Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Tomáš Kohan

User-Defined XML-to-Relational Mapping

Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková
Field of Study: Informatics

I would like above all to thank my supervisor, RNDr. Irena Mlýnková, for her guidance in shaping of my research, this continuous encouragement and major advices.
I am very grateful to all the people who supported my work by their help and advice too.

Contents

List of Figures

Název práce: Uživatelsky definované mapování XML dat do relačních databází
Autor: Tomáš Kohan
Katedra (ústav): Katedra softvérového inženýrství
Vedoucí bakalářské práce: RNDr. Irena Mlýnková
e-mail vedoucího: irena.mlynkova@gmail.com

Abstrakt: V předložené práci studujeme možnosti mapovaní XML dat do relačních systémů. V první části popíšeme základní termíny používané v práci a následně také základní techniky pro mapování XML dat do relačních databází. V další části se budeme zabívat teoretickými metodami jako MXM a ShreX, které byly navrženy na akademické půde. V třetí části popíšeme metody mapování, které jsou použité v některých komerčních systémech jako Oracle, DB2 a MS SQL. V druhé polovině práce navrhneme novou mapovací metodu (*XRM*), která přinese několik nových vlastností s tím, že původní pozitivní vlastnosti zústanou zachovány. V záveru rozebereme prototypovou implmentaci navrhované mapovací metody.

Klíčová slova: XML úložiště, mapování, XML do Relací, uživatelsky definované

Title: User-Defined XML-to-Relational Mapping
Author: Tomáš Kohan
Department: Department of Software Engineering
Supervisor: RNDr. Irena Mlýnková
Supervisor's e-mail address: irena.mlynkova@gmail.com

Abstract: In the present work we study opportunities of mapping the XML data into relational systems. In the first part we describe basic terminology used in this work and subsequently also basic techniques for mapping XML data into the relational database. In the next part we engaged in theoretical methods like MXM and ShreX, which were proposed on premises of a university or by a research group. In the third part we describe mapping methods, that are used in some commercial systems like Oracle, DB2 and MS SQL. In the whole second half of this work we propose a new mapping method (*XRM*), which bring in several new features, while the origin positive features are kept. At the end we analyze the prototype implementation of the proposed mapping method.

Keywords: XML Storage, mapping, XML-to-Relation, user-defined

Chapter 1

Introduction

Data storage, its exchange and effective processing are key areas of the modern age. It is natural to use opportunities of computer technology and computer networks for data management and communication (primarily the worldwide network the Internet). There were proposed many methods and languages for presenting, sharing and exchanging information in the past. As an example can be considered HTML[1] language. This language can be used to present information especially through web server and the HTTP[2] protocol. In the process the HTML language tends to be used particularly to describe visual presentation of a document. Semantic elements of the document structure became weaker. The expansion of computer engineering, especially of internet network, and its extending among continuously growing base of users begin enormous growth of number of documents using this language. In addition, users' visual form requests were growing along with opportunities of displays, and therefore majority of these documents became more complicated, less transparent and the amount of useful information in proportion to the whole document size was still descending. The most important disadvantage of HTML language for needs of electronic communication and information management is fixed set of verbalization instruments and therefore low extensibility and flexibility for particular usage.

The XML[3] language was created as an reaction to this needs and insufficiencies. The XML language is a subset of SGML[4] language as well as HTML language. Contrary to SGML the XML language is designed with regard to simplicity. This language is easy to learn for users and also electronic processing is easy and effective. Therefore XML documents can be processed in mobile devices and other devices with restricted calculation capacity. On the other hand, the XML language is focused especially on information structure description and their semantic meaning. Since

[1]HyperText Markup Language - http://www.w3.org/TR/REC-html40/
[2]HyperText Transfer Protocol
[3]eXtensible Markup Language - http://www.w3.org/TR/REC-xml
[4]Standardized General Markup Language

it is not possible to design an common language for semantic description of information from all fields of human activity, the XML language was created as an extensible standard. Every user can create instruments for description of information from each other's field (problem domain). However, two subjects can communicate and exchange information between them only by keeping joint document structure and joint set of means of expression. The XML Schema is one of them. The document schema not only support communication between subjects, but also supports much effective storing of these documents, eventually their compression or information look-up in the data bases founding on the XML language.

However, XML documents storage as files on file system and subsequent looking up in these documents is rarely powerful enough. File opening and random access to a certain part of this file is considered to be the slowest operation in computer engineering. That is just the reason why there were some different approaches to propose new methods. One of these is using an relational system as a storage backend. It combines database performance which is developing for a long time and simplicity, flexibility, human readability and exchangeability of XML documents. To take advantage of relational database, the XML document has to be suitably mapped onto relations in database. This mapping can be done using several methods. There are two groups of these methods. One group maps XML documents which do not have assigned an document with their structure and the second one uses this structure information. First group uses only generic mapping techniques while it do not know the structure of documents which are stored in the system. The later one can take an advantage by using some special mapping techniques.

1.1 Goals

This diploma thesis has two main goals. The first one is to summarize some mapping techniques and mapping algorithms. As well as the theoretical ones, also commercial ones. The second goal is to propose a new mapping technique which should group some advantages of current techniques and add some new interesting features. This proposal is then implemented as a prototype which shows the basic functionality of proposed method. The implementation uses already implemented libraries for work with XML documents and database connections.

