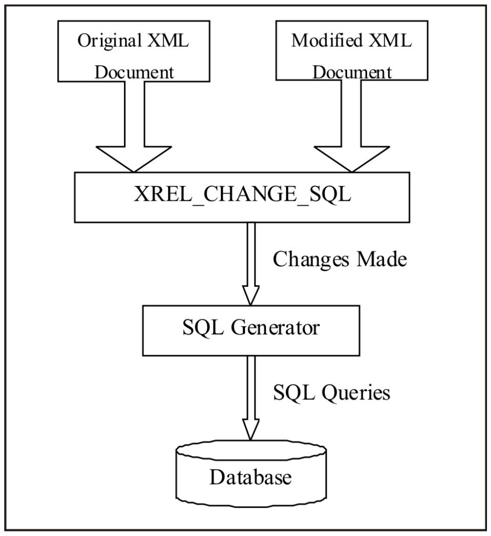


Figure. 17: System model for updating relational database

*Attribute(docID, pathID, start, end, value)*
*Text(docID, pathID, start, end, value, parentID, pstart)*
*Path(pathID, pathexp)*
*Document(docID, document*)
Where
*docID* - A unique ID for an XML documents
*pathID* - ID for a particular path expression
*start* - start position of a region
*end* - ending position of a region
*index, reindex* - represent occurrence order of an element among other similar elements of its parent
*parentID* - Unique value for the parent of a node
*pstart* - start postion of the parent region of an element
*value* - actual data present in a node
*pathID* - Unique value for a path expression
*pathexp* - path of an element from its root node
*document* - stores the entire document as a large structure

A lot of the concepts in the model are beyond the scope of this paper; hence the focus is only on the changes provided by the change detection tool and not on how they are detected. Some of the concepts which are essential for generating appropriate SQL queries for updating the database are discussed below, before that a tree representation of the XML document used in Figure 2 is given:

✦ *pathexp*: To understand the concept of pathexp consider Figure 21, the pathexp for the element or node Firstname can be represented as #/Books#/Book#/Author#/Firstname. There are pathexp for all the elements in the XML document from the root and each one of them is given a unique ID.

✦ *start, end*: The start and end attributes in the database represent the region of a particular element. For example in Figure 21 the element Author's region comprises of the Firstname and Lastname elements and the Book element's region comprises of the sub elements Title, Author and Price. The region is defined for each and every occurrence of the element and hence a region can be defined unique for that particular element. The start and end values of

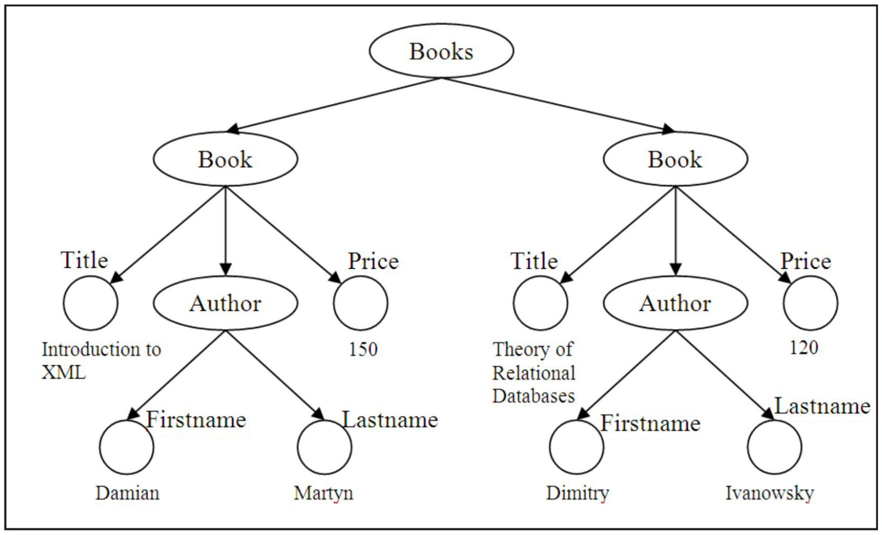


Figure. 18: Tree representation of XML document in Figure 2

an element are calculated as the number of bytes from the beginning of the document to the element and the number of bytes from the beginning of the document to the end of the element respectively.

