A Generic and Flexible Framework for Mapping XML Documents into Relations

Sihem Amer-Yahia
AT&T Labs – Research
sihem@research.att.com

Fang Du
OGI/OHSU
fangdu@cse.ogi.edu

Juliana Freire
OGI/OHSU
juliana@cse.ogi.edu

ABSTRACT

As Web applications manipulate an increasing volume of XML data, there is a growing need for reliable systems to store and provide efficient access to these data. The use of relational database systems for this purpose has attracted considerable interest with a view to leveraging their powerful and reliable data management services.

Due to the mismatch between the XML and the relational models and the many different ways to map a given XML document into relations, it is hard to tune a relational engine and ensure that XML queries will be evaluated efficiently. Several approaches have been proposed to address this problem. In fact, major database vendors provide several means for database developers to describe how to map XML documents into relational tables. However, the available solutions are proprietary, and tied to a particular database backend. In addition, they are either limited with respect to expressivity and the kinds of mappings they can represent, or they are too complex to use.

In this paper, we propose an XML-to-relational mapping framework (and system) that is portable, expressive, and easy to use. We describe a mapping scheme wherein mappings are defined through annotations in an XML Schema and thus, not tied to any particular database engine. The various mapping dimensions were taken into account in the design of the annotations, making the framework flexible; capable of expressing a wide range of existing mappings strategies; and easily extensible to incorporate new mapping strategies. In effect, together with the schema information, the mapping provides a formal specification of the mapping. We show how this specification can be leveraged in two important tasks: in mapping analysis to ensure both correctness and that desirable properties hold for a given mapping; and in data shredding and query translation.

1. INTRODUCTION

XML is becoming the predominant data format in a variety of application domains (*e.g.,* supply-chain, scientific data processing, telecommunication infrastructure). Many such applications produce and consume large volumes of XML data and thus require efficient and reliable storage systems.

The use of relational database systems for this purpose has attracted considerable interest both by the research community and the database vendors. By relying on relational engines, XML developers can benefit from a complete set of data management services (including concurrency control, crash recovery, and scalability) and from the highly optimized relational query processors. However, as the example below illustrates, storing and querying XML data in an RDBMS is an involved task.

Copyright is held by the author/owner(s).
WWW2004, May 17–22, 2004, New York, NY USA.
ACM xxx.xxx.

```
<element name="IMDB" type=''imdb''>
 <element name="SHOW">
  <sequence>
    <element name="TITLE" type="string"/>
    <element name="YEAR" type="integer"/>
    <element name="AKA" type="string"
             minOccurs="1" maxOccurs="10"/>
    <element name="REVIEW" type="ANYTYPE"
             minOccurs="0" maxOccurs="unbounded"/>
    <choice>
       <sequence>
          <element name="BOXOFFICE" type="integer"/>
          <element name="VIDEOSALES" type="integer"/>
       </sequence>
       <sequence>
          <element name="SEASONS" type="integer"/>
          <element name="EPISODE" type="ANYTYPE"
             minOccurs="0" maxOccurs="unbounded"/>
       </sequence>
     </choice>
   </sequence>
  </element>
 </element>
```

Figure 1: Excerpt of IMDB schema

Example 1.1 (Mapping show data). *Consider the following scenario. FakeFilm.com plans to deploy a new Web site that publishes information about movies and TV shows. Since they use a relational database, they need to map the existing show data that is available in XML from the Internet Movie Database (IMDB) into their database. An excerpt of the IMDB schema and a sample document are shown in Figures 1 and 2, respectively. The document illustrates the variability that the show schema allows. A show may contain zero or more reviews, 1 to 10 alternative titles (*i.e., `AKA`*), and either information about movies or TV shows.*

In order to store a movie document, first, a mapping must be constructed *that indicates how the elements should be stored in the relational database. Because of the variability in the data allowed by the schema, many different mappings and corresponding relational configurations can be derived to store documents conformant with the IMDB schema. Figure 3 illustrates some of the alternatives. Configuration (a) results from inlining as many elements as possible in the same table, roughly corresponding to the strategies presented in [16]. Configuration (b) is obtained from configuration (a) by partitioning the `Reviews` table into two tables: one that contains New York Times reviews, and another for reviews from other sources. Finally, configuration (c) is obtained from configuration (a) by splitting the `Show` table into Movie shows and TV shows.*

Once a particular mapping is selected, the XML data must be loaded *into the relational system,* i.e., *documents must be shredded*

```
<IMDB>
  <SHOW>
    <TITLE>Fugitive, The</TITLE>
    <YEAR>1993</YEAR>
    <AKA>Auf der Flucht</AKA>
    <AKA>Fuggitivo, Il</AKA>
    <REVIEW>
      <SUNTIMES>
         <REVIEWER>Roger Ebert</REVIEWER>
         <RATING>Two thumbs up!</RATING>
         <COMMENT>
            This is a fun action movie,
            Harrison Ford at his best. </COMMENT>
      </SUNTIMES>
    </REVIEW>
    <REVIEW>
       <NYT>
          The standard Hollywood summer
          movie strikes back. </NYT>
    </REVIEW>
    <BOX_OFFICE>183,752,965</BOX_OFFICE>
    <VIDEO_SALES>72,450,220</VIDEO_SALES>
  </SHOW>

  <SHOW>
    <TITLE>X Files, The</TITLE>
    <YEAR>1994</YEAR>
    <AKA>Akte X - Die unheimlichen
         Fälle des FBI</AKA>
    <AKA>Aux frontieres du Reel</AKA>
    <SEASONS> 10 </SEASONS>
    <EPISODE>
      <NAME>Ghost in the Machine</NAME>
      <GUEST_DIRECTOR> Jerrold Freedman </GUEST_DIRECTOR>
    </EPISODE>
    <EPISODE>
      <NAME>Fallen Angel</NAME>
      <GUEST_DIRECTOR> Larry Shaw </GUEST_DIRECTOr>
    </EPISODE>
  </SHOW>
  ....
</IMDB>
```