1.2 Thesis overview

In the introduction there is described the motivation for this thesis and there are proposed goals which should be reached. In the second chapter there are defined basic terms and technologies used in this book. Especially the XML language, XML Schema language for document structure description and the basics of relational databases. In the third chapter there are described mapping basics and several storage techniques. This techniques are used by mapping techniques which are described in chapter four (theoretical) and five (commercial). In the sixth chapter there is proposed a new mapping technique. The prototype implementation is described in the seventh chapter. The eight chapter describes experimental results and practical previews of proposed mapping technique. The last chapter contains final conclusion and opportunities of further improvement. At the very end of this thesis are several appendices which describes technical details of prototype implementation.

Chapter 2

Used technologies

In this chapter some basic technologies are introduced. It is necessary to handle this problems to continue to next chapters of this book. As the main topic is mapping XML to relational systems, sections describing XML and relational systems can be found here. Most of these techniques use XML Schema to define XML documents structure and additionally to define mapping specification, thus XML Schema is also mentioned.

2.1 XML

Format XML (eXtensible Markup Language[1]) was defined by W3 Consortium[2] as a format for transmission of general documents (data). The design came from older and more general SGML[3] standard. Documents in XML are automatically SGML documents too. The advantage of this format is simplicity, strict syntax and flexibility. It is intended to be used for publishing and exchanging data but also for its storing and searching.

The set of XML tags is not fixed. It can be additionally defined for different collections of documents differently. Definition of the set of XML tags can be included in XML document, referenced by a link or agreed in advance. The XML tags have a form of common parenthesis:

```
<name>John</name>
```

Elements

Tags in XML have markup function for certain parts of a document. Usually, tag is composed of two parts - *start-tag* (`<name>`) and *end-tag* (`</name>`). These tags enclose an *element*. Tags enclosing an empty string form an *empty-element*, e.g. `<name/>`. This implies that the element may

[1]http://www.w3.org/TR/REC-xml/
[2]World Wide Web consortium - http://www.w3.org
[3]Standard Generalized Markup Language - ISO 8879

be formed by two corresponding tags (and may enclose some content as string or other elements) or by one azygous tag (and therefore cannot contain content). Paired tags can be compared with common mathematical parenthesis as they have to be well-parenthesized. Every XML documents have to have single root element. The XML document which satisfy all these rules is called *well-formed*.

Moreover if the document has assigned also description of its structure, it is called *valid* or more precisely the document is valid against a given schemata, as long as its structure and content conform to this schemata. The structure of an XML document describes its feasible element names, content, attributes and permitted nesting of elements. This structure can be described by languages as DTD or XML Schema. The latter one is described later on.

Attributes

Every element can optionally contain attributes. Attributes are part of the element tag. In case of paired elements they are part of the opening tag. Attributes have a format of `name="value"` where the value is always closed in quotation marks.

```
<person id="25">
    <name>John</name>
    <surname>Neumann</surname>
    <scientist salary="80000"/>
</person>
```

Document schemata define not only which attributes are allowed but also their optionality and in case of some languages (for example XML Schema) also data type of the attribute.

Processing instructions

Processing instructions are instructions for superior application which processes the document. They have no meaning for the document data, they only influence the way the document is processed. Processing instructions are included in the document in the format

`<?identificator data?>`.

The most commonly used instruction for document processing is the XML document declaration. This instruction should be used in the very beginning of the XML document. Its format is

`<?xml version="1.1" encoding="UTF-8" standalone="no"?>`.

The first attribute defines the language version (nowadays it is version 1.1), second attribute `encoding` contains information about character set used in the document and the last attribute defines whether the document can be processed separately or it demands external definitions.

Entities

Entities are substituting artifacts of the XML language which are substituted by the content during the processing. For example, if very long name of organization is often used in XML documents, an entity name for the organization name can be defined and `&orgName;` can be used in document instead. This entity would be substituted for defined text at every place it is found. Entities format is `&name;&`. The definition looks like `<!ENTITY name "text">`.

In XML there are three predefined entities for the character which have special meaning. They are `&lt;` for <, `&gt;` for > and `&amp;` for &.

Comments

Comment is a text enclosed between strings `<!--` and `-->`. Everything that is between these two tags is immune to process.

CDATA

Inside CDATA section the document processing does not proceed and the content of this section is passed to output without processing. Within this section the characters <, > and & lose their special meaning. The section is enclosed into tags `<![CDATA[` and `]]>`. This XML fragment is suitable in case that XML document contains large text which should not be proceeded (it contains special characters, but with entirely different meaning).

Tho more detailed description of XML language can be found on the site of W3 Consortium [20].

2.2 XML Schema

XML Schema is language for describing structure of XML document. It is possible to describe valid elements, for each element valid attributes (incl. optionality and data type) and parent-child relationships. It enables creation of user-defined data types, defines order of subelements and also repetition rate of subelements. It provides the possibility to define implicit values. As well as XML language, the XML Schema language is also standardized by W3 Consortium.

Namespaces

A single document can contain tags from several schemata. In order to avoid complicated agreeing between schemata authors on naming and to avoid collisions of elements and attributes names, there were created so-called namespaces. Each schema defines elements belonging to explicit namespace. Within one namespace all names have to be unique (attributes names only in the context of own element indeed). Every namespace is uniquely identified by its URI[4]. Namespace of the element and all its subelements can be defined by the attribute `xmlns:prefix="URI of namespace"`. To use an element belonging to the specific namespace is used `<prefix:local-name-of-element .....>`, to use an attribute is used `prefix:local-name-of-attribute="value"`. The implicit namespace can be defined in every XML document. It is defined without the prefix and it is not necessary to write the prefix in front of neither elements nor attributes. Implicit namespace can be defined by `xmlns="URL of implicit namespace"`.

