A Mapping Schema and Interface for XML Stores

Sihem Amer-Yahia
AT&T Labs – Research
180 Park Ave
New Jersey, USA
sihem@research.att.com

Divesh Srivastava
AT&T Labs – Research
180 Park Ave
New Jersey, USA
divesh@research.att.com

ABSTRACT

Most XML storage efforts have focused on mapping documents to relational databases. Mapping choices range from storing documents verbatim to shredding documents into relations in various ways. These choices are usually hard-coded into each storage system which makes sharing loading and querying utilities and exchanging information between different XML storage systems hard. To address these issues, we designed MXM and IMXM, a mapping schema and an interface API to define and query XML-to-relational mappings.

A mapping is expressed as an instance of MXM. MXM is declarative, concise and captures most existing XML-to-relational mappings. Mappings can be expressed for documents for which no schema information is provided or documents that conform to either a DTD or an XML Schema. IMXM is an interface that allows querying of information contained in a MXM mapping. IMXM is designed as a library of functions which makes it easy to use inside any utility or application that needs to gain access to the XML-to-relational mapping. MXM is extensible and can incorporate new XML-to-relational mappings. We implemented a prototype to define mappings as instances of MXM and generate a repository of meta information on the XML and the relational data and the mapping choices. We implemented IMXM on top of this repository and used it for generating a relational schema and loading XML documents.

Categories and Subject Descriptors

H.2.5 [**Heterogeneous Databases**]: Data Translation; H.2.3 [**Languages**]: Data description languages (DDL)

General Terms

Design Languages

Keywords

XML-to-Relational Mapping, XML Storage/Loading/Publishing

1. INTRODUCTION

Multiple techniques for mapping XML documents into a relational database have been proposed, both in industry [3, 7, 14] and in research [5, 8, 10, 15, 16, 17, 18, 19]. Each technique comes with its own design choices. Mapping choices range from storing XML documents or XML fragments as CLOBs to shredding the XML documents according to a predefined relational schema. In all cases, mapping choices are hard-coded into the storage system in internal data structures. Ideally, all existing XML-to-relational mappings should be made accessible through a common interface enabling to share utilities such as loading and querying the stored XML and to develop applications such as data exchange that need access to mapping choices. In this paper, we develop MXM and IMXM, a mapping schema and an interface API to express and query XML-to-relational mappings.

The variety of XML data found today [21] as well as the flexibility of representation offered by XML raises challenging issues for storing XML data in traditional relational systems [2]. Some XML applications built on top of the stored data might need only limited information on the structure of input documents, others rely on detailed information on the content and the structure of input documents to guarantee efficiency. For example, e-commerce applications that publish catalogs or portions of catalogs might need only coarse information about how XML documents are stored. On the other hand, applications that need to correlate data in an XML document such as medical records data [11] rely on a finer knowledge of the document structure and data types. Consequently, existing XML-to-relational mappings have explored several ways of mapping document content and structure.

Existing mapping proposals vary in their inlining/outlining choice (attributes are usually inlined, sub-elements might be) and in the way document structure is captured (using key/foreign key values, using special-purpose fields, ...). Utilities such as the translation of XQuery queries into SQL, XML document loading programs, XML publishing programs cannot be shared among multiple storage systems because each of them depends on the specified storage choices. In addition, applications such as data exchange, built on top of multiple XML stores, need to gain access to the XML-to-relational mapping information. These utilities and applications would greatly benefit from having a common interface to the mapping choices in each storage system.
Our contributions are as follows:

- We designed MXM, a schema to express mappings from XML to relations. MXM is concise declarative and extensible. MXM can be used for mapping documents for which no schema information is provided, or documents that conform to a DTD or documents that conform to an XML Schema. A mapping is defined as an instance of MXM.

- We developed IMXM, an interface to query mappings. IMXM was designed to query all choices made in a mapping. IMXM is a library of functions that can be easily used by any utility or application that needs access to the stored XML data.

- We implemented a prototype in which a mapping (defined as an instance of MXM) is parsed and stored in a mapping repository on top of which we implemented IMXM. We used IMXM to create the target relational schema. We also used IMXM to develop a loader that reads XML documents and stores them in a relational database using information from the repository.

Section 2 discusses different XML-to-relational mapping examples and gives our design desiderata. MXM and IMXM are described in Section 3 and Section 4. Section 5 contains implementation details. Section 6 discusses related work. Conclusions and ongoing work are given in Section 7.