Figure 2: Sample IMDB document

into tuples and loaded into the relational tables. Finally, at run-time, XML queries must be translated *into equivalent SQL queries over the mapped data.*

As Example 1.1 illustrates, the storage problem has many dimensions, including: mapping strategies; mapping definition language; data shredding; and query translation. Several mapping strategies *e.g.,* [3, 5, 9, 16, 17, 15]) and query translation algorithms (see [11] for a survey) have been proposed in the literature. Commercial RDBMSs provide different mapping mechanisms which can be automatic, using a pre-defined (and fixed) strategy, or manual, where the user must manually define the mapping using a proprietary language provided by the database vendor.

None of these solutions, however, addresses all the storage problems in a single framework. For example, work on mapping strategies often have little or no details about query translation [11]. Commercial RDBMS provide no query translation, *i.e.,* the mapped XML data must be queried through SQL, which requires users to have knowledge of the mapping details; and some useful mappings cannot be expressed using their proprietary languages.

In this paper, we propose *MDF*, a mapping definition framework that provides the first comprehensive solution to the relational storage of XML data. Key to the framework is a formal specification of a mapping which provides a simple, yet powerful, mapping scheme. The mapping scheme is flexible and is able to express a wide range of mapping strategies. Information in the mapping specification also allows the implementation of generic loaders and query translators, which are independent from the mapping strategy. In addition, it allows automated analysis of mappings, *e.g.,* to determine whether a mapping is *correct* and *lossless*.

Contributions. In sum, our main contributions are:

- We propose a novel scheme for defining XML-to-relational mappings;
- We describe the implementation of a system, based on this scheme, which provides the first comprehensive solution to the relational storage of XML data.

Outline. The rest of the paper is organized as follows. Section 2 gives an overview of our approach. We survey some of the techniques used to capture identity, structure and order of XML documents in Section 3. In Section 4, we discuss the different dimensions of a mapping and we introduce our generic mapping scheme. The architecture and implementation of the system, including the mapping interface, the loader and the query translator, are described in Section 5. Related work is reviewed in Section 6. We conclude in Section 7 with directions for future work.

2. OVERVIEW

There are many different ways to express XML-to-relational mappings. Languages such as XSLT [22] or IBM's DAD [10] can be used to define mappings that perform arbitrary transformations over an XML document. However, there are drawbacks in using such complex languages for mapping. First and foremost, users must learn these languages, which may require a steep learning curve. In addition, since they allow virtually any mapping, it is hard to reason about the constructed mappings. For example, it can be very hard to look at an XSLT program and determine whether the mapping it represents preserves all the information in the original document. Lastly, since complex transformations are allowed, the actual shredding of the documents can be very expensive, both in terms of processing and memory requirements; and specialized query translation engines may need to be designed on a per-application basis.

Thus, instead of using a language that allows arbitrary transformations, we use a more restrictive (and declarative) approach. Our mapping definition framework (*MDF*) relies on *annotating an input XML Schema with a limited number of pre-defined annotations*, thereby controlling the mappings that users can define. It is worthy of note that although less expressive than XSLT, a wide variety of mapping strategies can be expressed by combining these annotations, including strategies defined in the literature (*e.g.,* [3, 5, 9, 16, 17, 15]).

The ability to specify mappings using annotations has many benefits: it provides great flexibility in the choices of how to map elements, attributes and document structure; it is extensible – by defining new annotations new mapping choices can be made available (*e.g.,* support for different data models); it is portable – mapping specifications are independent from the target relational database.

Another advantage of the schema-annotation-based approach is the ability to analyze the specification. Different kinds of analyses are possible, including consistency and different notions of *correctness*. Understanding the properties of mappings is especially useful in a practical tool, to guide users in specifying their mappings and to ensure that the requirements of the applications are met by the mapping.

An important characteristic of a mapping is *losslessness*, *i.e.,* all the information in the source data is preserved in the target data. We develop a method to guide users in specifying lossless mappings by identifying which portions of a user-defined mapping might cause loss of information and enforcing a mapping for those portions.

