A Comprehensive Solution to the XML-to-Relational Mapping Problem

Sihem Amer-Yahia
AT&T Labs – Research
180 Park Ave
Florham Park, NJ 07932, USA
sihem@research.att.com

Fang Du
OGI/OHSU
20000 NW Walker Rd
Beaverton, OR 97006, USA
fangdu@cse.ogi.edu

Juliana Freire*
OGI/OHSU
20000 NW Walker Rd
Beaverton, OR 97006, USA
juliana@cse.ogi.edu

ABSTRACT

The use of relational database management systems (RDBMSs) to store and query XML data has attracted considerable interest with a view to leveraging their powerful and reliable data management services. Due to the mismatch between the relational and XML data models, it is necessary to first shred and load the XML data into relational tables, and then translate XML queries over the original data into equivalent SQL queries over the mapped tables. Although there is a rich literature on XML-relational storage, none of the existing solutions addresses all the storage problems in a single framework. Works on mapping strategies often have little or no details about query translation, and proposals for query translation often target a specific mapping strategy. XML-storage solutions provided by RDBMS also have limitations. Notably, they are tied to a specific backend and use proprietary mapping languages, which not only may require a steep learning curve but often are unable to express certain desirable mappings.

In order to address these limitations, we developed *ShreX*, a XML-to-relational mapping framework and system that provides the first comprehensive solution to the relational storage of XML data. Mappings in *ShreX* are defined through annotations to an XML Schema. The use of XML Schema simplifies the mapping process, since it does not require users to master a new specialized mapping language. The use of annotations allows mapping choices to be combined in many different ways. As a result, *ShreX* not only supports all the mapping strategies proposed in the literature, but also new useful strategies that had not been considered previously. *ShreX* provides generic (and automatic) document shredding and query translation capabilities, and it is also portable — its mapping specifications are independent of the database backend.

Categories and Subject Descriptors

H.2 [**Database Management**]: Languages

*Current address: School of Computing – University of Utah, juliana@cs.utah.edu

WIDM'04, November 12–13, 2004, Washington, DC, USA.

General Terms

Management

Keywords

XML Storage, Mapping techniques, Relational Databases, XML Shredding

1. INTRODUCTION

As 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.

In order to store an XML document in a relational database, the tree-structure of the XML document must first be mapped into an equivalent, flat, relational schema. XML documents are then shredded and loaded into the mapped tables. Finally, at runtime, XML queries are translated into SQL, submitted to the RDMBS, and the results are then translated into XML.

There is a rich literature addressing the issue of managing XML documents in relational backends. Several mapping strategies (*e.g.*, [4, 9, 12, 22, 21]) and query translation algorithms (see [16] for a survey) have been proposed. In addition, support for XML storage is already available in most commercial RDBMSs. However, none of these solutions addresses all the storage problems in a single framework. Works on mapping strategies often have little or no details about query translation [16]; and proposals for query translation often target a specific and fixed mapping strategy. In addition, many of the available mapping solutions hard-code mapping choices [22]. As we discussed in [4], a given mapping strategy is unlikely to be the best choice for all applications — the ideal is to customize a mapping based on an application's characteristics, *i.e.*, its data and access patterns. Thus, hard-coding mapping choices can result in inefficient mappings. Although some of the solutions proposed by relational vendors do provide flexible mechanisms to define mappings, these solutions are proprietary and tied to a specific relational backend. This is a serious limitation. Since XML is widely used for data exchange, it is quite plausible that applications may need to store a given document (or different views of a document) in distinct database backends. Having to define distinct mappings, using different proprietary interfaces, is time-consuming and can add substantial costs to application development and maintenance.

ShreX (Shredding XML) [11] is a freely available system[1] that addresses many of the limitations of existing mapping solutions.

[1] *ShreX* is available at http://www.cse.ogi.edu/ShreX.