✦ *parentID*: Each element is considered as a parent to its sub elements and each sub element is uniquely identified using the parentID its parent. All the ID's in the XREL_Change_SQL database schema are auto-generated.

The attribute and text table store the values of the attributes and elements respectively along with the pathID, parentID and the region, thus it becomes easy to find a particular value in the XML document using these attributes. The value of the pathexp #/Books#/Book#/Author#/Firstname for the XML document in Figure 16 could be either 'Damian' or 'Dimitry', but since the *parentID*, the *region* and the *index* uniquely identify an element it becomes easy to obtain the value for the particular element using the same *pathexp*. For example if the Firstname element for the title 'Introduction to XML' has a $parentID = 12$ and the region is start $= 15$, $end = 35$ then the value of it is uniquely identified as 'Damian'. In similar way the other Firstname element values can also be found with the help of the unique parentID and the region information.

4.1.2 ***Matching sub-trees.*** The change detection tool performs some pre-processing on the two version of the XML documents to detect the changes. The detection of changes is done by detecting the matching sub-trees. There are different kinds of matching done by the tool like matching *leaf node parent, pure sub-trees and complex sub-trees.* A *leaf node parent* can be described as a node whose children contain only text and attributes like the node Author in Figure 21. A *pure sub-tree* is described as the node which contains only sub-trees as its children and no children containing text or attributes like the node Books in Figure 21. A complex sub-tree can be defined as a node which is a combination of children containing text, attributes and sub-trees like the node Book in Figure 21.

The matching is done based on the pathexp of two nodes and their values of their children. Let T1 and T2 be two leaf node parents from the original and modified document and c1 and c2 be their children respectively. A match is obtained if pathID(T1) = pathID(T2) and value(c1) = value(c2). There are two types of matching, one is to find the leaf node parents with unique attributes as their children and the other is to find the matching for leaf node parents which do not contain any single unique attribute which identifies them. In the former case the matching becomes easy since the unique attribute is sufficient enough to identify the leaf node parents, but in the later since not a single attribute can identify the nodes a combination of attributes and/or text elements is decided as the key and the perfect match is found.

## 4.2 Updating Database

In this section, we describe in detail of what kind of SQL queries need to be generated to update the database based on the changes provided by the change detection tool. The change detection tool only compares two XML documents and provides the changes, thus data is manipulated in the XML level. Suppose a node was inserted the change detection tool just detects it, but in database terms the node might have already been present in the relational data and just was assigned to another parent in addition to the existing parent. In this paper the focus is in updating the database; and the concept of relations and the XML document is very much inter-related. The idea is how to translate the changes provided by the tool in the XML level into relational database level. There are different kinds of changes reported by the tool which are,

- leaf node insert
- leaf node delete
- leaf node update
- sub-tree insert
- sub-tree delete
- sub-tree move

The aim here is to convert these changes into SQL queries which can update the relations in the database. First, consider an XML document represented as a tree shown in Figure 22 which is generated using the algorithms in Chapter 4 based on the database schema in Figure 6. In this example the nodes are provided with values from the tables in Table 7. This example is used till the end for explaining the different kinds of changes made.

The complex sub-tree parents are associated with an ID attribute which is given inside the bubble of a node due to space constraints. The ID attribute is nothing but the primary key of that node in the relational table. The ID attribute helps in finding the relation between two tables and even the ancestor information. The ID attribute can also be related to the *parentID* in the XREL_Change_SQL database schema. The ID attribute is supplied to the XML when it is generated and is provided to the user, the changes may not necessarily contain the attribute ID. An interesting thing to note in the XML tree is that all the complex nodes have the names of the table whose columns are their children with the value. An assumption made here is that a Course, Professor cannot be under two Departments and a link table is not provided between them.