2. MOTIVATION

Existing XML-to-relational mapping techniques can be classified into three categories [2]:

Generic Techniques such as the `Edge`, `Attribute` and `Universal` that were first introduced in [10] to store XML documents in tables and then used in [19]. These techniques do not use DTD or XML Schema information and view each input document as a tree which is mapped into a generic relational schema that captures elements, attributes and document structure.

Schema-driven Techniques that derive a relational schema from a DTD or XML Schema. This is done using either a fixed or a flexible set of rules. Fixed mappings (`Basic`, `Shared` and `Hybrid`) are defined from DTDs to tables [17]. Flexible mappings [5] use an XML Schema. In both cases, some broad principles are applied. *Repetition* of sub-elements is modeled in separate tables. *Non-repeated* sub-elements may be "inlined" in their parent table. *Optionality* is handled using nullable fields. *Choice* is represented using multiple tables or a universal table with nullable fields. Document structure is captured through specific field values.

User-defined Techniques refer to the ones proposed by commercial systems [3, 7, 14] and rely on the user to give the target relational schema. The mapping is provided either programmatically through special purpose queries or using a declarative interface (annotated schemas). Microsoft SQL Server also implements the generic `Edge` approach.

In [16], the authors provide a description and a comparison of multiple techniques to capture order in XML documents including the techniques described in [1, 12, 23].

2.1 Mapping Examples

We consider the simple DTD given below where each address book has an owner who has a set of emails (both represented as attributes). An address book could be organized as a (possibly empty) sequence of pairs of fullname (or lastname) and telephone information (telephone numbers).

```
<!DOCTYPE addressBook [
<!ELEMENT addressBook addressBookContent>
<!ATTLIST addressBook owner CDATA email CDATA>
<!ENTITY addressBookContent (fnamelnameTelephone)*>
<!ENTITY fnamelnameTelephone (fnamelname,telephone)>
<!ENTITY fnamelname (fullname|lastname)>
<!ELEMENT fullname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>]>
```

In order to store XML documents that conform to this DTD, elements, attributes and document structure need to be captured. In addition, more information might need to be captured if an XML Schema [20] is given. For example, in the XML Schema below, type information can be associated to attributes (instead of them being all strings). The attribute `email` is a list of strings (that could be the different possible emails of an address book's owner) and the attribute `telephone` is an integer (representing a telephone number in this case).

```
<schema>
  <element name="addressBook">
    <attribute name="owner" type="string"/>
    <attribute name="email" type="strList"/>
    <group ref="addressBookContent"/>
  </element>
  <group name="addressBookContent">
    <choice>
      <group ref="fnamelnameTelephone"
             minOccurs=0 maxOccurs="unbounded"/>
    </choice>
  </group>
  <group name="fnamelnameTelephone">
    <sequence>
      <group ref="fnamelname"/>
      <element name="telephone" type="integer"/>
    </sequence>
  </group>
  <group name="fnamelname">
    <choice>
      <element name="fullname" type="string"/>
      <element name="lastname" type="string"/>
    </choice>
  </group>
  <simpleType name="strList">
    <list itemType="string"/>
  </simpleType>
</schema>
```

This XML Schema reflects as much as possible the DTD. The reader must note that we have decomposed the DTD into entities and the XML Schema into groups. Entities should be seen as macros. Groups are generic names that designate a sequence group (ordered set), an all group (unordered set) or a choice [20]. For example, the entity `fnamelnameTelephone` is modeled as a sequence group with the same name in the XML Schema. This decomposition is intentional as it is one of the key points of our approach and offers additional flexibility in expressing XML-to-relational mappings. In particular, the mapping approach described in [5] is based on the principle of mapping groups in XML Schema into tables in the relational schema. This mechanism abstracts the mapping from the XML Schema name typing mechanism and guarantees the uniqueness of names in the XML Schema. The reader must understand that the purpose of this rewriting is to try to address the same mapping issues from DTDs and from XML Schemas. A more formal definition should be given in a longer version of this paper.