To the best of our knowledge, *ShreX* is the first system to provide a comprehensive solution to the relational storage of XML data. While designing *ShreX*, our goal was to build a system that was: *flexible*, *i.e.*, able to support a wide range of mapping strategies so that users could choose, depending on their application, how to map the XML data into relations; *portable* across multiple RDBMSs *i.e.*, a given mapping specification can be used for any database backend; *extensible i.e.*, allow the definition of new mappings; and *easy-to-use*, even for non-experts.

A key component of *ShreX* is its mapping definition framework. In *ShreX*, an XML-to-relational mapping is specified through annotations over an XML schema. In contrast to specialized mapping languages, which may require a steep learning curve, the use of XML Schema makes it easy to define mappings. Another advantage of using XML Schema is that mapping definitions can be validated *for free*.

Since annotations can be combined in many different ways, *ShreX* is able to express a wide range of mappings. Besides mapping strategies proposed in the literature and strategies supported by database vendors, new useful strategies that have not been previously considered can be easily defined in *ShreX*.

ShreX also provides generic (mapping-independent) functions for document shredding and query translation. This is made possible by an API which provides access to the mapping information. The use of annotations over an XML schema and an API to access mapping information makes the system extensible, since new annotations and new API functions can be added to support new mapping choices.

Outline and contributions. In this paper, we describe the *ShreX* system, its design and features. In Section 2, we give an overview of existing mapping techniques, and identify important requirements which guided our design. We present the architecture of *ShreX* in Section 3, where we describe the mapping framework and the implementation of the current prototype.

2. OVERVIEW OF MAPPING TECHNIQUES AND SYSTEMS

There is a substantial body of work on using relational databases to store XML documents. The various approaches differ in which meta-data they use (*i.e.*, schema or schemaless); how the relational configuration is generated; and which information is preserved on the relational side. Table 1 summarizes these differences, which are discussed below. In order to illustrate some of the techniques, we use the following example.

Example 2.1 (Mapping show data). *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 sample schema describes information about shows, where a* `SHOW` *has a* `TITLE`*, a* `YEAR`*, zero to ten* `AKA`*s (alternative titles), and a set of* `REVIEW`*s. A* `SHOW` *additionally has information about its* `BOXOFFICE` *and* `VIDEOSALES`*, if it is a movie, or information about* `SEASONS` *and* `EPISODES`*, if it is a TV show. The document illustrates the variability that the show schema allows.*

Schema-aware versus schema-oblivious. Mapping strategies can be broadly classified into schema-aware and schema-oblivious. Techniques which store XML documents in *generic* (pre-defined) relational tables are called schema-oblivious. One of the first proposals

```
<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="0" 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

```
<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

Techniques	Schema-aware	Cost-based	Constraint-preserving	Order-preserving	Automatic-generated	Backend
Stored [9]	No	No	No	Yes	Automatic	Relational
Edge [13]	No	No	No	Yes	Automatic	Relational
Interval [8]	No	No	No	Yes	Automatic	Relational
XRel [27]	No	No	No	Yes	Automatic	Relational
[23]	No	No	No	Yes	Automatic	Relational
[22]	Yes	No	No	No	Automatic	Relational
[20]	Yes	No	No	No	Automatic	Object-Relational
[17] [6]	Yes	No	Yes	No	Automatic	Relational
LegoDB [4]	Yes	Yes	No	No	Automatic	Relational
Oracle XML DB [16]	Yes	No	No	Yes	Automatic and customization	Object-Relational
DB2 XML Extender [14]	Yes	No	No	Yes	Manual	Relational
MS SQL Server [18]	Schema-aware and Schema-oblivious	No	No	Yes	Automatic and Customization	Relational

Table 1: Classification of the existing XML-to-Relational Storage Techniques

Source	Ordinal	Tag	Flag	Target
1	1	IMDB	Ref	2
2	1	SHOW	Ref	3
3	1	TITLE	String	4

Node	Value
4	Fugitive, The

Figure 3: Edge-based Relational Schema Design

for mapping XML documents was the Edge scheme [12], a schema-oblivious approach that explicitly stores all the edges in a document tree. Figure 3 illustrates a fragment of the relational tables obtained by applying the Edge mapping to the XML document given in 2. Other schema-oblivious techniques include [8, 27].