Once a mapping is defined, applications that use the mapped data may need access to the details of the mapping. For example, an application that translates XQuery queries into SQL needs detailed

```
TABLE Show
   ( Show_id INT,
     title STRING,
     year INT,
     box_office INT,
     video_sales INT,
     seasons INT )

TABLE Review
   ( Reviews_id INT,
     tilde STRING,
     reviews STRING,
     parent_Show INT )

TABLE Episode
   ( Episode_id INT,
     episode STRING )

....
```

(a)

```
TABLE Show
   ( Show_id INT,
     title STRING,
     year INT,
     box_office INT,
     video_sales INT,
     seasons INT)

TABLE NYT_Reviews
   ( Reviews_id INT,
     review STRING,
     parent_Show INT )

TABLE Reviews
   ( Reviews_id INT,
     tilde STRING,
     review STRING,
     parent_Show INT )

TABLE Episode
   ( Episode_id INT,
     episode STRING )
....
```

(b)

```
TABLE Movie_Show
   ( Movie_Show_id INT,
     title STRING,
     year INT,
     box_office INT,
     video_sales INT )

TABLE TV_Show
   ( TV_Show_id INT,
     type STRING,
     title STRING,
     year INT,
     seasons INT )

TABLE Reviews
   ( Reviews_id INT,
     tilde STRING,
     review STRING,
     parent_Show INT )

TABLE Episode
   ( Episode_id INT,
     episode STRING )
....
```

(c)

Figure 3: Three storage mappings for shows

knowledge of the document structure and data types. Our framework provides an API to mapping information. The use of this API is illustrated in Section 5.

3. IDENTITY, STRUCTURE AND ORDER

XML-to-relational mappings define how the structure, element identity and order in XML documents are represented in a relational database. Existing mapping techniques generate a unique identifier for each node in the XML document tree which are used capture document structure. In what follows, we give a brief overview of some of the techniques proposed in the literature. Readers are referred to [18] for more details.

Key, Foreign Key and Ordinal. A simple way to capture parent/child relationships in an XML document is to assign a unique identifier to each element, and have a foreign key in the child record that points to the identifier of its parent. For example, in Figure 3(a), a foreign key `parent_Show` is created in `TABLE Review` which refers to a record in `TABLE Show`. Sibling order can be capture using an ordinal value (that can be the key of the element itself). We refer to this technique as KFO for Key, Foreign key and Ordinal. KFO is used in a number of mapping strategies (see *e.g.*, [9, 3]). As an example, the `Edge` table defined in [9] uses KFO.

Interval Encoding. In *interval encoding*, a unique $\{start, end\}$ interval identifies each node in the document tree. This interval can be generated in multiple ways. The most common method is to create a unique identifier, $start$, for each node in a preorder traversal of the document tree, and a unique identifier, end, in a postorder traversal. A nice property of this encoding is that the interval of a node is included in the interval of its parent node. In order to distinguish children from descendants, a level number is recorded with each node. This technique is used in the TIMBER [14] system.

Dewey The Dewey Decimal Classification was originally developed for general knowledge classification [6]. This encoding records, at each node, the path from the node to the document root by concatenating the identifiers of the nodes along that path. Thus, the property of Dewey is that the identifier of a node contains its parent node identifier and the level at which the node is in the document tree. For example, if the identifier at a node is `1.2.24.65`, then we know that the node is at the fourth level in the tree and that the identifier of its parent node is `1.2.24`. This encoding is used in LDAP directories and has been applied to XML storage in [12].

4. DEFINING MAPPINGS

Our goal in designing *MDF* was to create a declarative specification for mappings that allows users to express as well as combine multiple storage techniques (*i.e.*, different ways to capture structure and order). In order to achieve this goal, we identified orthogonal dimensions of a mapping.

Definition 4.1 (Element Mapping). An element mapping, EM, *is a function that maps an element into a table, column or CLOB in the relational schema.*

Definition 4.2 (Attribute Mapping). An attribute mapping, AM, *is a function that maps an attribute into a table, column or CLOB in the relational schema.*

Definition 4.3 (Structure Mapping). A structure mapping SM *defines an identity, structure and order mapping as described in Section 3.*

Definition 4.4 (Mapping). A mapping is defined by a quadruple (XS, EM, AM, SM) *where* XS *is an input XML Schema,* EM *is a set of element mappings,* AM *is a set of attribute mappings and,* SM *is a structure mapping.*

We refer to *EM*, *AM* and *SM* as the dimensions of a mapping. A declarative mapping specification should permit each mapping dimension to be defined independently, thus allowing different choices for the mapping dimensions to be combined, and enabling flexible and expressive mappings.

In *MDF*, a mapping is expressed by annotating an input XML Schema with element, attribute and structure mappings. Thus, we define appropriate annotations that capture each of these mapping dimensions.

4.1 Schema Annotations

Annotations can be associated to attributes, elements and groups in the input XML Schema. The annotation syntax corresponds to adding attributes from a namespace called ***mdf*** to a given input XML Schema. By using a different namespace, documents conforming to the initial schema still conform to the annotated schema,

Annotation attributes	Target	Value	Action
outline	attribute or element	true, false	If value is true, a relational table is created for the attribute or element. Otherwise, the attribute or element is mapped to one or multiple columns in its containing table (*i.e.,* inlined).
tablename	attribute, element or group	string	The string is used as the table name.
columnname	attribute or element of simple type	string	The string is used as the column name.
sqltype	attribute or element of simple type	string	The string overrides the SQL type of a column.
structurescheme	root element	KFO, Interval, Dewey	Specifies structure mapping.
edgemapping	element	true, false	If value is true, the element and its descendants are shredded according to Edge mapping [9].
maptoclob	attribute or element	true, false	If value is true, the element or attribute is mapped to a CLOB column.