We describe several relational schemas for mapping the above DTD and XML Schema. The first relational schema, RS1, given below, is similar to the schema-driven mappings of [5] and [17] (to the extent that in RS1, table names are specified by the user). Every element definition in the DTD is stored in a separate table. Every attribute is inlined in the element that contains it. The tag of each element is captured in tag. The name of each attribute is captured in the name of the field that contains its value. Document structure is captured by three fields: KEY (which is a unique identifier assigned to each element), PARENT (which is a foreign key to the KEY field contained in the parent element) and ORDINAL (for sibling order).

```
RS1
RaddressBook[addressBook_KEY,tag,owner_VALUE,email_VALUE]
fullnameTable[fullname_KEY,tag,fullname_VALUE,
              fullname_ORDINAL,fullname_PARENT]
lastnameTable[lastname_KEY,tag,lastname_VALUE,
              lastname_ORDINAL,lastname_PARENT]
telephones[telephone_KEY,tag,telephone_VALUE,
           telephone_ORDINAL,telephone_PARENT]
```

Another common way of capturing XML structure is to inline sub-elements inside their parent element. Inlining techniques are explored both in [5] and in [17]. In RS2, the sub-elements fullname, lastname and telephone are inlined in their parent element addressBook. The inlining is possible only if each addressBook contains only one occurrence of the pair (fullname,telephone) or (lastname,telephone), otherwise, additional processing is required (e.g, concatenating values into strings). In order to capture the structure of elements and attributes that are not inlined (that are outlined), RS2 uses an external table, Parent_Child that contains a foreign key to the parent element, a foreign key to the child element and an ordinal number to capture sibling order. To allow maximal sharing, outlining of attributes should also be permitted. This is the case of the attribute email for which a separate table has been created. The relationship between addressBook and email is captured in the Parent_Child table. Since attributes in XML are not ordered, the value of the field Child_ORDINAL for emails is not relevant. Note also that in RS2, none of the tables has a tag field because the name of the table reflects the tag of the element or the attribute it is storing.

```
RS2
Parent_Child[Key_PARENT,Key_CHILD,Child_ORDINAL]
addressBook[addressBook_KEY,owner_VALUE,fullname_VALUE,
            lastname_VALUE,telephone_VALUE]
email[email_KEY,email_VALUE]
```

In RS3, the structure of documents is captured using a preorder/postorder as described in [16]. Each tuple has unique preorder and postorder identifiers that correspond to traversing the XML document in this order. The preorder field in each table is also used as a key field. Each tuple has also a level number and a document identifier. Since this representation is powerful enough to capture the structure of an XML document, the PARENT and ORDINAL fields are not needed anymore. Note that users could specify table names (e.g., Rlastname).

```
RS3
addressBook[addressBook_PRE,addressBook_POS,addressBook_LEV,
            addressBook_DOC,owner_VALUE,email_VALUE]
fullname[fullname_PRE,fullname_VALUE,fullname_ORDINAL,
         fullname_PARENT]
Rlastname[lastname_KEY,tag,lastname_VALUE,lastname_ORDINAL,
          lastname_PARENT]
telephone[telephone_KEY,telephone_VALUE,telephone_ORDINAL,
          telephone_PARENT]
```

In addition to variance in capturing structure, XML-to-relational mappings may differ in the way they capture groups. As an example, in RS4, given below, the sequence group formed by the choice between elements fullname and lastname and the element telephone is captured by storing these elements together in the same table. This mapping is explored in [5]. Structure is captured using a Parent_Child table.

```
RS4
Parent_Child[Key_PARENT,Key_CHILD,Child_ORDINAL]
addressBook[addressBook_KEY,owner_VALUE,email_VALUE]
fnamelnameTelephone[fnamelnameTelephone_KEY,fullname_VALUE,
                    lastname_VALUE,telephone_VALUE]
```

Another aspect of variance in capturing element content comes from capturing choices. For example, the choice between fullname and lastname is inlined in the same table both in RS4 and in RS5. This choice can also be captured by outlining those elements into separate tables and capturing their relationships to their parent sequence (see variant below).

```
RS5
Parent_Child[Key_PARENT,Key_CHILD,Child_ORDINAL]
addressBook[addressBook_KEY,owner_VALUE,email_VALUE]
fnamelnameTelephone[fnamelnameTelephone_KEY,telephone_VALUE]
fnamelname[fnamelname_KEY,fullname_VALUE,lastname_VALUE]
```

In order to outline the elements fullname and lastname, we can write:

```
fnamelnameTelephone[fnamelnameTelephone_KEY,telephone_VALUE]
fullname[fullname_KEY,fullname_VALUE]
lastname[lastname_KEY,lastname_VALUE]
```

The mechanism that enables mapping groups is very powerful and allows to derive any level of inlining/outlining. these mappings are captured in [5]. However, document structure is always captured as in schema RS1.