Departing from generic mappings, several specialized strategies have been proposed which make use of schema information to generate efficient mappings. Whereas Edge typically requires many joins for navigating and/or reconstructing the document, Shanmugasundaram et al [22] describe three specialized strategies which use schema information to minimize data fragmentation by *inlining*, whenever possible, the content of certain elements as columns in the relation that represents their parents. Figure 4(a) is an example of a relational configuration obtained by the shared inlining strategy proposed in [22].

Mapping primitives. Several techniques have been proposed which define a set of rules to map XML Schema primitives into their relational counterparts. For example, shared inlining specifies that elements which have multiple occurrences must be mapped into tables, whereas elements with a single occurrence should be mapped as a column of the table corresponding to its parent element. The LegoDB system [4] exploits a richer set of mapping primitives. In addition to parent-child relationships, LegoDB also also takes into account additional schema constructs such as choice and repetition, and it allows multiple mapping functions for a given construct. For example, besides the option to create a table for a set-valued element, LegoDB also considers inlining one or more occurrences of the repeated element within its parent (through the repetition split transformation). Figure 4 illustrates some of the relational configurations that can be generated by LegoDB for the schema of Figure 1. Note that while most techniques consider primitives that map XML constructs to pure relational systems, some [15, 20] leverage object-relational features of relational systems.

Fixed versus cost-based schema design. Most mapping strategies are fixed, *i.e.*, they fix the mapping function (see *e.g.*, [22, 12]). In contrast, LegoDB [4, 19] takes a cost-based approach to derive a mapping that best suits a given application — characterized by a schema, query workload and document samples. LegoDB uses the information in the XML schema to derive several possible mapping alternatives, and selects the one which leads to the lowest cost for executing a given query workload over sample documents.

Preserving order and structure. 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 4(a), a foreign key `parent_Show` is created in `TABLE Review` which refers to a record in `TABLE Show`. Sibling order can be captured 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, including `Edge` [12], as Figure 3 illustrates.

Different numbering schemes are possible for assigning ids to elements. Examples include Dewey and interval encoding. The Dewey Decimal Classification was originally developed for general knowledge classification [10]. 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. 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 [8], and is illustrated in Figure 5.

Schema-aware techniques [4, 6, 17, 20, 22] have focused on structural and constraint mapping, often ignoring the order among elements. Because these techniques ignore order, the resulting mappings are *lossy*. For example, the mapping strategies in [22] do not allow mapped documents to be faithfully reconstructed.[2]

[2]Note that that although [22] does keep unique keys for elements and ordinals for siblings, order information may be lost during the DTD simplification process.