Table 1: Annotation Attributes

which can be used to validate them. The annotation attributes are summarized in Table 1. Each row in the table contains an annotation attribute, its target (*i.e.,* element, attribute, group and simple type), its possible values and its action, depending on its target and value. Below, we use examples to illustrate the flexibility of the annotations supported in *MDF*.

4.1.1 Outline, tablename, columnname, sqltype.

In Figure 2(a), the element TITLE is a simple type under the complex type SHOW. In order to outline TITLE, we set its attribute **outline** from namespace ***mdf*** to true. As illustrated in Figure 4, the element TITLE is mapped to a new table **Showtitle** as specified by the annotation attribute **tablename**. However, the attribute **outline** in element YEAR is set of false which causes to inline this element in the table corresponding to SHOW. Note also in this figure the use of the annotations **sqltype** and **columnname** – these attributes are added to the YEAR element to specify that it should be mapped to a column with name **Showyear** and SQL type **NUMBER(4)** in the table corresponding to SHOW.

4.1.2 Structurescheme, edgemapping

The attribute **struturescheme** can be specified at the root element to define which structure mapping is used to capture element identity, document structure and order. By using the annotation:

```
<element name="IMDB" type="imdb"
         mdf:structurescheme="Dewey" />
```

we specify that "Dewey" will be the structure mapping used throughout the XML document. If the annotation attribute **edgemapping** is used in the **REVIEW** as follows:

```
<element name="REVIEW" type="ANYTYPE"
         minOccurs="0" maxOccurs="unbounded"
         mdf:edgemapping="true" />
```

the REVIEW element and its descendants are mapped using Edge [9], *i.e.,* a single table to store all the elements of the document. The following table[1] is created:

```
TABLE Review( ParentID VARCHAR(128),
                   source VARCHAR(128),
                   ordinal VARCHAR(128),
                   attrname VARCHAR(128),
                   flag VARCHAR(128),
                   value VARCHAR(128))
```

[1]For a detailed explanation about the columns in the **Review** table, we refer readers to [9].

```
<element name="SHOW">
  <sequence>
    <element name="TITLE" type="string"
        mdf:outline="true"
        mdf:tablename="Showtitle"/>
    <element name="YEAR" type="integer"
        mdf:outline="false"
        mdf:columnname="Showyear"
        mdf:sqltype="NUMBER(4)"/>
  </sequence>
</element>
```

$$\xrightarrow{}$$
$$map\ into$$

```
TABLE SHOW( ID VARCHAR(128),
            Showyear NUMBER(4),
            BOXOFFICES NUMBER(10),
            SEASONS VARCHAR(128))
TABLE Review( ID VARCHAR(128),
              ParentID VARCHAR(128),
              REVIEW VARCHAR(128))
TABLE Showtitle( ID, VARCHAR(128),
                     ParentID VARCHAR(128),
                     TITLE NUMBER(10))
```

Figure 4: Outlining simple types

Note that since Edge does not require an input schema, it is particularly useful for mapping unconstrained elements, *i.e.,* elements of ANYTYPE. The use of Edge is also beneficial for parts of the schema that are expected to be updated often.

4.1.3 Union Distribution

Union distribution is an example of a complex mapping that can be expressed by our schema annotation scheme. In the sample IMDB Schema of Figure 1, a SHOW may be either a movie or TV show. We can use our annotations to derive a schema that stores shows in two tables – one for movies and one for TV shows. This is similar to the union distribution rule defined in [3]. The annotated schema and derived relational configuration are shown in Figure 5.

Our annotations do not depend on a particular target relational schema which makes them portable on any relational system.

4.2 Mapping Properties

Mappings can be quite complex. Since many different choices are available for each mapping dimension, and especially for large schemata, it is easy to make mistakes while defining a mapping. It is thus important to be able to *automatically check* whether a mapping is *correct* and *lossless*. For example, whether it generates multiple tables with the same name in the relational schema or whether all elements in the document have been mapped. Below,

```
<element name="SHOW">
  <choice>
    <group ref="imdb:Movie"
      mdf:tablename="Movie" />
    <group ref="imdb:TV"
      mdf:tablename="TV"/>
    </choice>
</element>
<group name="MOVIE" >
  <sequence>
    <element name="TITLE" type="string"/>
    <element name="YEAR" type="integer"/>
    <element name="BOXOFFICE" type="integer"/>
    <element name="REVIEW" type="string"
      minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</group>
<group name="TV" >
  <sequence>
    <element name="TITLE" type="string"/>
    <element name="YEAR" type="integer"/>
    <element name="SEASONS" type="string"/>
  </sequence>
</group>
```

$$\overrightarrow{map\ into}$$

```
TABLE SHOW( ID VARCHAR(128) )
TABLE MOVIE( ID VARCHAR(128),
             ParentID VARCHAR(128),
             TITLE NUMBER(10),
             YEAR NUMBER(10),
             BOXOFFICES NUMBER(10))
TABLE TV ( ID VARCHAR(128),
           ParentID VARCHAR(128),
           TITLE NUMBER(10),
           YEAR NUMBER(10),
           SEASONS VARCHAR(128))

TABLE REVIEW( ID VARCHAR(128),
              ParentID VARCHAR(128),
              REVIEW VARCHAR(128))
```

Figure 5: Applying union distribution

we define the notions of mapping correctness and losslessness.