Finally, RS6 shows the case where documents are mapped to a CLOB. CLOBs could be used to store any XML fragment [3, 14]. CLOBs could also be created to store attributes.

```
RS6
CaddressBook[addressBook_KEY,text_VALUE]
```

2.2 Mapping Desiderata

Except for the use of an external relation to map structure (which is an additional feature of our mappings), all other examples are captured in existing XML-to-relational mappings. Our goal is to provide a declarative mechanism to specify the mappings that have been proposed in both research and industry and offer an interface to access information about mappings. The interface should allow to query the mapping choices as well as details about the XML and relational schemas. We are not aiming at capturing all possible XML-to-relational mappings. We want to design our mapping language to be extensible so that new XML-to-relational mappings could be incorporated into the language and the interface. We believe that by capturing existing research and industry proposals, our language remains simple

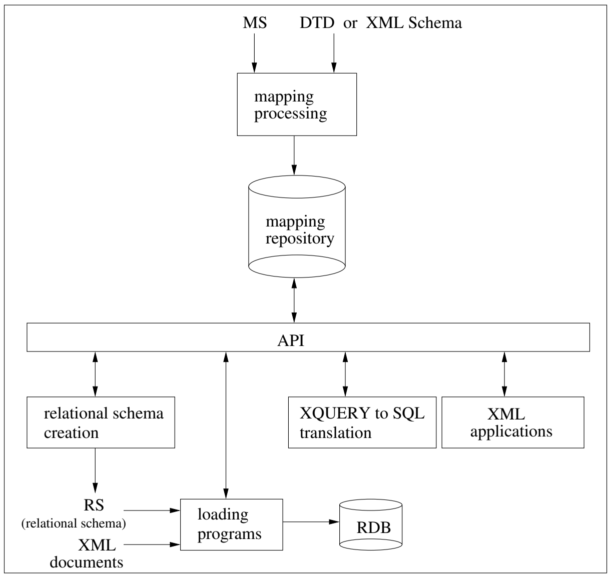

Figure 1: Overall Architecture

enough to use while enabling sharing of utility programs and develop applications.

Capturing generic techniques described in [10] is straightforward. These techniques are uniquely addressed by their name and correspond to generating a relational schema that is the same for any document.

Capturing fixed schema-driven techniques described in [17] is also straightforward. The flexible schema-driven technique of [5] is based on rewriting the input XML Schema into equivalent XML Schemas and selecting the most efficient one based on a query mix and a set of statistics. The XML Schema rewriting step is prior to any XML-to-relational mapping. Therefore, in order to capture the mappings described in [5], we only need to be able to express their XML-to-relational mapping rules. In these rules, document structure is captured using parent pointers as in `RS1` and element names are described using a `tag` field. All we need to guarantee is that our mapping schema enables these rules. The last mappings are the user-defined mappings where document structure is captured in the same way as in [5]. Finally, we capture structure mappings described in [16].

3. DESIGN OF MXM

3.1 Design Choices

In order to make the design of our mapping language extensible, we express its grammar in the W3C XML Schema [20]. The design of MXM is summarized in Figure 1.

Symmetry of elements and attributes: We map attributes and elements similarly. Attributes can be outlined for maximal sharing. When they are, their relationship to their containing elements is captured in the same manner as document structure except that attributes are not ordered. Outlining attributes has one appealing benefit. The relational model imposes some limitations when mapping attributes whose type is complex to a field in a table. Thus, we could convert attributes with a complex type into strings by concatenating values or we could allow outlining of attributes and thus capture complex types more naturally. Both choices are expressible in our mappings.

Mapping groups: The ability of mapping groups offers additional flexibility in the mapping (see the relational schema `RS4` given in Section 2.1). This design choice also enables the use of DTDs and XML Schemas to describe input documents. In DTDs, entities are assimilated to groups and non-terminal nodes are used to specify the XML-to-relational mapping. When an XML Schema is given, element, attribute and group names are used in the mapping. In XML Schema, two elements with two different (tag) names, might have the same type (in our case, they would refer to the same group name). If a group is mapped to a table, we allow the possibility of specifying that a tag field needs to be created in the table. Sometimes, groups and tags are used interchangeably, in which case, the user could specify that the table to which the type is mapped has the same name as the group, in which case, no tag field is created in the table. The user could also specify that a tag field should be created in the table and assign to the table a different name. Our mapping schema allows all these cases.