Assigning schema to document

The schema can be assigned to the document by using the attribute `schemaLocation` in the element definition where the namespace is defined. The content of the attribute `schemaLocation` is space-separated list of pairs `URI-of-namaspace schema-location`. Items of the pair are also separated by space. If there is no namespace defined for the schema, it is assigned to the document by attribute `noNamespaceSchemaLocation`, whose content is the schema location. Every document can have assigned only one schema without specified namespace (otherwise name collisions could come up). All names from schema without namespace assigned are written without prefix, thus they belong to implicit namespace.

Syntax

XML Schema language is the specific usage of XML language. For the description of schema of XML document therefore is no need to bring about other language. Naturally there exists the structure definition for the XML Schema language XML documents written in XML Schema language. As XML Schema document is a XML document it has one root element, namely element `schema`. As well as all the other elements of XML Schema document, this element is also from namespace with URI `http://www.w3.org/2001/XMLSchema`.

[4]Uniform Resource Identifier

Elements

Available elements in the document are defined by tag `element`. If the tag `element` is the lineal descendant of the element `schema`, it defines global element. Global element is accessible from whole schema and it can be used as root element of the document. If the tag `element` is used inside the definition of complex type, it defines local element. Contrary to DTD XML Schema allows to define several different root elements. Attribute `name` defines the name of the element. The most important definition within element is the definition of its type. Type can be either simple or complex. Type can be defined either by using attribute `type` or as nesting element.

```
<xs:element name="person">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="name" type="TName" />
         <xs:element name="surname" type="TSurname" />
         <xs:element name="scientist" type="TScientist" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Attributes

Every element can have defined a set of available attributes. The individual attributes are defined by the tag `attribute`. The name of the attribute is defined by attribute `name`. The attribute occurrence is modified by attribute `use`. It can take the value optional, prohibited or required. As well as elements, attributes have to have specified data type. It can be specified by attribute `type` or by specifying type as nested element. Attribute `ref` can optionally refer to the globally defined attribute and the attribute `default` specify implicit attribute value.

```
<xs:complexType name="TScientist" >
   <xs:attribute name="salary" type="xs:positiveInteger" />
</xs:complexType>
```

Simple types

Simple data type cannot contain elements or attributes. The basic data type in XML Schema is type string. XML Schema allows to derive type by way of restricting the superior type. Simple types can be divided into two groups - built-in and user-defined.

Complex types

Complex type is defined by element `complexType`. Complex type contains a set of attribute definitions and definitions of simple or complex content. The simple content is restriction of simple type or extension of simple type by attributes. Complex content is restriction or extension of complex type.

- ➢ **Sequence** is a data type which selects a sequence of items in fixed order, therefore all the items have to occur (depending on their own repetition rate) and in fixed order exactly as they occur in element `sequence`. The content of the element `sequence` can be `element`, `choice`, `sequence` or `all`.
- ➢ **Choice** is a data type for exclusive selection of item from a set. Therefore the result is exactly one item from a set. The content of the element `choice` can be `element`, `choice`, `sequence` or `all`.
- ➢ **All** is a data type for occurrence of all elements from a set. Each element has to occur exactly once. However, the order of elements is not fixed. This element is an extension compared to DTD and can significantly help to make schemata more readable and transparent. The content of element `all` can be only the element `element`. This restriction is accepted to ensure deterministic data model.

For example, the type

```
<xs:all>
   <xs:element name="A" type="TA" />
   <xs:element name="B" type="TB" />
</xs:all>
```

corresponds with the sequence of elements