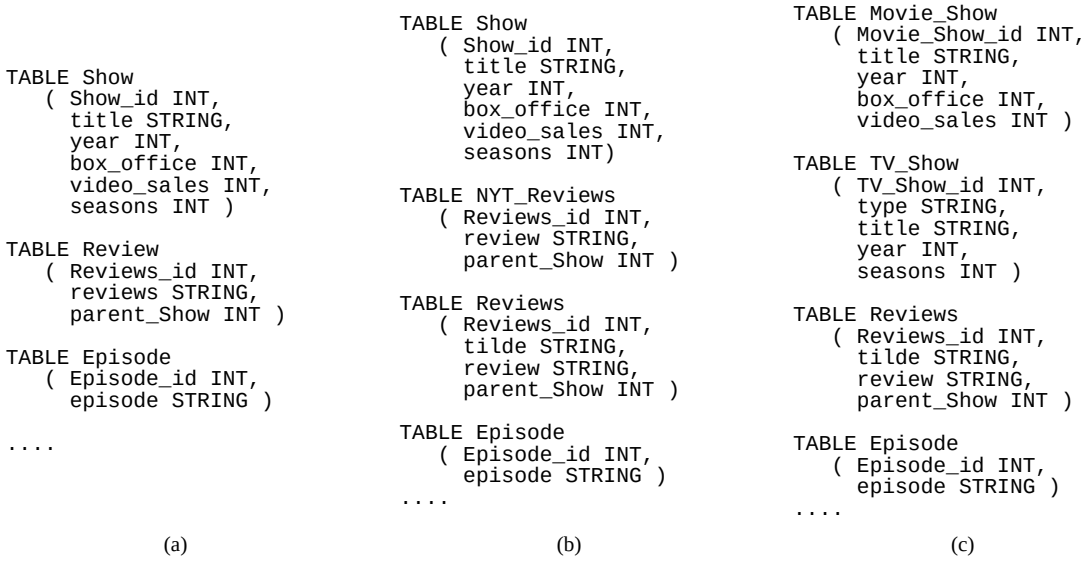

Figure 4: Three storage mappings for shows

It is worthy of note that capturing document order adds overheads to the mapping process, both with respect to storage use and query evaluation costs. Tatarinov et al [23] studied the performance implications of several techniques to maintain order information.

Node Label	Left-end point	Right-end point
...		
REVIEW	14	28
SUNTIMES	15	27
REVIEWER	16	19
”Roger Ebert”	17	18
RATING	20	23
”Two thumbs up!”	21	22
COMMENT	24	27
”This is a fun ...”	25	26
...		

Figure 5: Interval-based Relational Schema Design

Automatic-generation versus Manual-specification. All mapping techniques proposed in the research literature provide an automated means to derive mappings. Commercial products [14, 18] allow users to manually specify mappings between XML and relational schemata. Languages such as XSLT [25], XQuery [3], and IBM's DAD [14] can be used to define mappings that perform arbitrary transformations over an XML document. Although these approaches allow substantial flexibility in creating mappings, they have drawbacks. Since there may be many different ways to map a given document (or schema) into relations, in order to design a *good* mapping, a developer must have a good understanding of both XML and relational technologies. Especially for large schemata, it can be tedious to write these specifications, and more importantly, to verify whether they are *correct* (*e.g.*, whether all elements are mapped). Also, since arbitrary 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 be needed on a per-application basis.

Discussion. As discussed above, there are many different ways of mapping XML documents into relational tables. Different approaches use different means to capture element identity, document structure, and order. A flexible mapping middleware system, that is able to represent all different mappings, must be able to support multiple choices for these dimensions.

Another important requirement for a mapping system is the ability to support all mapping tasks, *i.e.*, schema design, shredding and loading, and query translation. Most techniques proposed in the literature focus on a specific task. In addition, proposed systems often hard-code shredding and query translation for a specific mapping; making it hard not only to re-use them, but also to compare different approaches for these tasks.

3. THE SYSTEM

A key component of *ShreX* is the annotation-based framework to define mappings: a mapping is expressed by annotating an XML schema. The various mapping dimensions were taken into account in the design of *ShreX* annotations. As a result, different mapping choices (*i.e.*, different ways to capture structure and order) can be easily combined to create new mapping strategies. The use of annotations also makes the system extensible, as new annotations can be added to express new mapping choices; and portable, since the mapping definition is independent from the underlying relational database.

ShreX also provides an API to access mapping information. This API allows the design of generic functions for the mapping tasks, *i.e.*, functions that are not tied to the specifics of a particular mapping strategy.

In what follows, we describe the architecture of *ShreX* and explain how the mapping specification is used to validate mappings, shred and load XML documents, and translate XML queries to SQL (Section 3.1). We then describe how mappings are defined in Section 3.2.

3.1 Architecture Overview

The architecture of *ShreX* is shown in Figure 6. Users can either manually annotate an input schema, or use the interface provided by the system. The *annotation processor* parses an annotated XML schema, checks the validity of the mapping and creates the corresponding relational schema. In addition, the mapping information is made persistent in the *mapping repository*. The *document shred-*