Mapping document structure: In order to capture multiple possibilities of mapping document structure, each possibility is given a distinct name which is used in the mapping. For example, `PCO` is used to specify that document structure will be captured by the parent, child and ordinal fields in each table. `PREPOS` is used to specify that preorder, postorder, level and document identifiers should be used. `DEWEY` is used to mean the Dewey Decimal Classification developed for general knowledge classification [12]. Document structure encoding is specified once for a mapping and used uniformly across all XML documents. Finally, some encoding such as the `Edge` approach described in [10] and the `Hybrid` approach described in [17], are very specific choices that are also addressed by their name in the mapping.

Naming tables and fields: In our design, we allow flexible naming of tables. If the user specifies a table name, it will be used, otherwise, a default name will be generated. However, field names are system-generated because they are used to either capture structure (e.g., `_PARENT` and `_ORDINAL` fields) or to capture inlined values (i.e., the `_VALUE` field). An extension to our design might allow the user to specify field names.

3.2 Grammar for MXM

We define a `XtoRMapping` as being composed of a mapping of document structure, `StructMap`, a mapping of elements, attributes and groups into tables, `TableMap`, and a mapping of elements and attributes into CLOBs, `CLOBMap`.

```
<xsd:schema>
  <element name="XtoRMapping">
    <attribute name="from" type="string" use="required"/>
    <attribute name="to" type="string" use="required"/>
    <element name="StructMap"/>
      <attribute name="whichMap" type="string"
                 use="required"/>
    <element name="TableMap">
      <element name="table"
               minOccurs="1" maxOccurs="unbounded">
        <attribute name="whichTable" type="string"
                   use="optional"/>
        <attribute name="tagField" type="string"
                   use="optional"/>
        <element name="sourceName" type="string"
                 minOccurs="1" maxOccurs="unbounded"/>
      </element>
    </element>
    <element name="CLOBMap">
      <element name="CLOB" type="sourceNames"
               minOccurs="1" maxOccurs="unbounded">
        <attribute name="whichCLOB" type="string"
                   use="optional"/>
        <element name="sourceName" type="string"
                 use="required"/>
      </element>
    </element>
  </element>
</xsd:schema>
```

StructMap indicates which one of the following techniques is used to capture document structure between elements and outlined sub-elements and attributes. The attribute whichMap can have one of the following values empty, "PREPOS", "DEWEY", "PCO", "EXTREL", "EDGE", "ATTRIBUTE", "UNIVERSAL", "BASIC", "SHARED", "HYBRID". These values could be extended to incorporate new structure mappings. The effect of each of "PREPOS", "DEWEY", "PCO" is to generate appropriate fields in each table. The effect of "EXTREL" is to generate a Parent_Child table and a key field in each table (see example schema RS2 in Section 2.1). Finally, the effect of "EDGE", "ATTRIBUTE", "UNIVERSAL", "BASIC", "SHARED" and "HYBRID" is to create the tables that correspond to each method. In these cases, no other table will be created. If the value of whichMap is empty, structure will not be captured.

TableMap is used to create tables from elements, attributes or groups (choice, sequence and allgroup). A table might be assigned a name (otherwise it is automatically generated by concatenating all source names together). It is also possible to specify whether a tag field should be created or not (see discussion in Section 2.2). In the case input documents are described with a DTD, any non-terminal node name in the DTD can be used as a source name. In the case input documents are described with an XML Schema, element, attribute and group names could be used as source names to create tables. In both cases, when multiple source names are used, data that corresponds to these names is stored in the same table.

Finally, CLOBMap indicates the creation of a CLOB from a source name. The name of a CLOB is either given or generated automatically. A CLOB containing all the substructure rooted at the specified name is created.

A consequence of our mapping schema design is the possibility for the user to specify which mapping should be used for document structure. In addition to simulating existing mapping proposals, this flexibility offers additional advantages for future applications. Our design could also be expanded to permit dual mappings as in the design of storing LDAP data in a relational backend [13]. In this design, LDAP entities are stored in two forms: a textual form that resembles CLOBs and a relational form.