Definition 4.5 (Valid Relational Schema). A valid schema is a schema where (i) table names are distinct, (ii) CLOB names are distinct, (iii) each table has at least one field which is its key defined by one of the structure mapping methods given in Section 3 and, (iv) field names within the same table are distinct.

Definition 4.6 (Correct Mapping). A correct mapping is a mapping that generates a valid relational schema.

Definition 4.7 (Lossless Mapping). A mapping (XS, EM, AM, SM), is lossless if: (0) it is correct and, (1) EM defines an element mapping for each element in XS, (2) AM defines an attribute mapping for each attribute in XS and, (3) SM defines an order mapping.

Whereas a lossless mapping is always correct, a correct mapping may be lossy, *e.g.,* some elements may not be mapped into the relational configuration. It is worthy of note that some order mappings may not preserve all the order information in the document, *e.g.,* a KFO identity scheme may not preserve order among siblings. Since different applications have different requirements for mapping properties, it is useful to define different notions of losslessness. For example, for an application that does not require the order among sibling nodes to be preserved, selecting KFO will lead to a *correct* mapping for that particular application. In Section 5, we describe how mapping properties are enforced.

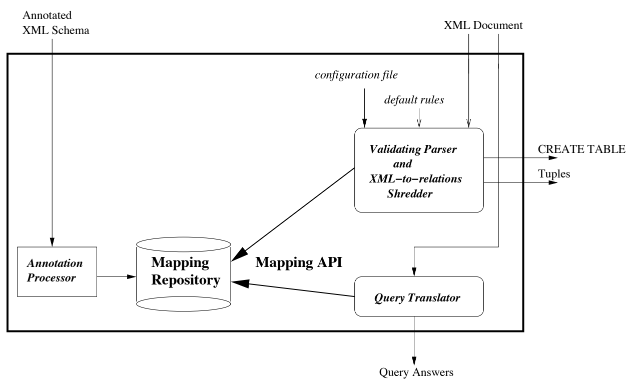


Figure 6: Architecture

5. IMPLEMENTATION

In this section, we describe the general architecture of *MDF* [2] and explain how we make use of the mapping specification to perform analyses, data shredding and query translation.

5.1 Architecture

The architecture of *MDF* is shown in Figure 6. Users can either manually annotate an input schema, or use the user interface provided by the system. The *annotation processor* parses an annotated XML Schema and creates a relational schema and a *mapping repository* to store all mapping information. The *mapping analyzer* can be used to verify mapping correctness and losslessness through a series of checks in order to indicate to the user which portions of his mapping are lossless. The *document shredder* accepts an input document and uses mapping information stored in the mapping repository to generate tuples in the corresponding tables. The mapping repository is also accessed by the *query translator* to generate SQL queries from XPath queries. Both modules access the mapping repository through an API that is made available in *MDF*.

5.2 User Interface

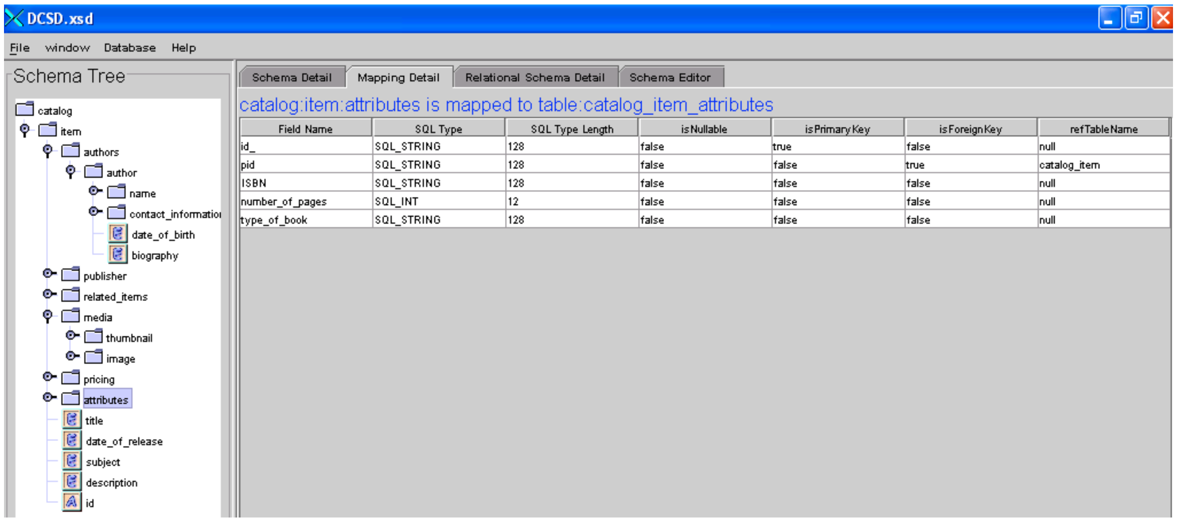
The *MDF* system provides a graphical user interface that helps user define and customize mappings. The interface displays the schema tree and corresponding relational tables, allowing users to visually check the connections between the XML elements and ther relational counterparts, as well as interactively modify the mapping specification. Some screendumps of the user interface are shown in Figure 7.