```
<A /><B /> and <B /><A />
```

Other features of XML Schema

Other features of XML Schema language like substitutions, groups, u-nique values of attributes, keys and foreign keys, tag for any element or attribute, notation and others, are not important in scope of this book. More detailed specification can be found on site of W3 Consortium [21].

Comparison with DTD

In this section important improvements XML Schema language against DTD are highlighted. The DTD provides basic grammar for defining an

XML Document in terms of the metadata that comprise the shape of the document. An XML Schema provides this, plus a detailed way to define what the data can and cannot contain. It provides far more control for the developer over what is legal, and it provides an Object Oriented approach with all the benefits this entails.

2.3 Relational system

A relational database is a database that conforms to the relational model, and could also be defined as a set of relations or a database built in an Relational database management system.

A relational database management system (RDBMS) is a system that manages data using the relational model. Frequently, the term "RDBMS" is inaccurately used as a generic label for the relational database concept. Most current RDBMSs (for example: Oracle, Microsoft SQL Server, DB2, MySQL, PostgreSQL) deviate significantly from the relational model and are more accurately called SQL database management products (DBMS).

Strictly, a relational database is a collection of relations (frequently called tables). Other items are frequently considered as part of the database, as they help organize and structure the data, in addition to forcing the database to conform to a set of requirements.

Relations or tables

A relation is defined as a set of tuples that all have the same attributes. This is usually represented as a table, which is data organized in rows and columns. In a relational database, all of the data stored in a column should be in the same domain (i.e. data type). In the relational model, the tuples should not have any ordering. This means both that there should be no order to the tuples, and that the tuples should not impose an order of the attributes. Put differently, neither the rows nor the columns should have an order to them.

While this is the desired result, it is not universally achieved. The SQL standard requires columns to have a defined order. All data stored in a computer has to have an order, as the memory of a computer is linear. Also, when the data is returned, there must be an order in which the data is returned (because all transfer protocols are linear, and coincidentally enough, humans read in a linear fashion). The point here is that this order must never make a logical difference in the system. Frequently orders are imposed, which impact performance, but they should never change the result of a query on the database. In practice, several of the DBMSs that are considered "relational" impose an order that makes a logical difference.

Constraints

Constraints are a way of providing restrictions on the kinds of data that can be stored in the relations. These are usually defined (formally) in the form of expressions that result in a boolean value, indicating whether or not the constraint holds. Constraints are a way of implementing business rules into the database.

Under the strictest sense, constraints are not considered a part of the relational database, but because of the integral role, which they play in organizing data, they are usually considered a part of the database.

Data domain

A data domain (or usually just domain), is a set of possible values for a given attribute. Because it does constrain the values the data can hold, it could be considered as a constraint, but because attributes must specify a domain, it could just be considered a part of the relation's definition. Mathematically, a domain can be expressed as "all values for this attribute must be an element of the specified set."

Keys

A tuple usually represents some object and its associated data, whether that object is a physical object or a concept. A key is a kind of constraint which requires that the object, or critical information about the object, is not duplicated. For example, a family might like to have a constraint such that no two people in the immediate family have the same name. If information about family members were stored in a database, a key could be placed over the family member's name. In a University, they have no such luxury. Each student is typically assigned a Student ID, which are used as keys for individual students stored in the school database. Keys can have more than one column, for example, a nation may impose a restriction that a province cannot have two cities by the same name. So, when cities are stored in a relation, there would be a key defined over province and city name. This would allow for two different provinces to have a town called Springfield (because their province would be different), but not two cities with the same name in the same province. A key over more than one attribute is called a compound key. Theoretically, a key can even be over zero attributes. This would enforce that there cannot be more than one tuple in the relation.

Most relations have at least one key defined on it. Because a relation is defined in the relational model as being a set, it cannot have duplicate rows. Some DBMSs do not enforce this. If a DBMS does enforce this, it means that there is always at least one key on each relation, namely the key involving all of the attributes of the relation.

A key could be defined formally by requiring that the cardinality of the relation should be equal to the cardinality of the relation projected over the columns of the key.

A key, in this context, refers to any set of attributes which uniquely span the relation. In particular, this is called a superkey. A candidate key is a minimal superkey, meaning that none of the attributes in the key could be removed from the key and that attribute set would still be a key. Many DBMSs have a concept of a primary key. The primary key (usually a candidate key) is the key most often used to identify a tuple. In some RDBMSs, the primary key of a base relvar is the storage key (sometimes clustered key), meaning that that is how the data is stored physically. If the value of the primary key is actual interesting data with logical ties to the data (like a name) for the tuple, it is called a natural key. If the key is generated and does not have any logical connection to the rest of the data in the tuple, it is called a surrogate key. Other candidate keys that were not chosen as the primary key are called alternate keys.

Foreign keys

A foreign key is not a key by the previous definition. Rather, a foreign key is a reference to a key in another table. Meaning that the referencing tuple has, as part of its attributes, the values of a key in the referenced tuple that corresponds to the relationship.

A foreign key could be described formally as “For all tuples in the referencing relation projected over the referencing attributes, there must exist a tuple in the referenced relation projected over those same attributes such that the values in each of the referencing attributes match the corresponding values in the referenced attributes”.

Transition constraints

A transition constraint is a way of enforcing that the data does not enter an impossible state because of a previous state. For example, it should not be possible for a person to change from being “married” to being “single, never married”. The only valid states after “married” might be “divorced”, “widowed”, or “deceased”.

Other constraints

Other constraints of various different kinds can be created to enforce various kinds of business rules. They can be as simple as “the number of cars an individual owns must be non-negative” or complex patterns like “If the work that an employee performs is ’Hazardous Materials Transport’ then that employee’s age must be at least 18 years, and the employee’s certifications must include 'Hazmat endorsement', and company insurance for that employee must include life insurance."

Chapter 3

Overview of mapping techniques

This chapter is devoted to the overview of XML document structure and storage techniques.

From one point of view XML documents can be divided into two groups by content type: *regular* and *mixed*. Regular documents resemble "de-normalized" relational data. They have regular structure and uses usually scalar values. Mixed documents are more flexible and their values can be inter-leaved with XML markup. As an example can be taken any catalog that has its items stored as XML documents and description of each item is saved as XHTML text.

3.1 Regular Document Example

With the purpose of exchanging and processing healthcare documents, Health Level Seven(HL7)[9] develops document standards for the healthcare industry. The Figure 3.1 describes medical report. A patient has individual information such as family and given names, date of birth and observations.

HL7 documents illustrates the flexibility in regular documents. The patient's driver license is optional. The number of observations is arbitrary. The relationship between parent and child is significant, but the order is insignificant.

Queries that manipulate this documents are usually based on select-project-join and sorting by value. A typical query might return all patients that were born before exact date, sorted by family name and first name or patients that have particular observation. Also grouping operation may occur.

```
<HL7>
    <PATIENT IDNum="PATID1234">
        <PaNa>
            <FaNa>Jones</FaNa>
            <GiNa>William</GiNa>
        </PaNa>
        <DTofBi>
            <date>1961-06-13</date>
        </DTofBi>
        <OBX>
            <ObsVa>150</ObsVa>
            <ObsId>Na</ObsId>
            <AbnFl>Above high</AbnFl>
        </OBX>
        <OBX>
        ...
    </PATIENT>
    ...
</HL7>
```

Figure 3.1: Example of regular XML document

3.2 Mixed Document Example

The Library of Congress put out congressional bills in form of XML documents[12]. The bill contains information such as congress, session, etc. and action description, which may embody annotated text.

The Figure 3.2 shows how annotated text can be involved in document. Also the presence of white spaces and new lines in the annotated text might be meaningful. Queries that operate on mixed documents can be divided into three categories:

Queries on text only. These queries are similar to Information Retrieval (IR) full-text search queries [19, 8]. For example, keyword, stemming and proximity search. A query, which returns all bills that have words “English” and “Coyne” within eight words, is example of proximity search.

Queries on text and structure. These queries combine both search for keywords within text and query the relative order of elements and text. A query that returns text elements containing words “English” and “Coyne” within eight words and their preceding text element is an example of query on text and structure. Numerous proposals [19, 3, 16, 1] have studied how to combine IR technique with document structure.

Queries that span structure. Here belong queries that ignore markups at all. For example a query that returns all the bills that contain text ”referred to the Committee on Financial Services”. This query must