3.3 Default Mapping Rules

We chose to make the specification of MXM short and concise and enforce the following default mapping rules: (1) If not given, table and CLOB names are system-generated. (2) Field names that capture document structure (hierarchy and tag information) are always system-generated. (3) Field names that capture inlined element and attribute values are always system-generated. (4) When repeated elements or complex type attributes are inlined, their values are concatenated into a single field value. (5) If a table is not created for a source name (element, attribute and group), it is inlined by default. (6) When elements or attributes are inlined, the name of the field that contains their value is their tag name concatenated to _VALUE. (7) The Parent_Child table is generated automatically when EXTREL is specified as a way to capture document structure.

In order to avoid hard-coding the semantics of default mapping rules into each application, these rules could be specified in a "configuration file" that would be parsed and stored along with other mapping information. The main benefit of writing default rules as a separate specification is that they could be made queryable and thus applications built on top of the mapping information could be abstracted from any hard-coded choice.

3.4 Examples of MXM Mappings

We give the MXM specification of each mapping example in Section 2.1. The first one specifies that structure is captured by parent and ordinal values. It also specifies table names.

```
<XtoRMapping from="XS" to="RS1">
  <StructMap whichMap="PCO"/>
  <TableMap>
    <table whichTable="RaddressBook">
      <sourceName> addressBook </sourceName>
    </table>
    <table whichTable="fullnameTable">
      <sourceName> fullname </sourceName>
    </table>
    <table whichTable="lastnameTable">
      <sourceName> lastname </sourceName>
    </table>
    <table whichTable="telephones">
      <sourceName> telephone </sourceName>
    </table>
  </TableMap>
  <CLOBMap/>
</XtoRMapping>
```

The mapping of RS2 illustrates the use of an external relation Parent_Child to capture document structure. In addition, since none of fullname, lastname or telephone and none of the groups that contain them creates a table, they are all inlined. Finally, email is outlined in its own table. The relationship between the elements addressBook and email is captured in the Parent_Child table, therefore, enabling to store the list of emails of the same owner in multiple tuples in the email table. Table names are generated automatically since they are not specified.

```
<XtoRMapping from="XS" to="RS2">
  <StructMap whichMap="EXTREL"/>
  <TableMap>
    <table tagfield="NONE">
      <sourceName> addressBook </sourceName>
    </table>
    <table tagfield="NONE">
```

```
      <sourceName> email </sourceName>
    </table>
  </TableMap>
  <CLOBMap/>
</XtoRMapping>
```

RS3 shows the use of the preorder/postorder approach to capture document structure.

```
<XtoRMapping from="XS" to="RS3">
  <StructMap whichMap="PREPOS"/>
  <TableMap>
    <table tagfield="NONE">
      <sourceName> addressBook </sourceName>
    </table>
    <table tagfield="NONE">
      <sourceName> fullname </sourceName>
    </table>
    <table whichTable="Rlastname">
      <sourceName> lastname </sourceName>
    </table>
    <table tagfield="NONE">
      <sourceName> telephone </sourceName>
    </table>
  </TableMap>
  <CLOBMap/>
</XtoRMapping>
```

RS4 shows the possibility to map a sequence group into a table. Since none of the groups, elements or attributes used in the sequence group is used to create a separate table, they are all inlined inside the `fnamelnameTelephone` table. Structure is stored in the `Parent_Child` table.

```
<XtoRMapping from="XS" to="RS4">
  <StructMap whichMap="EXTREL"/>
  <TableMap>
    <table tagfield="NONE">
      <sourceName> addressBook </sourceName>
    </table>
    <table tagfield="NONE">
      <sourceName> fnamelnameTelephone
    </sourceName> </table>
  </TableMap>
  <CLOBMap/>
</XtoRMapping>
```

RS5 shows a variant of RS4 where the choice group `fnamelname` is outlined.

```
<XtoRMapping from="XS" to="RS5">
  <StructMap whichMap="EXTREL"/>
  <TableMap>
    <table tagfield="NONE">
      <sourceName> addressBook </sourceName>
    </table>
    <table tagfield="NONE">
      <sourceName> fnamelnameTelephone </sourceName>
    </table>
    <table tagfield="NONE">
      <sourceName> fnamelname </sourceName>
    </table>
  </TableMap>
  <CLOBMap/>
</XtoRMapping>
```