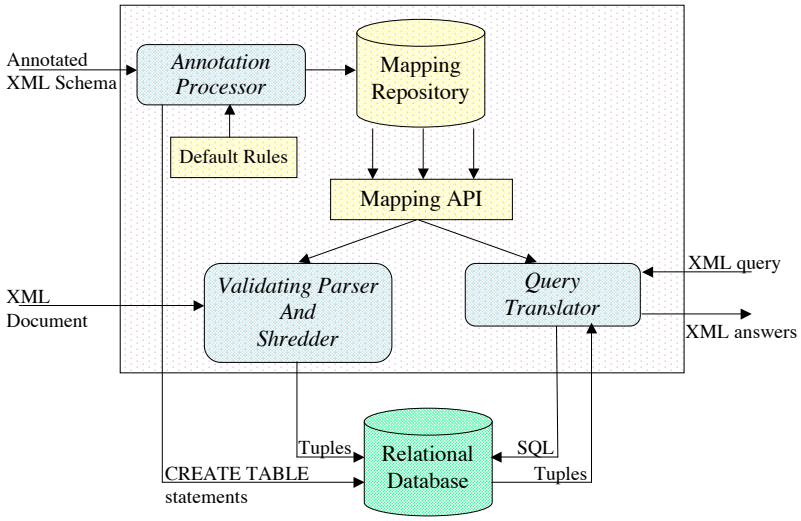

Figure 6: *ShreX* Architecture

der accepts as input a document, and uses the mapping API to access the information in the mapping repository in order to generate the tuples and load them into the relational database. The mapping repository is also accessed by the *query translator*, which generates SQL queries from XML queries.

User Interface. *ShreX* provides a graphical user interface that helps users define and customize mappings. The interface displays the XML schema and corresponding relational tables, allowing users to visually check the connections between the XML elements and their relational counterparts, as well as interactively modify the mapping specification.

Annotation Processor. This module is in charge of parsing an annotated XML schema, checking the validity of a mapping, generating a mapping repository and producing the `CREATE TABLE` statements necessary to construct the relational schema. In order to check the validity of a mapping, the annotation processor validates the input (annotated) schema against an XML schema for annotations [1]. The current version of ShreX supports simple validity checks such as verifying whether annotations are attached to the appropriate elements, and whether table and attribute names are unique in the mapping definition. Additional checks are possible, for example, verifying whether a mapping is *lossless — i.e.*, whether the document can be reconstructed from the mapped tables.

Writing an annotation for every element and attribute definition in an XML schema can be tedious, especially for large schemata. Thus, *ShreX* provides a set of default rules that is used to *complete* mapping specifications. For example, single-occurrence elements such as show `TITLE` are inlined by default. Users can define new rules, and obviously, override existing default rules by adding a specific annotation to the input XML schema.

Mapping Repository and API. The mapping information extracted by the annotation processor is stored in a database — the mapping repository. Making this information persistent avoids the need to 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. *ShreX* provides an API to the mapping repository that allows access to information such as: how elements and attributes are mapped

isTable(ElemName|AttName),

which mapping is used to capture document structure

getStructureScheme(),

and which tables are available in the relational schema:

getTableInfo(TableName).

Table 2 summarizes the functions provided by the API. This API allows users to write mapping-independent code which works regardless of the specific features of a particular mapping.

Document Shredder. The shredder is in charge of generating tuples, field values and CLOBs from an input document. It piggybacks on a standard XML parser. While a document is parsed, the shredder uses the mapping API in order to retrieve information about how a particular element or attribute is mapped, and generates the appropriate tuples accordingly.

ShreX uses the SAX interface of Xerces [24], which is both efficient and scalable. In the current implementation, while the document is parsed, the tuples are written to a file. After parsing is finished, the output file is bulkloaded into the relational backend. Even using this naïve implementation, the system is able to shred and load a 1GB document into DB2 in less than 30 minutes. It is worth pointing out that significantly smaller files cannot be loaded using some commercial solutions [26]. Note that the shredded 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.

Query Translator. In the current implementation, the query translator (to SQL) supports a subset of XPath that includes child and descendant axes and position-based predicates. Similar to the document shredder, the query translator does not hard-code mapping choices, instead it uses the information provided by the mapping API to dynamically decide how to perform the translation.