5.3 Annotation Processor

The annotation processor is in charge of parsing an annotated XML Schema, checking mapping correctness and losslessness, generating a mapping repository and producing the `CREATE TABLE` statements necessary to construct the relational schema. Correctness checks are performed by a simple algorithm that follows the definitions given in Section 4.2. More complex analyses are planned for future versions of the system.

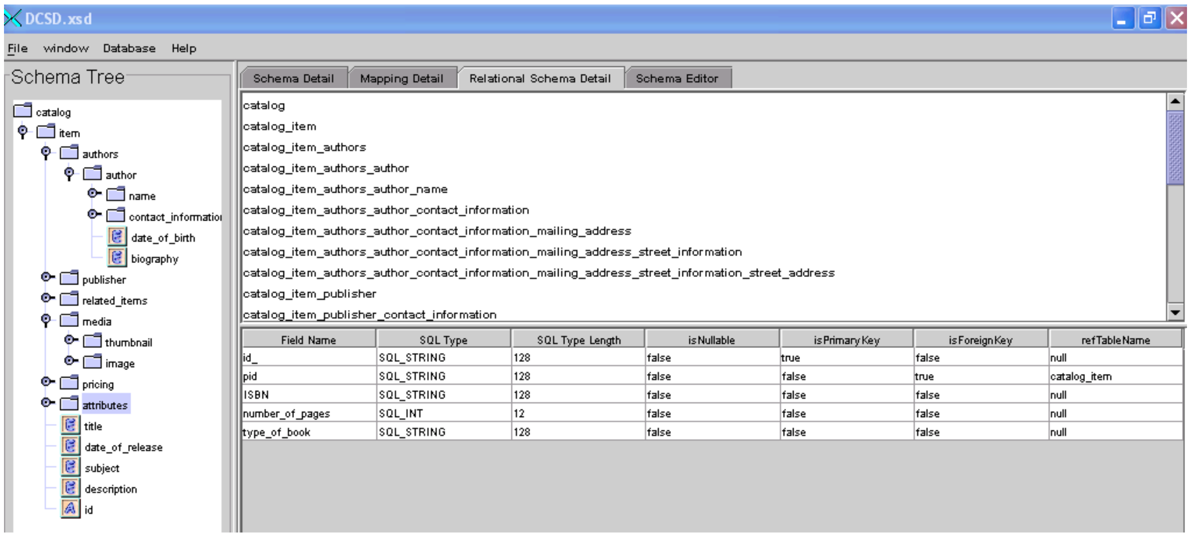
In order to simplify the process of checking for losslessness, the system provides a set of default rules that will be used to *complete* mapping specifications. For example, if the user does not specify a mapping for a portion of the input XML Schema, these default rules are applied to that portion of the schema. Default mapping rules have another benefit: they enable the concise definition of mapping

[2]Our system is currently available upon request.



catalog:item:attributes is mapped to table:catalog_item_attributes

Field Name	SQL Type	SQL Type Length	isNullable	isPrimaryKey	isForeignKey	refTableName
id_	SQL_STRING	128	false	true	false	null
pid	SQL_STRING	128	false	false	true	catalog_item
ISBN	SQL_STRING	128	false	false	false	null
number_of_pages	SQL_INT	12	false	false	false	null
type_of_book	SQL_STRING	128	false	false	false	null



catalog
catalog_item
catalog_item_authors
catalog_item_authors_author
catalog_item_authors_author_name
catalog_item_authors_author_contact_information
catalog_item_authors_author_contact_information_mailing_address
catalog_item_authors_author_contact_information_mailing_address_street_information
catalog_item_authors_author_contact_information_mailing_address_street_information_street_address
catalog_item_publisher
catalog_item_publisher_contact_information

Field Name	SQL Type	SQL Type Length	isNullable	isPrimaryKey	isForeignKey	refTableName
id_	SQL_STRING	128	false	true	false	null
pid	SQL_STRING	128	false	false	true	catalog_item
ISBN	SQL_STRING	128	false	false	false	null
number_of_pages	SQL_INT	12	false	false	false	null
type_of_book	SQL_STRING	128	false	false	false	null

Figure 7: Screendumps of the *MDF* GUI

specifications. Mapping every element and attribute definition in an XML Schema can be tedious, especially for large schemata. Using default rules frees users from having to annotate each element and attribute in an XML Schema. Note that our default mapping rules are not hard-coded in the system; user-defined rules can be added to or replace the built-in rules.

5.4 Mapping Repository and API

Mapping information is made persistent and an application programming interface (API) is provided that gives access to details of the mapping, such as, how elements and attributes are mapped, which mapping is used to capture document structure and which tables are available in the relational schema. Table 2 summarizes some the functions provided by the API. Making mapping information persistent avoids the need re-parse a mapping specification each time a document is loaded into the target database or that a query needs to be translated into SQL.