The next variant of RS5 shows the possibility of outlining `fullname` and `lastname` from the choice group, thereby creating a separate table for each of them.

```
<XtoRMapping from="XS" to="RS5">
  <StructMap whichMap="EXTREL"/>
  <TableMap>
    <table tagfield="NONE">
      <sourceName> addressBook </sourceName>
    </table>
    <table tagfield="NONE">
      <sourceName> fnamelnameTelephone </sourceName>
    </table>
    <table tagfield="NONE">
      <sourceName> fullname </sourceName>
    </table>
    <table tagfield="NONE">
      <sourceName> lastname </sourceName>
    </table>
  </TableMap>
  <CLOBMap/>
</XtoRMapping>
```

Finally, RS6 shows the creation of one CLOB to contain the whole document. No table mapping is given in this case. The attribute `whichMap` is not specified, thus structure is not captured.

```
<XtoRMapping from="XS" to="RS6">
  <StructMap>
  </TableMap>
  <CLOBMap>
    <CLOB whichCLOB="CaddressBook">
      <sourceName> addressBook </sourceName>
    </CLOB>
  </CLOBMap>
</XtoRMapping>
```

4. DESIGN OF IMXM

Since MXM has an XML syntax, it could be queried using a language such as XQuery. XQuery is powerful and could be used to extract any information from the mapping. However, it assumes knowledge of the language itself. We designed a simple interface API composed of a set of functions that query element and attribute mappings and information on the generated relational schema. The first version of the API contains the following functions: `getStructMap()`, `isInlined(ElemName|AttName)`, `getTableName(ElemName|AttName)`, `getCLOBName(ElemName|AttName)`, `getFields(TableName)`, `getFieldType(FieldName)`.

In addition, if default mapping rules are made queryable through the API, programs and applications built on top of the mapping information could use this API and avoid to hard-code default choices. Examples of functions that query defaults are `getDefTableNaming()` that returns the default for naming tables, `getDefValNaming()` that returns the default naming of fields that contain values and finally, `IsDefInline()` which indicates whether the default is to inline or outline elements and attributes.

The interface could also be designed at multiple granularities. For example, instead of combining several functions in the API, a function that returns a set of information about the mapping of a particular element (mapping of its attributes, mapping of its sub-elements, ...) could be useful. The granularity of the API depends on the applications that will make use of it.

5. IMPLEMENTATION OF MXM

We implemented MXM and IMXM on top of a relational system. The mapping schema MS is parsed and stored in a repository. We define a relational schema that captures the information stored in the repository. When a mapping schema is parsed, it is stored in a relational database. We implemented IMXM as a library of functions in C with ODBC calls to the database. IMXM could also be implemented in Java using JDBC calls. Using IMXM, we implemented a relational schema generator and a set of loading programs that parse XML documents and populate tables.

The mapping repository conforms to the relational schema given below. In order to apply the default mapping rules described in Section 3.3, we need to record information on the input DTD or XML Schema (if any). This is what the first set of tables describes.

`ElemGroup` associates element names (tags) with groups they map to (that could be the same as the element itself). `AttGroup` associates attributes to groups. `GroupInfo` is the information about existing groups where `whichGroup` might have one of these values `empty`, `choice`, `sequence` or `all`. `GroupGroup` captures relationships between parent groups and their children groups.

```
ElemGroup[EName,Gkey]                StructInfo[whichMap]
AttGroup[AName,Gkey]                 GroupTable[GKey,TKey]
GroupInfo[GKey,GName,whichGroup]     GroupField[GKey,FKey]
GroupGroup[GPKey,GCKey]              GroupCLOB[GKey,BKey]

TableInfo[TKey,TName]
FieldInfo[FKey,FName,TKey]
CLOBInfo[BKey,BName]
```

When the mapping is parsed, the remaining tables are populated. `StructInfo` contains information on structure mapping. `GroupTable` associates groups with tables. `GroupField` associates groups with fields. `GroupCLOB` associates group with CLOBs. The last set of tables describe the relational schema that is generated.

6. RELATED WORK

The schema adjunct framework described in [22] is a mechanism that associates XML fragments to an input XML Schema by embedding mappings into the original schema. The main difference between this work and ours is that mappings in the schema adjunct framework have to be embedded in the original schema which has to be either a DTD or an XML Schema. Therefore, schema-less documents cannot be mapped using this framework. In addition, every single entity in the original schema has to be explicitly mapped which might make the specification of a mapping very long. Finally, the schema adjunct framework does not handle mappings of document structure.