Database API. *ShreX* mappings specifications are portable and independent from any particular database backend. However, lower-level functions must deal with the peculiarities of different database systems. For example, different databases provide different bulk-loading options and commands, and *ShreX* must be able to invoke these commands. We have designed a generic yet simple database JAVA programming interface which allows users to hook an RDBMS to *ShreX* by implementing the functions in the interface. Plug-ins for DB2, Oracle and MySQL are available in the current release of *ShreX*.

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 mapped to a table.
isField	attribute or element name	true, false	determines whether the input has been mapped to a field.
isEdge	element name	true, false	determines whether the input has been mapped using edge-mapping.
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.
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.

Table 2: Main API Functions

3.2 Mapping Definition

Mappings are expressed by annotating an input XML schema. Annotations define how a given XML fragment should be mapped into the relational model. Annotations are expressed using attributes from a namespace called *shrex*, and can be associated to attributes, elements and groups. The annotation attributes supported by *ShreX* are shown in Table 3. Figure 7 illustrates the use of some of the annotations (shown in boldface). The use of a specific namespace for annotations helps separate the validation of an input document against its XML schema from the validation of the mapping specification.

Mapping identity, structure and order. An important aspect of a mapping is how it captures element identity, document structure and order. In *ShreX*, the choice of structure mapping can be specified through the **structurescheme** attribute (see Table 3). For example, in Figure 7, the structure scheme selected for the document is Dewey (see annotation in the root element). Other supported schemes include: key-foreign-key for parent-child relationships and ordinal for siblings ("KFO") [4]; and interval encoding [8]. The ability to define multiple document structure schemes is a feature that is unique to *ShreX*.

```
<element name="SHOW"
      shrex:structurescheme="Dewey" />
  <sequence>
    <element name="TITLE" type="string"
        shrex:outline="true"
        shrex:tablename="Showtitle"/>
    <element name="YEAR" type="integer"
        shrex:outline="false"
        shrex:columnname="Showyear"
        shrex:sqltype="NUMBER(4)"/>
    <element name="REVIEW" type="ANYTYPE"
         minOccurs="0" maxOccurs="unbounded"
         shrex:edgemapping="true"/>
    <element name="AKA" type="string"
         minOccurs="0" maxOccurs="unbounded"/>
  <sequence>
</element>
```

Figure 7: Annotated movie schema

Outline, tablename, columnname, sqltype. Annotations are used to specify how individual elements and attributes in a document are represented in the relational schema. Figure 8 shows the relational configuration for the annotated schema of Figure 7. The annotation **outline="true"** in the element `TITLE` indicates that it should be mapped to a separate table; and the annotation **tablename** specifies that this table should be named **Showtitle**. The element `YEAR`, on the other hand, has its **outline** attribute set to false, consequently it is inlined in the table corresponding to its parent element, `SHOW`. The annotations **sqltype** and **columnname** in the `YEAR` element specify that it should be mapped to a column named **Showyear** and SQL type **NUMBER(4)**. Although not illustrated in the example, an element can also be mapped into a CLOB, using the annotation **maptoclob**.

```
TABLE SHOW( ID VARCHAR(128),
Showyear NUMBER(4))

TABLE Showtitle( ID VARCHAR(128),
  ParentID VARCHAR(128), TITLE VARCHAR(512))

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

TABLE AKA( ID VARCHAR(128),
  ParentID VARCHAR(128), AKA VARCHAR(512))
```

Figure 8: Relational configuration for movie schema

Mapping schemaless documents. The use of annotated schemata in *ShreX* does not preclude the system from expressing generic (schemaless) mappings. For example, in Figure 7, the annotation **edgemapping="true"** in the element `REVIEW` indicates that `REVIEW` and its descendants are mapped using Edge mapping [13], *i.e.*, a single table to store all the `REVIEW` elements and contents. This functionality is specially useful to map elements whose structures are not known in advance, such as elements of type `ANYTYPE`.

Annotations naturally allow the definition of mappings that combine different mapping strategies. Note that, in this example, part of the document is mapped using a generic mapping – Edge, and part is mapped using a schema-aware strategy.