```
<bill bill-stage="Introduction">
    <congress>110th CONGRESS</congress>
    <session>1st session</session>
    <legis-num>H.R. 133</legis-num>
    <current-chamber>IN THE HOUSE OF REPRESENTATIVES
    </current-chamber>
    <action date="June 5, 2008">
        <action-desc>
            <sponsor>Mr. English</sponsor> (for himself
             and <co-sponsor>Mr. Coyne</co-sponsor>)
             introduced the following bill; which was
             referred to the <committee-name>Committee on
             Financial Services</committee-name>...
        </action-desc>
    </action>
</bill>
```

Figure 3.2: Example of mixed XML document

ignore markup <committee-name> to be successful. In [24], the authors present technique to process this kind of queries.

3.3 Properties of XML Data and Queries

XPath and XQuery strongly depends on XML document properties. Thus these properties have strong impact on storage design.

Attributes vs. Elements. Attributes and elements might contain similar values. Scalars or lists. The problem is whether to store attributes and elements the same way or choose some different method. The only difference between attributes and elements is that the order of attributes is insignificant whereas order of elements is not and that elements might contain some more complex structure. XPath and XQuery are designed to query elements and attributes values as well as their relationship.

Heterogeneity. XML document might be heterogeneous due to repetition and alternation. Two documents observing the same XML scheme or DTD can have totally different structure. There are several proposals [25], which study possibilities of involving these heterogeneity into relational systems. XPath 2.0[7] permits querying this heterogeneous documents (`PATIENT/(SURGERY|CHECKUPS)`).

Identity and structure. The element identity depends on element position within the document. Element identity is essential for querying. For example, it is used to define basic operations like element equality and union, intersection and difference of element sequences in XPath. Identity may be significant for some applications, but it is always significant for mixed documents. XPath axes are defined in global document order.

Structure dependence. Parent-child relationship is defined for elements in XML documents. This relationship can include dependency as well. Children are often accessed through parents or other ancestors. This dependence can be used to determine whether "direct" access (e.g. indexing) to elements is necessary or not and decide to either inline dependent elements with parents or not.

Presence of a schema. Although many existing XML documents do not conform to a predefined XML schema, every XML document which is used in data exchange application will have an associated schema that specifies the type of terminal data and constraints on document structure. Schema should be used whenever XML document is used for storage if it exists.

3.4 Storage techniques

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. Next paragraphs describe basic differences between these techniques.

Schema-aware versus schema-obvious. All mapping techniques can be broadly classified into schema-aware and schema-obvious. Schema-obvious techniques store XML documents into predefined generic relational tables. One of the first proposals for mapping XML documents was Edge scheme which stores all the edges in a document tree. (see Section 3.4.1).

On the other hand from generic mappings several specialized strategies have been proposed which make use of schema information to generate efficient mappings. These techniques in general use inlining of elements into their parents whenever it is possible. This reduce the number of joins needed for navigating and/or reconstructing the document.

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, while elements with only one occurrence should be inlined into parents table as a column. The LegoDB system[18] exploits a richer set of mapping primitives. It allows to map also other schema constructs such as choice and repetition. For example, through repetition split transformation it is possible to inline one or more occurrences of the repeated element into its parent table. Note that while most techniques consider primitives that map XML constructs to pure relational systems, some [15, 22] leverage object-relational features of relational systems.

Fixed versus cost-based schema design. Most mapping strategies are fixed, i.e. they fix mapping function. In contrast, LegoDB [18, 13] takes a cost-based approach to derive a mapping that best suits a given application - characterized by a schema, query workload and sample documents. LegoDB creates several mapping possibilities according to the given schema and uses cost for executing query workload on sample documents to work out the most appropriate mapping.

Automated-generation versus Manual-specification. Vast majority of mapping techniques that are proposed int the research literature provide an automated means to derive mappings. Commercial systems [10, 5] allow users to manually specify mappings between XML document and relational schemata. Languages such as XSLT[33], XQuery[27], and IBM's DAD[10] can be used to define mappings that perform arbitrary transformations over an XML document. These types of mappings are very flexible, however they have their drawbacks. It is not easy to define a *good* mapping. Especially if the schema is huge, it is difficult to manually construct mapping that maps all the elements in the document and is efficient at once. This presses the developer to be well acquainted with both, XML and relational technologies. 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.

Preserving order and structure. The main focus of all methods to map XML documents into relational databases is on capturing elements identity, document structure and order. These information can be used to reconstruct the original XML document (except comments, processing instructions, etc.). Only a little research was done on documents with mixed content.

All existing techniques rely on computing unique identifiers for each node in XML document. These identifiers are computed and used differently. In order to reconstruct XML documents multiple joins have to be done. Some techniques use ordered structure of XML document and optimize join algorithms such as sort-merge[23, 4].

Next sections mention some storage techniques to capture identity, document structure and order.

3.4.1 Foreign keys

The simplest way to capture document structure is to use foreign keys to preserve parent/child relationship. Child node holds the unique identifier of its parents. If all nodes have some ordinal value assigned it can be used to keep order. This method is called KFO (Key-Foreign key-Ordinal). Key and foreign key hold structure while ordinal keeps order. Most of the relational systems use this method. It preserves a tree structure of XML document while there is no need to have XML schema of a document. There are four alternatives for edge mapping and two for value mapping. They are discussed in detail in [6]. Each node is assigned some unique integer. Terminal values are either stored in separate table or inlined in relation that stores edges. The edge alternatives are:

- There exists just one relation which holds all edges. Its structure is `[source,ordinal,name,flag,target]`. `source` and `target` are the identifiers of the edge end-points; `name` is the tag on the edge; `ordinal` is an integer that captures the order of siblings and `flag` indicates whether the target of an edge is a node or a value.
- The `Attributes` relations alternative is the result of horizontally partitioning of the previous alternative. Here are created as many relations as there is distinct names in the Edge alternative. It means that all types of relations are captured in different relation. The structure of this relations is the same except the `name`.
- Finally, the `Universal` and the `Normalized Universal` alternatives result from applying full outer-join to the `Attribute` relations.

Of course, many other possible variations exist. Mixed documents could be captured using special value "text" for text nodes and "element" for element nodes in the text.

3.4.2 Interval encoding

Using this algorithm means to create an interval for each node of the document. This interval include intervals of all nodes in its subtree. Interval borders should be unique through the whole XML document. The left border is usually created by preorder traversal and the right one by postorder traversal. In order to distinguish children from descendants the level number is also stored. In [23, 4], the authors develop algorithm with nice linearity properties for searching ancestors, children and descendants of nodes.

3.4.3 Dewey decimal classification

Dewey decimal classification was developed for general knowledge classification[17]. The encoding is based on assigning unique integer to each node and the id of nodes consists of these unique integer of current node concatenated with the id of its parent. It means that each nodes id contain whole path starting with current node and ending with root node.

As id of each node is composed of unique integer (which can be an ordinal keeping node order) and id of parent node, Dewey is the most complete encoding for capturing node identity, document structure and order. However, in Interval encoding only four integer values for each node are needed, in Dewey the deeper in the XML document tree is the node placed, the longer the id of the node is.

3.4.4 Paths

In [31], the authors present XRel mapping method, which uses path storing into following relational schema.