**Data Preprocessing:** Before getting into the database modification operations, some pre-processing needs to be done which is common to all kinds of operations. First the node which is inserted or deleted or moved is processed to obtain the parent and ancestor information. For example the *parentID* of all the complex nodes have to be extracted since this information is useful for updating all the relations to which the node is associated. The *parentID* is generated by the tool which can be matched with the ID attribute which was provided mandatory for every node. Only complex nodes have ID attribute associated with them, thus the rest of the pure sub-tree nodes are skipped. A modified nodes' related information has to be found from the root, a bottom-up approach is employed to get the ID's of the complex nodes along with their names. For example if the node ProfName containing the value 'Justin Heinz' is the one affected, then the path of the node is traversed backwards to obtain the ID of Course and then the Department from the path attribute and text table of the XREL_Change_SQL database. This information can be useful since the tool which detects the changes doesn't provide the relational details, for example the Professor table is not only associated with course but also with the Department

Table VI: Sample tables for relational database schema in Figure 6

**Department**

| IdDept | DeptName | ACR |
|---|---|---|
| 1 | Computer Science | C.S |
| 2 | Electrical Engineering | E.E |

**Course**

| IdCourse | IdDept | CourseName |
|---|---|---|
| 1 | 1 | Automata Theory |
| 2 | 1 | Relational Databases |
| 3 | 2 | Circuit Theory |

**Professor**

| IdProf | IdDept | ProfName |
|---|---|---|
| 1 | 1 | Justin Heinz |
| 2 | 1 | Steve Hanson |
| 3 | 1 | James Carroll |

**ProfCourse**

| IdProf | IdCourse |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |

table as found in Figure 6. In case of update operation the immediate parent information alone is enough since only the tuple is modified for values in the column.

Although different kinds of operation are mentioned by the change detection tool the operations involving sub-trees are the ones which will be useful. The reason being the modifications like insert, delete, move are not done at the leaf node level as this will alter the database schema, suppose a new node was introduced to the leaf node parent Professor of Figure 22, in relational database terms it's a new column added to the Professor table. The focus is only to modify the data. But update operations can be performed since they can happen even in the leaf node and do not alter the database schema. The different kinds of operations on the database are discussed in detail below:

4.2.1 ***Sub-tree insert.*** In general the insert operation is somewhat complicated compared to the rest. The main issue is how to know whether the node inserted is already present in the database. There might be cases where a professor who is associated with one course may be assigned to some other course as well. In such cases inserting the professor details again in the table causes unnecessary loss of space and sometimes ambiguity issues. Suppose a Professor is added to the course 'Relational Databases' in Figure 22 there are two cases possible:
1. *The data is in the table:* The tool provides the new inserted node along with path the *parentID, start, end* region and the *value* which was inserted. Here the concept of matching leaf node parents comes into picture. If the node inserted is already present somewhere in the original document the tool detects the match and provides the information that the newly inserted node is similar to the one that already exists. Suppose Professor 'Justin Heinz' is assigned to the Course 'Relational Databases' as well, the operation carried here is to modify the table Professor containing the value 'Justin Heinz' and other link tables which they are associated with, the SQL queries are provided in Table 8. With the help of Data Preprocessing the ID's of the parents and ancestors are obtained which is nothing but the ID's of Course and Department. In addition to the Course Professor 'Justin Heinz' is assigned to 'Relational Databases' as well. Thus the first operation would be to link the immediate parent namely Courses with the Professor.

The algorithm can be defined as follows:

I. Select the ID's of all parents which are foreign keys in table of the node inserted starting from the immediate to the root.