5.5 Document Shredder

The document shredder uses the mapping information to shred input documents that conform to the mapped schema into tuples that will populate tables in the target relational schema. Since mapping annotations are specified using attributes from a different namespace, the document shredder can validate the input XML document against the XML Schema. Tuples are generated while the document is parsed (using a standard XML parser). In our implementation, we use the SAX interface of Xerces [20], which is not only is efficient but also scalable. For example, the system was able to shred and load a 1GB document into a commercial RDBMS in less than 30 minutes.

Given an element or attribute in the input XML document, the document shredder is in charge of generating the appropriate tuple, field or CLOB. To do so, it must be aware of the mapping defined for that element or attribute. The shredder was designed to be generic and independent from the mapping specification. The key idea is to use the mapping API in order to retrieve information about how a particular element or attribute is mapped.

API Function	Input	Output	Semantics
structMap		KFO, Interval, Dewey	returns which structure mapping is used.
isTable	attribute or element name	true, false	determines whether the input has been maed to a table.
isField	attribute or element name	true, false	determines whether the input has been maed to a field.
isCLOB	attribute or element name	true, false	determines whether the input has been maed to a CLOB.
getTableName	attribute or element name	string	returns the name of the table used to map input.
getFieldName	attribute or element name	string	returns the name of the field used to map input.
getCLOBName	attribute or element name	string	returns the name of the CLOB used to map input.
getTableInfo	table name	table description	returns the table description in the relational schema.
getFieldInfo	field name	field description	returns the field description in the relational schema.
getCLOBInfo	CLOB name	CLOB description	returns the CLOB description in the relational schema.

Table 2: API Functions

Finally, the shredder is flexible and allows users to set various parameters (*e.g.,* target database system, login information, bulk loading option) either through the command line or through a configuration file. For more details about the shredded, see [8].

5.6 Query Translator

We developed an XPath-to-SQL query translator that supports a subset of XPath. Similar to the document shredder, the query translator is generic and does not hard-code mapping choices, instead it uses the information provided by mapping API to perform the translation.

The translation algorithm works on a subset of XPath syntax that includes: descendant/child axis; position based predicate of the format [position()=n]; and simple path based expression predicate. The algorithm consists of the following steps:[3]

```
Step 1: Resolve wildcards, so that a set of
        simple paths is obtained
Step 2: For each simple path, consult the mapping
        API and bind XML-to-relational mapping
        information  to the nodes in the path
Step 3: Generate SQL query for the
        annotated path
Step 4: Union the SQL queries (each of
        them corresponds to one path)
```

Step 2 uses some of the API functions, including: **isOutlined**, **getTableName**, **getTableName** and **getColumnName**. The information obtained by these functions enables the translator to dynamically decide how to perform the translation. For example, for a path `/IMDB/SHOW/TITLE`, if `TITLE` is inlined into `SHOW`, a selection query is generated (*i.e.*, SELECT title FROM SHOW); but if `TITLE` is outlined (see Figure 4), a join is generated instead (*i.e.,* SELECT title FROM SHOW, Showtitle WHERE SHOW.id = Showtitle.ParentID).

6. RELATED WORK

Although XML support in commercial relational engines is improving rapidly [10, 13, 21], the level of support varies widely across systems. Some practical problems include:

[3]For more details on the algorithm see [7].

Proprietary solutions: Major commercial systems provide proprietary solutions for loading XML documents. For example, the IBM DB2 XML Extender [10] requires users to write a Document Access Definition specification, consequently, developers must learn a new language in order to use DB2 as a back-end. In addition, if the same documents need to be loaded under the same mapping onto multiple databases, the developer will need to learn multiple mapping schemes and will need to write a different mapping specification for each.

Lack of flexibility: Although most commercial solutions provide powerful mapping schemes, some useful mapping strategies are not supported. For example, in Oracle's annotated schema, it is not possible to specify that *part* of the data is to be stored using a generic mapping such as Edge [9]. In addition, all systems hard-code the use of KFO (key-foreign key) to map element identity and document structure.

Scalability: Some loading solutions are not scalable. For example, SQLServer's OpenXML requires that XML documents be compiled into an internal DOM representation. As a result, the size of documents that can be loaded is limited by the size of main memory.

In what follows, we describe the main characteristics of existing mapping strategies and tools. For a more detailed description, the reader is referred to [1].

IBM DB2 XML Extender [10] defines the Data Access Definition (DAD) syntax to specify mappings and proprietary procedures to shred and build documents. Unlike DB2, we use an annotated XML Schema. DB2 extender requires that identifiers be present in the XML document. In order to correctly map repetitions, the parent element must have a key so that the table created for the repeated child can point to its parent's key. In addition, unlike our system which is not limited in the size of the documents it can handle, the maximum document size the IBM solution can handle is 1MB, and resulting tables can contain no more that 1024 rows.