```
Path[pathId,pathexp]
Element[docId,pathId,start,end,index,reindex]
Attribute[docId,pathId,start,end,value]
Text[docId,patchId,start,end,value]
```

There exists a single relation for each type of XML document node (element, attribute, text). Additional relation `Path` is used to avoid path redundancies. Each path is stored as a string and a sub-string matching optimizes all queries. As path can be redundant it does not hold enough information to reconstruct the document. Hence there are stored the start point and end point of the region where the node occurs in the document. Node order is captured by attribute `index` and for query optimalization also `reindex`, which holds reverse node order.

By storing paths, Xrel reduces the number of join operations that need to be performed to recover document structure. It also uses B+-trees and R-trees.

Chapter 4

Theoretical methods

In this chapter there are described some theoretical methods that were developed by research groups. Goal of these groups is to provide some general methods to express XML-to-Relation mapping. 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.

For these reasons several non-commercial approaches have been proposed. They are database system independent, portable, expressive, and easy to use.

4.1 MXM

The authors of MXM (My Xml Mapper) proposed in [2, 29] tried to provide a declarative mechanism to express existing XML-to-Relational mappings and offer an interface to query these mappings. MXM was designed to be independent on whether DTD or XML Schema is used to describe XML documents. In order to achieve these goals, some orthogonal aspects of mapping were identified:

Elements and attributes. Elements and attributes are mapped similarly. Therefore, attribute outlining is allowed. Outlining of attributes captures complex types more naturally. In this case, the relationship between attribute and its containing element is captured in the same manner as document structure except of attributes order.

Groups. The ability to map groups offers additional flexibility. For example, when an XML schema is given, element attribute and group names are used in the mapping. In DTDs, entities are assimilated to groups and nonterminal nodes are used to specify the XML-to-relations mapping.

Document structure. Users are allowed to choose how document structure is captured. This choice is then used uniformly across all XML documents. Each possibility for mapping structure is identified by a unique name, which is used in the mapping. The consequence of this design is extensibility. For example, an external relation can be used to map document structure by adding a reserved word in our system that could be specified in the mapping.

Table names. Every table name can be defined by user, otherwise it is automatically generated.

Defaults. In order to avoid hard-coding the semantics of default mapping rules into each application, default mapping rules can be specified in a "configuration file" that is expressed in MXM. Hence, defaults could be shared by multiple MXM mapping and applications built on top of the XML store can query them. As a part of MXM specification some standard default mapping rules were presented: (1) If not given, table a nd 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.

4.1.1 MXM grammar and examples

The grammar of MXM (see Figure 4.1) is expressed by XML schema. Whole MXM mapping is defined as `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`.

`StructMap` indicates which mapping technique is used to capture document structure between elements and outlined elements and attributes. The attribute `whichMap` can have one of the following values: empty, KFO, INTERVAL, DEWEY, PATH, EXTREL, EDGE, ATTRIBUTE, UNIVERSAL, BASIC, SHARED, HYBRID. All of them refer to the technique described in Section 3.4. EXTREL outlines KFO in an external table.

`TableMap` is used to create tables from source elements, attributes and groups (choice, sequence and allgroup). A table might be assigned a name (otherwise it is automatically generated by concatenating all source names together and `tagField` is used to distinguish between tuples cor-