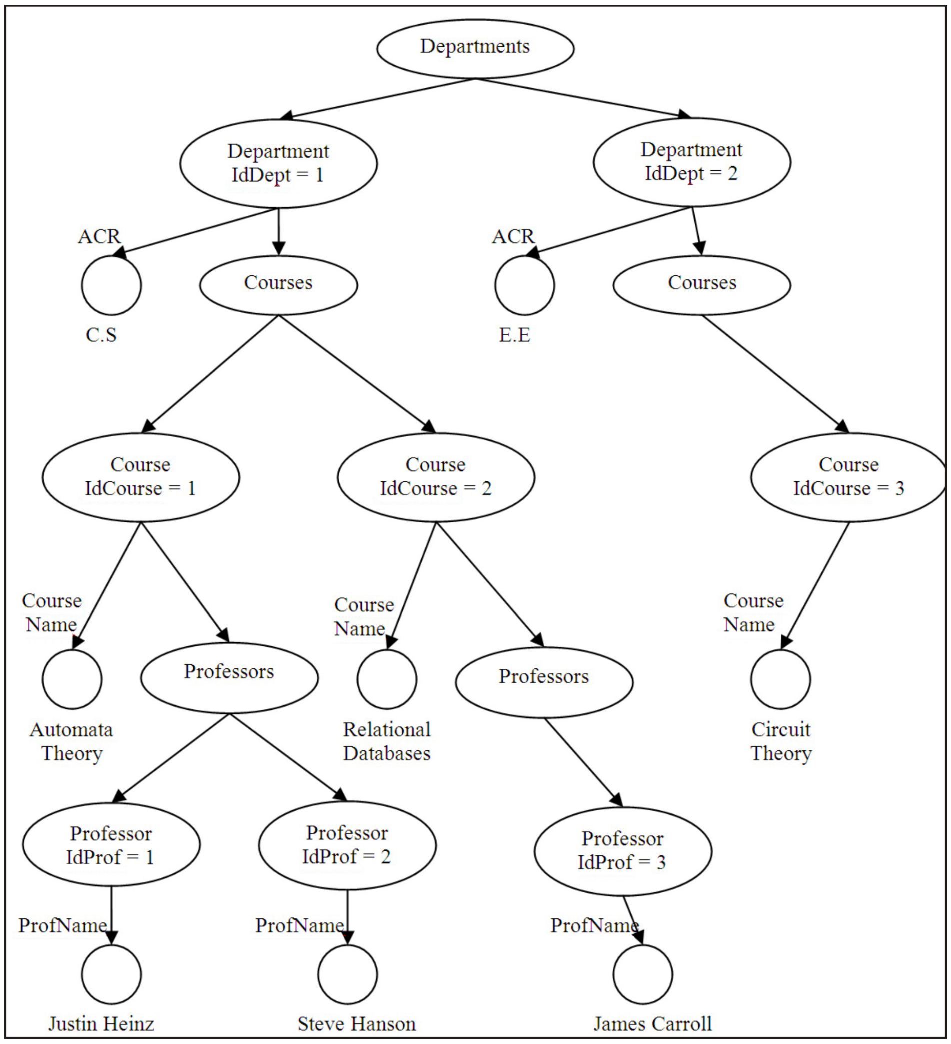


Figure. 19: XML tree representation of data in Table 7

II. If (Tuple exists with ID's as foreign keys and the value) Update link tables; Else insert a new row with the foreign keys obtained with the values; In this case there is the tuple in the Professor table containing the value 'Justin Heinz' is related to the Department with IdDept = 1 and hence there is no need for any changes in the Professor table, only the link table is modified by inserting a tuple relating the IdProf and IdCourse.

Table VII: Inserting existing node in table

| |
|---|
| INSERT INTO ProfCourse (IdProf, IdCourse) VALUES (1, 2); |

2. *Data is new*: Suppose a totally new Professor 'Oliver King' is added to the Course 'Circuit Theory' under the Department 'E.E', the operation is to insert the tuple in the relational table and in the link tables associated. The ID's are obtained through the Data Preprocessing step. The new values of the Professor node inserted are provided by the tool. First an insert operation is performed on the Professor table followed by updating the link tables if necessary, the operation is showed in Table 9.

Table VIII: Inserting a new node in table

```
INSERT INTO Professor (IdProf, IdDept, ProfName) VALUES (4, 2, 'Oliver King');
INSERT INTO ProfCourse (IdProf, IdCourse) VALUES (4, 3);
```

In case of more than one sub-tree in the sub-tree inserted the order is from top to down. The higher level sub-tree is inserted first using the algorithms above followed by the next.

4.2.2 ***Sub-Tree Delete.*** The delete operation is similar to insertion. The node deleted from the XML tree cannot be deleted immediately from the tables since the data might be related to a tuple in some other table. Hence if a node is deleted the matching sub-trees are found; if there is a matching sub-tree it means the data is related to some other table as well. Now there are two possibilities:

1. *Data is related to other table*: In this case the ID's of all the parents from the root are extracted in the Data Processing step. The ID of the node deleted is provided by the change detection tool along with the value which was provided mandatory. Suppose the Professor 'Steve Hanson' is deleted from the XML tree in Figure 22 and the Professor is related to some other course, the tuple in the link table ProfCourse is deleted which relates the Course and the Professor, the SQL query is provided in Table 10.