Transformation-based mappings. Additional mapping strategies are supported by combining annotations with the schema transformations proposed in [4]. For example, if repetition split is applied to `AKA` in the original schema, *i.e.*, `AKA*` → `AKA?, AKA*`, the first occurrence of `AKA` could be inlined in the table `SHOW`:

```
TABLE SHOW( ID VARCHAR(128),
            Showyear NUMBER(4),
            AKA VARCHAR(512))
```

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 [13].
maptoclob	attribute or element	true, false	If value is true, the element or attribute is mapped to a CLOB column.

Table 3: Annotation Attributes. Each row in the table contains an annotation attribute, its target (*i.e.*, element, attribute, and group to which it applies), its possible values and effect.

Union distribution [4] is an example of another mapping choice that can be expressed by a combining schema transformation and *ShreX* annotations. Recall that in the IMDB schema of Figure 1, a `SHOW` may be either a movie or TV show. By creating two new groups, one for movies and one for TV shows, and indicating that each group should correspond to a table, we can generate the configuration shown in Figure 4(c).

Mapping Expressiveness. In what follows, we show a few examples to illustrate how our mapping definition framework expresses existing schema-based mapping techniques. In particular, we describe how the strategies proposed in [22] can be expressed using *ShreX* annotations. The sample schema describes information about `Movie` and `TV` elements, where both have a `TITLE` element.

```
<element name="Movie">
  <sequence>
    <element name="TITLE" type="string"
        shrex:tablename="MovieTitle"/>
  <sequence>
</element>
<element name="TV">
  <sequence>
    <element name="TITLE" type="string"
        shrex:tablename="TVTitle"/>
  <sequence>
</element>
```

Figure 9: Schema Annotations equivalent to Basic Inlining

The *basic inlining* technique creates a relation for every element in the input schema. Figure 9 illustrates the annotations (shown in boldface) which correspond to basic inlining. The annotation **tablename="MovieTitle"** in the element `TITLE` inside `Movie` indicates it should be mapped to a separate table with name **MovieTitle**. Similarly, The annotation **tablename="TVTitle"** in the element `TITLE` inside `TV` indicates that a table **TVTitle** should be generated for it. As a result, a table is created for each element defined in the XML schema:

```
TABLE Movie( ID VARCHAR(128))
TABLE MovieTitle( ID VARCHAR(128),
  ParentID VARCHAR(128), TITLE VARCHAR(512))
TABLE TV( ID VARCHAR(128))
TABLE TVTitle( ID VARCHAR(128),
  ParentID VARCHAR(128), TITLE VARCHAR(512))
```

The *shared inlining* technique identifies the elements that have multiple parents, such as the `TITLE` element, and uses a single table to store the different instances of this element. Figure 10 illustrates the annotations which generates a shared-inlining relational configuration. The annotations in both `TITLE` elements set the table name to **Title**, implying that they will be mapped to the same table. The resulting relational configuration is as follows:

```
TABLE Movie( ID VARCHAR(128))
TABLE TV( ID VARCHAR(128))
TABLE Title( ID VARCHAR(128),
  MovieID VARCHAR(128), TVID VARCHAR(128),
  TITLE VARCHAR(512))
```

```
<element name="Movie">
  <sequence>
    <element name="TITLE" type="string"
        shrex:tablename="Title"/>
  <sequence>
</element>
<element name="TV">
  <sequence>
    <element name="TITLE" type="string"
        shrex:tablename="Title"/>
  <sequence>
</element>
```

Figure 10: Schema Annotations equivalent to Shared Inlining

The *hybrid inlining* technique further inlines the `TITLE` element and reduces the generated relational tables to two. Hybrid inlining corresponds to the default mapping rules used in *ShreX*, hence no annotation is needed – the input schema, as is, produces the following relational configuration:

```
TABLE Movie( ID VARCHAR(128), TITLE VARCHAR(512))
TABLE TV(ID VARCHAR(128), TITLE VARCHAR(512))
```

4. DISCUSSION

To the best of our knowledge, *ShreX* is the first comprehensive system for managing (*i.e.*, shredding, loading and querying) XML documents in relational databases. *ShreX* has several novel features including the ability to define mixed-mapping strategies and to specify a document structure scheme. The implementation of *ShreX* is modular and easily accommodates changes and extensions to the system. For instance, if a new mapping technique is developed and users want to incorporate it into *ShreX*, it suffices to add a few annotation attributes and extend the annotation processor to handle the new attributes.