Microsoft SQL Server [13] implements Edge, OpenXML and and XSD, an annotation-based approach. OpenXML compiles XML documents into DOM and XPath is used to decompose documents into tables. This approach is flexible but not scalable since the document must fit in memory. Besides, it requires the mapping to be defined through programming. The XSD approach, although based on annotations, is less expressive than *MDF*, *e.g.*, it does not allow mapping groups into a table. Furthermore, although SQL Server handles **E**dge, it does not allow to combine it with other mappings. Overall, the annotation proposed are less expressive than in *MDF*.

Oracle 9iR2 introduced Oracle XML DB [21] that provides a default mapping and allows users to customize it by annotating an XML Schema. In addition, Oracle also provides proprietary functions to shred and recover documents. Oracle has a fixed way of encoding document structure and does not support hybrid mappings. On the other hand, it allows mappings from elements to objects. We can easily add annotations that use these Oracle-specific features.

The schema adjunct framework described in [19] uses annotations to associate mappings to XML fragments. However, every entity in the XML Schema has to be explicitly mapped which might be tedious. In addition, it does not provide the flexibility provided in our system such as the ability to specify different mappings for document structure and the ability to express hybrid mappings.

Recently, MXM [2] has been proposed as a declarative mechanism to express XML-to-relational mappings. Our mapping specification shares the flexibility of MXM while having the advantage of using an XML Schema syntax.

Bourret et al [4] XML-DBMS, a generic tool for loading XML documents into relational tables. Although similar in motivation, there are important distinctions between our framework and XML-DBMS. The mappings supported by their tools are limited to the basic, shared, and hybrid techniques described in [16]. In contrast, *MDF* provides greater flexibility and is able to express a wide range of mapping techniques. In addition, an important contribution of our work is a comprehensive framework for representing mappings that can be used not only for shredding documents, but also for query translation.

7. DISCUSSION

In this paper, we presented a novel XML-to-relational mapping framework and established its utility for building applications that need to store and query XML data in relational databases. The framework defines a set of annotations that can be added to an XML Schema to define both which physical representations to use for the XML documents, as well as lower-level options. For example, as illustrated in Section 4.1, the union distribution operation can be represented as an annotation: by creating new groups for the elements within the choice construct. One may also use annotations to specify information at the field level, *e.g.,* **sqltype** to specify the SQL type of a field.

To the best of our knowledge, *MDF* is the first system that provides an end-to-end storage and querying solution for XML documents in relational databases. In addition, one of our main objectives in designing the annotations were to make the system flexible and able to represent many different mapping strategies. The availability of the mapping information coupled with the mapping API makes it easy for applications to use the mapping information. In fact, *MDF* can serve as a platform to implement and compare different mappings and query translation algorithms – until now, such a comparison has not been possible [11].

There is room for many improvements in *MDF*. In the short term, we are planning to include: support for target data models other than relational (*e.g.,* object-relational) and support for more complex correctness checks.

8. REFERENCES

[1] S. Amer-Yahia. Storage techniques and mapping schemas for XML. Technical Report TD-5P4L7B, AT&T Labs-Research, May 2003.

[2] S. Amer-Yahia and D. Srivastava. A mapping scheme and interface for XML stores. In *Proc. of WIDM*, 2002.

[3] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of ICDE*, pages 64–75, 2002.

[4] R. Bourret, C. Bornhvd, and A. P. Buchmann. A generic load/extract utility for data transfer between XML documents and relational databases. In *WECWIS*, pages 134–143, 2000.

[5] A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-structured data with STORED. In *Proc. of SIGMOD*, pages 431–442, 1999.

[6] Introduction to the dewey decimal classification. online computer library center. http://www.oclc.org/dewey/about/about_the_ddc.htm.

[7] F. Du. Translating xpath into sql. Technical report, OGI/OHSU, 2003. Available at `http://www.cse.ogi.edu/˜fangdu/xpath.html`.

[8] F. Du, S. Amer-Yahia, and J. Freire. XS: A generic schema based XML shredder. Technical report, OGI/OHSU, 2003. Available at `http://www.cse.ogi.edu/˜fangdu/xsreport.html`.

[9] D. Florescu and D. Kossman. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[10] IBM DB2 XML Extender. http://www4.ibm.com/software/data/db2/extenders/xmlext.html.

[11] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Proc. XSym*, 2003.

[12] P. Marron and G. Lausen. On processing XML in ldap. In *Proc. of VLDB*, pages 601–610, 2001.

[13] Microsoft support for XML. http://msdn.microsoft.com/ sqlxml.

[14] S. Paparizos and et al. Timber: A native system for querying XML. In *Proc. of SIGMOD*, page 672, 2003. Demonstration.

[15] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proc. of WebDB*, pages 47–52, 2000.

[16] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, pages 302–314, 1999.

[17] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and retrieval of XML documents using object-relational databases. In *Proc. of DEXA*, pages 206–217, 1999.

[18] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of SIGMOD*, pages 204–215, 2002.

[19] S. Vorthmann, J. Robie, and L. Buck. The schema adjunct framework. http://www.extensibility.com/resources/saf_dec2000.htm, Dec. 2000.

[20] Xerces Java parser 1.4.3. http://xml.apache.org/xerces-j.

[21] Oracle's XML SQL utility. http://technet.oracle.com/tech/xml/oracle_xsu.

[22] Xsl transformations (xslt). `http://www.w3.org/TR/xslt`.