Table IX: Deleting data related to other tables

```
DELETE FROM ProfCourse (Idprof, IdCourse) VALUES (2, 2);
```

2.*Data is not related to any other table*: This case is very simple, if a node is deleted from the tree and is not related to any table the node in the relational table is deleted entirely based on the ID attribute of the deleted node provided by the tool.

In case of deleting a sub-tree containing more than one sub-tree in it parent deleted

4.2.3 ***Sub-Tree Move*** . A move is carried out by updating the foreign keys in the relational tables and does not have any insertions for any normal tables; the link tables might be updated. The change detection tool provides the ID of the parent to which the new node is moved and also the ID of the node itself. Thus it becomes easy to just update the foreign key from the old value to the new one. Suppose the Professor 'Justin Heinz' was moved from Course 'Automata theory' to 'Relational Databases' the link table ProfCourse is updated with the IdCourse of 'Relational Databases' against the Professor IdProf, the SQL queries are provided in Table 11. In case of more than one sub-tree top-down approach is used to update the foreign keys.

Table X: SQL query for move operation

```
UPDATE ProfCourse SET IdCourse = 2 WHERE IdProf = 1;
```

4.2.4 ***Leaf Node Update*** . The update is the only operation that can be done at the leaf node level since it does not change the relational schema. The update is relatively easier than the rest since the ID of its parent is sufficient enough to update the fields in the corresponding table. If the name of Professor 'Justin Heinz' is changed to 'Roger Goodwill' the Professor table is updated based on the IdProf, the SQL Query is provided in Table 12.

## 5. CONCLUSION AND FUTURE WORK

This paper provided a method to create XML documents from relational databases and similarly updates on XML data is translated to relational databases. In our approach, users have the freedom of specifying what data they need from the various tables in a single snap shot to generate XML views. Updating the relational database through XML views using change detection tools

Table XI: SQL Query for Leaf node update

```
UPDATE Professor SET ProfName = 'Roger Goodwill' WHERE IdProf = 1;
```

can be implemented to detect changes provided by the user and automatically convert the changes without the user having to know much about the underlying database. The proposal is also open to any change detection tool that can efficiently detect changes and is easy to plug-in and convert the rules into SQL queries.

## REFERENCES

A. Deutsch, M. Fernandez, D. Florescu,A. Levy and D. Suciu 1999. XML-QL: A Query Language for XML. In *Proceedings of the International World Wide Web (WWW) Conference*, Toronto, May 1999.

A. Gupta and I. S. Mumick, eds., 1999. Materialized Views: Techniques, Implementations and Applications, MIT Press, 1999.

Antonio Badia. Efficient Creation and Maintenance of XML Views on Relational Databases.

Apache Software Foundation. 2002. Apache Xindice. 2002. *http://xml.apache.org/xindice/.*

Bohannon, P., Ganguly, S. ,Korth, H. ,Narayan, P. and Shenoy, P. 2002. Optimizing view queries in ROLEX to support navigable result trees. In *VLDB*. Hong Kong, China.

Braganholo, Vanessa P., Davidson , Susan B., Heuser and Carlos A 2006. A Framework to Allow Updates Through XML Views. *ACM Transactions on Database Systems*, vol. 31, no. 3, pp. 839-886, Sept. 2006

Braganholo, V., Davidson,S. B. and Heuser, C. A 2003. UXQuery: building updatable XML views over relational databases. In *Simposio Brasileiro de Banco de Dados, SBBD. Belo Horizonte: Departamento de Ciencia da Computacao/UFMG*, Manaus, AM, Brasil, 26-40, 2003.

I. Tatarinov , Z. G. Ives , A. Y. Halevy and D. S. Weld 2001. Updating XML. In *Proc. of SIGMOD Conference*, 2001.

J. McHugh , S. Abiteboul,R. Goldman,D. Quass and Widom. 1997. Lore: A database management system for semistructured data. *SIGMOD Record*, 54 - 66, September 1997.

J. Shanmugasundaram, E. Shekita,R. Barr,M. Carey,B. Lindsay,H. Pirahesh and B. Reinwald 2000. Efficiently Publishing Relational Data as XML Documents, In *Proceedings of the VLDB Conference*, Egypt, September 2000.