ShreX makes it simple to study and compare the performance of multiple query translation techniques using different mapping strategies. By making the source code available, we hope *ShreX* will serve as a platform to develop and evaluate new mapping strategies, query translation and optimization algorithms.

Our preliminary experiments show that, for shredding and loading XML documents, *ShreX* is highly scalable — it is able to shred very large documents, and reasonably efficient (even with the current naïve implementation). As future work, we intend to do a detailed performance analysis for query translation.

A problem that is orthogonal to designing mapping strategies is the translation of constraints in the document schema to the corresponding relational schema. For instance, the propagation of keys [7] and functional dependencies [5] have been studied. Besides capturing the semantics of the original document schema, these techniques have been shown to improve the mappings by, *e.g.*, reducing the storage of redundant information [5]. Constraints are also important when updates are allowed over the mapped relations. Barbosa et al [2] defined sufficient conditions for guaranteeing that mappings preserve enough information so that the mapped document remains valid in the presence of updates. We plan to provide a constraint translation module and support for updates in a future release of *ShreX*.

Acknowledgments. The National Science Foundation partially supports Juliana Freire under grant EIA-0323604.

5. REFERENCES

[1] S. Amer-Yahia, F. Du, and J. Freire. A generic and flexible framework for mapping XML documents into relations. Technical report, OGI/OHSU, 2004.

[2] D. Barbosa, J. Freire, and A. Mendelzon. Information preservation in XML-to-relational mappings. In *Proceedings of the International XML Database Symposium (XSym)*, pages 66–81, 2004.

[3] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C Working Draft, July 2004.

[4] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 64–75, 2002.

[5] Y. Chen, S. Davidson, C. Hara, and Y. Zheng. RRXS: Redundancy reducing XML storage in relations. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

[6] Y. Chen, S. B. Davidson, and Y. Zheng. Constraints preserving schema mapping from XML to relations. In *Proceedings of the Workshop on Web and Databases (WebDB)*, pages 7–12, 2002.

[7] S. Davidson, W. Fan, C. Hara, and J. Qin. Propagating XML constraints to relations. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2003.

[8] D. DeHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval coding. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003.

[9] A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-structured data with STORED. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 431–442, 1999.

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

[11] F. Du, S. Amer-Yahia, and J. Freire. ShreX: Managing XML documents in relational databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

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

[13] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML in a relational database. Technical Report 3680, INRIA, 1999.

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

[15] M. Klettke and H. Meyer. XML and object-relational database systems - enhancing structural mappings based on statistics. In *Proceedings of the Workshop on Web and Databases (WebDB)*, pages 63–68, 2000.

[16] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Proceedings of the International XML Database Symposium (XSym)*, 2003.

[17] D. Lee and W. W. Chu. Constraints-preserving transformation from XML document type definition to relational schema. In *International Conference on Conceptual Modeling (ER)*, 2000.

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

[19] M. Ramanath, J. Freire, J. Haritsa, and P. Roy. Searching for efficient XML-to-relational mappings. In *Proceedings of the International XML Database Symposium (XSym)*, 2003.

[20] K. Runapongsa and J. M. Patel. Storing and querying XML data in object-relational DBMSs. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2002.

[21] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proceedings of the Workshop on Web and Databases (WebDB)*, pages 47–52, 2000.

[22] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[23] I. Tatarinov, S. Viglas, K. Beyer, J. S. m, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–215, 2002.

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

[25] XSL transformations (XSLT). http://www.w3.org/TR/xslt.

[26] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 621–632, 2004.

[27] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. In *ACM Transactions on Internet Technology*, pages 110–141, 2001.