```
<xsd:schema>
    <element name ="XtoRMapping">
        <attribute name="from" type="string"/>
        <attribute name="to" type="string"/>
        <element name="StructMap">
            <attribute name="whichMap" type="string"/>
        </element>
        <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" minOccurs="1"
                    maxOccurs="unbounded">
                <attibute name="whichCLOB" type="String"
                    use="optional"/>
                <element name="sourceName" type="string"
                    minOccurs="0"/>
            </element>
        </element>
    </element>
</xsd:schema>
```

Figure 4.1: MXM grammar

responding to the same source name). Tag field is optional. In the case input document are described by XML Schema, element, attribute and group names can be used as source names. In the case DTD is used, any non-terminal node name can be used as source name.

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

The Figure 4.2 illustrates MXM mapping example. It captures document structure using KFO and explicitly defines user-given table names.

The name of table, which contains date of births, is auto-generated since it is not defined by the user. The same is true for the table containing observations. In this case, the tag field called `ElemName` is defined and

```
<XtoRMapping from="HL7" to="RelSchema1">
    <StructMap whichMap="KFO"/>
    <TableMap>
        <table whichTable="fullnameTable">
            <sourceName> FaNa </sourceName>
        </table>
        <table whichTable="lastnameTable">
            <sourceName> GiNa </sourceName>
        </table>
        <table>
            <sourceName> DTofBi </sourceName>
        </table>
        <table tagField="ElemName">
            <sourceName> OBX </sourceName>
        </table>
    </TableMap>
    <CLOBMap/>
</XtoRMapping>
```

Figure 4.2: Example of MXM mapping using KFO

```
<XtoRMapping from="HL7" to="RelSchema2">
    <CLOBMap>
        <CLOB whichCLOB="PatientCLOB">
            <sourceName> PATIENT </sourceName>
        </CLOB>
    </CLOBMap>
</XtoRMapping>
```

Figure 4.3: Example of MXM mapping using CLOB

stores the name of element to which the particular tuple corresponds.

The second example (Figure 4.3) shows the creation of a CLOB that contains the whole document. A CLOB should be created only for a fragment of input documents. The element `whichMap` is not specified in this case.

4.1.2 IMXM

The IMXM is a simple interface defined by a set of functions that query mappings and information on the generated relational schema. Example of these functions are:

- `getStructMap()`
- `isInlined(ElemName|AttName)`

- ➢ `getTableName(ElemName|AttName)`
- ➢ `getCLOBName(ElemName|AttName)`
- ➢ `getFields(TableName)`
- ➢ `getFieldType(FieldName)`

Additional functions are used to query defaults.

- ➢ `getDefTableNaming()`
- ➢ `getDefValNaming()`
- ➢ `isDefinline()`

The interface could also by designed at multiple granularities. For example, there can exist a function, which returns all mapping information for specified element such as table name, attribute inlining, sub-elements mapping, etc. The granularity of MXM depends on the application that will make use of it.

MXM and IMXM are implemented on the top of a relational system. IMXM is implemented as a library in C with ODBC calls to database and thus it can use any backend relational system. Using IMXM, relational schema generator as well as a set of loading programs that parse XML documents and populate tables were developed.

The mapping repository conforms to the relational schema given bellow. In order to apply the default mapping rules described in Section 4.1, some information needs to be recorded on the input DTD or XML Schema (if any). This is described in the first set of tables.

`ElemGroup` associates element names (tags) with groups they map to. `AttGroup` associates attributes to groups. `GroupInfo` holds the information about existing groups where `whichGroup` might have one of these values: `empty`, `choice`, `sequence` or `all`. `GroupGroup` captures relationship between parent groups and 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 created. The table `StructInfo` contains information on structure mapping. Groups are associated with tables by `GroupTable`. `GroupField` associates groups with fields. `GroupCLOB` associates group with CLOBs. The last set of tables describes the relational schema that is generated.

4.2 ShreX

ShreX [30] is annotation-based framework. Annotated XML Schema is used to define mapping. Various mapping dimensions are taken into account in design of ShreX annotations. As a result, different mapping choices can be easily combined to create new mapping strategies. The use of annotations makes ShreX very extensible, as new annotations can be added to express new mapping techniques, and very portable, since mapping definition is completely independent on the underlying relational system.

ShreX also provides an API to access mapping informations. It allows define a set of generic functions for the mapping tasks, i.e. functions that are not tied with specifics of a particular mapping strategy.

4.2.1 Architecture overview

Figure 4.4 shows the architecture of ShreX. Users can either manually annotate an input schema, or use the interface provided by the system. *Annotation processor* parses the input annotated XML Schema, checks schema validity according to predefined validity rules and creates the corresponding relational schema. *Mapping repository* is a persistent storage of mapping information created by annotation processor. *Validation parser and Shredder* accepts and validates input XML documents, uses the mapping API to access mapping information and according to them generates the tuples and loads them into *Relational database*. The mapping repository is also accessed by *query translator*, which generates SQL queries from XML queries.

User interface. ShreX provides a graphical user interface, which helps users to define and customize mappings. The interface displays XML Schema and corresponding relational tables that allows users visually check the connections between the XML Schema elements and their relational counterparts as well as interactively modify this connections.

Annotation processor. This module is in charge of parsing the input annotated schema, generating mapping repository and producing `CREATE TABLE` SQL statements for creating corresponding relational schema. The input schema is validated against the XML Schema for annotations. Current version of ShreX supports some simple additional checks. For example, checking whether all annotations are attached to the suitable elements and whether table and attribute names are unique in the mapping definition. More additional checks are possible, for example, verifying whether a mapping is *lossless* - i.e. whether the document can be reconstructed from the mapping tables.

Writing an annotation for every element attribute in XML Schema, especially the large ones, can be tedious. Thus, ShreX provides a set of default mapping rules that is used to *complete* mapping specifications.It

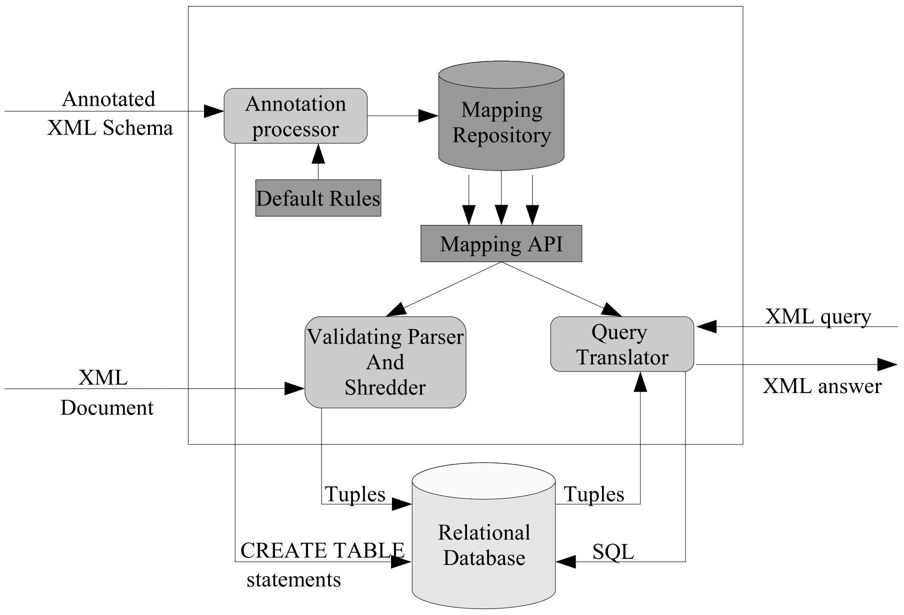

Figure 4.4: ShreX architecture

means that each mapping specification explicitly defined in XML Schema by annotations has higher priority and thus it overrides the default rule. For example, single-occurrence elements, such as `FIRSTNAME`, are always inlined into parent's table by default. Users can define new default rules by adding new annotations into the input XML schema.

Mapping repository and API. The mapping information extracted by annotation processor is stored in a database - the mapping repository. By making this information persistent ShreX avoids re-parsing an input XML schema each time a document is loaded into target database or the XML query needs to by translated. ShreX provides an API for querying the mapping repository. This set of functions is used whenever any 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)`), is needed. Table 4.1 summarizes the functions, which are provided by the API. This API allows users develop mapping-independent code, which works regardless of the specific features of a particular mapping.

Document shredder. This module is in charge of generating tuples, field values and CLOBs from an input XML document. While document is parsed, shredder uses mapping API to query mapping repository for mapping information how a particular element or attribute is mapped and generates the appropriate tuples accordingly.

ShreX uses the SAX interface of Xerces [32], which is both efficient and scalable. In the current implementation are all tuples firstly written to a file and after parsing whole document, they are bulkloaded into the relational backend. The shredder allows users to set various parameters (e.g., target database system, login information, bulk loading options) either through the command line or through configuration file.

Query translator. Query translation (to SQL) currently supports only a subset of XPath. Child and descendant axes, position-based predicates and simple path based expression predicates are implemented. The algorithm consists of the following steps:[1]