M. Carey , D. Florescu ,Z. Ives,Y. Lu,J. Shanmugasundaram,E. Shekita, and S. subramanian 2000. XPERANTO: publishing object-relational data as XML. *Distributed and Parallel Databases 19(2-3).*,67-86.

M. F. Fernandez , W. C. Tan, and D. Suciu 2000. SilkRoute: Trading Between Relations and XML. *In 9th International World Wide Web Conference (WWW)*, May 2000.

Microsoft Corporation. Reducing network traffic in unstructured P2P systems using top-k queries. *http://msdn2.microsoft.com//ru-ru/library/ms172063.aspx*

M. Scholl , S. Abiteboul ,F. Bancilhon,N. Bidoit,S. Gamerman ,D. Plateau ,P. Richard and A. Verroust 1987. VERSO: A Database Machine Based On Nested Relations, *Nested Relations and Complex Objects*, Germany, April 1987.

P. Dadam , K. Kuespert,F. Andersen ,H. Blanken ,R. Erbe ,J. Guenauer ,V. Lum ,P. Pistor , and . Walch 1986. A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies, In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, Washington D.C., May 1986.

S Ablteboul and N Bldort Non First Normal Form Relatlons to Represent Hlerarchlcally Organized Data, In *Proceedings of Third ACM SIGACT-SIGMOD Sympoamm on Prmclples of Database Systems*, 1984, 191-200.

Sathya Sundaram 2005. XRel_Change_SQL: A Change Detection System for Unordered XML Documents, MS Thesis, University of Missouri - Rolla, Aug 2005.

S. Banerjee, V. Krishnamurthy,M. Krishnaprasad and R. Murthy 2000. Oracle8i - The XML Enabled Data Management System. In *Proceedings of the International Conference on Data Engineering (ICDE)*, California, March 2000.

Shanmugasundaram, J., Kiernan, J. ,Shekita, E. ,Fan, C. and Funderburk, J. 2001. Querying XML views of relational data. In *VLDB*. Roma, Italy.

Software AG. 2002. Tamino XML server. 2002. *http://www.softwareag.com/corporate/products/tamino/default.asp*

U. Dayal and P. A. Bernstein 1982. On The Correct Translation Of Update Operations on Relational Views. In *ACM Transactions on Database Systems*, 1982.

Vittori, C. , Dorneles, C., and Heuser, C. 2001. Creating xml documents from relational data sources. In *Proceedings of EC-Web 2001*, Munich, Germany, Sept. 2001, pp. 60-70.

World Wide Web Consortium (W3C),. Extensible Markup Language (XML) 1.0 (2nd edition), W3C Recommendation, October 2000. *http://www.w3.org/TR/2000/REC-xml-20001006*.

Xiaolei Qian and Gio Wiederhold 1991. Incremental Recomputation of Active Relational Expressions, *TKDE*, 3(3), Sept. 1991.

Yoshifumi Masunaga1984. A Relational database view update translation mechanism. In *proceedings of the Tenth International Conference on Very Large Databases (VLDB)*, Singapore, 1984.

Yoshikawa M., Amagasa, T., Shimura, T. and Uemura, S. 2006. Xrel: A path-based approach to storage and retrieval of xml documents using relational databases. *ACM Transactions on Internet Technology 1*, 2001, 110-141.

**Sanjay Kumar Madria** is an Associate Professor in the Department of Computer Science at the Missouri University of Science and Technology, USA. He received his Ph.D. in 1995 in Computer Science from Indian Institute of Technology, Delhi, India. He has published more than 120 Journal and conference papers in the areas of mobile computing, sensor networks, security, XML, web data warehousing, and databases in general. He co-authored a book entitled "Web Data Management: A Warehouse Approach" published by Springer-Verlag. He received UMR faculty excellence award in 2007, Japanese Society for Promotion of Science invitational fellowship in 2006, and Air Force Research Lab's visiting faculty fellowship in 2008. He is IEEE Senior Member and a speaker under IEEE Distinguished Visitor program.

**Janarthanan Eindhal** obtained his bachelors degree in Information Technology at Hindustan College of Engineering, Madras University, Chennai, India, in May 2004. He received his MS by thesis degree in Computer Science from from the University of Missouri-Rolla, USA, in May 2007. Since then he is working as software engineer in the industry.