The IBM solution for storing XML documents in the relational system DB2 relies on a mapping schema [7]. This schema is a simplified version of an XML schema where only elements and attributes are used. Elements and attributes can be annotated with the tables and fields to which they are mapped. This solution assumes that the input schema is given in the IBM format. It offers limited mappings because it allows mapping elements and attributes only (groups cannot be mapped) and does not allow to map elements to field names. However, it allows the specification of arbitrary predicates to partition elements into multiple tables. Our approach would need to be extended to support this mechanism. Finally, the IBM solution does not provide an API.

In the Rainbow system [24], XQuery is used to specify a mapping. While this language is powerful enough to express existing mappings, its specification would be much longer and it is not clear how to use it for this purpose. Also, no interface API is provided.

The work presented in [9] describes XML-to-relational transformations that preserve constraints on DTDs. Our mapping schema proposal does not handle constraints and would need to be extended.

7. CONCLUSION

We presented MXM, a flexible XML-to-relational mapping schema. MXM could be used to map schema-less documents as well as documents that conform to either a DTD or an XML Schema. MXM is designed in XML which makes it easily extensible to express other mapping choices including mapping XML data to a target system that is different from the relational one. We also designed IMXM, an interface to the information contained in a MXM mapping, the source DTD or XML Schema and the generated relational schema.

MXM and IMXM help sharing efforts when developing loading and querying utilities on top of XML storage systems. In addition, they enable building applications for data exchange between multiple stores in which XML data might have been mapped differently.

We are currently investigating extensions to MXM to support additional mappings. We are also using IMXM to enable data transfer from one relational store to another.

8. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE 2002.

[2] S. Amer-Yahia, M. Fernández. Techniques for Storing XML (Tutorial). ICDE 2002.

[3] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, R. Murthy. Oracle8i - The XML Enabled Data Management System. ICDE 2000.

[4] D. Barbosa, A. Barta, A. Mendelzon, G. Mihaila, F. Rizzolo, P. Rodriguez-Gianolli. ToX - The Toronto XML Engine. International Workshop on Information Integration on the Web, 2001.

[5] P. Bohannon, J. Freire, P. Roy, J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. ICDE 2002.

[6] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, M. Stefanescu XQuery 1.0: An XML Query Language. http://www.w3.org/TR/query-datamodel/.

[7] J. M. Cheng, J. Xu. XML and DB2. ICDE 2000.

[8] A. Deutsch, M. Fernandez, D. Suciu. Storing Semistructured Data with STORED. SIGMOD 1999.

[9] D. Lee, W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. ER 2000.

[10] D. Florescu, D. Kossman. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. IEEE Data Eng. Bulletin 1999.

[11] HL7. http://www.hl7.org/.

[12] Online Computer Library Center. Introduction to the Dewey Decimal Classification. http://www.oclc.org/oclc/fp/about/about_the_ddc.htm.

[13] OpenLDAP. www.openldap.org/.

[14] M. Rys, Microsoft. Bringing the Internet to Your Database: Using SQLServer 2000 and XML to Build Loosely-Coupled Systems. ICDE 2001.

[15] A. Schmidt, M. Kersten, M. Windhouwer, F. Waas. Efficient Relational Storage and Retrieval of XML Documents. WebDB 2000.

[16] J. Shanmugasundaram, I. Tatarinov, E. Viglas, K. Beyer, E. Shekita, C. Zhang. Storing and Querying Ordered XML using a Relational Database System. SIGMOD 2002.

[17] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. VLDB 1999.

[18] T. Shimura, M. Yoshikawa, S. Uemura. Storage and Retrieval of XML Documents using Object-Relational Databases. DEXA 1999.

[19] F. Tian, D. J. DeWitt, J. Chen, C. Zhang. The Design and Performance Evaluation of Various XML Storage Strategies. Wisconsin Database Group. Technical Report 2001.

[20] XML Schema Primer. http://www.w3.org/TR/xmlschema-0/.

[21] Sources of XML Data. http://www.xml.org/.

[22] S. Vorthmann, J. Robie, L. Buck. The Schema Adjunct Framework. http://www.extensibility.com/resources/saf_dec2000.htm.

[23] C. Zhang et al. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD 2001.

[24] X. Zhang et al. Rainbow: Mapping-Driven XQuery Processing System. SIGMOD Conference 2002.