```
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)
```

The query translator does not hard-code mapping choices, instead of it it uses the mapping API to dynamically decide how to perform the translation.

Database API. The mappings specifications are portable and independent on the selected relational backend. However, low-level functions must deal with the peculiarities of different database systems. For example, different databases provide different bulkloading options and commands. Since ShreX must be able to invoke these commands, generic database JAVA programming interface was designed, 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.

4.2.2 Mapping definition

Mappings are expressed by annotating an input XML schema. These annotations define how elements or attributes should be mapped into the relational model. Annotations are expressed using attributes from a namespace called *shrex* and can be associated to elements, attributes or groups. Table 4.2 lists all annotations supported by ShreX. The usage of these annotations is illustrated in Figure 4.5. Using the special namespace allows separate the validation of an input document against the XML schema from the validation of the mapping specification.

Mapping identity, structure and order. As mentioned in Section 3.4, every mapping technique must preserve identity, structure and order

[1]For more details on the algorithm see [28].

```
<element name="SHOW" shrex:structurescheme="Dewey" />
    <sequence>
        <element name="TITLE" type="string"
            shrex:outline="true"
            shrex:tablename="ShowTitlt" />
        <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 4.5: Annotated XML schema

to successfully reconstruct an XML document stored in relational system. There exists a special annotation `structurescheme` (see Table 4.2) to define which storage technique should be used. For example, in Figure 4.5, the structure scheme selected for the document is Dewey (see annotation in the root element). ShreX supports Dewey, KFO and interval encoding storage techniques.

Outline, tablename, columnname and sqltype. Annotations are used to specify how individual elements and attributes in a document are represented in a relational schema. The annotation `outline="true"` indicates that the element `TITLE` should be outlined into an external table. This table should be named `Showtitile` according to the annotation `tablename="Showtitle"`. On the other hand the element `YEAR` should be inlined in its parent's table. The annotations `columnname` and `sqltype` determine that the name of the column, to which element `YEAR` should be mapped, is `Showyrear` and its type should be `NUMBER(4)`.

Mapping schemaless documents. ShreX can also handle XML document, which do not have an XML schema assigned. Annotation `edgemapping` can be used for elements or whole documents if their structure is not known in advance. It is especially useful for elements of type `ANYTYPE`. If this annotation is set to true, the element and all its descendants are mapped using general `Edge` mapping technique.

Transformation-based mappings. Combination of schema annotations with the schema transformations proposed in [18] provides additional mapping strategies. For example, if repetition split is applied to `AKA` in the original schema, i.e. `AKA`$*$ $\rightarrow$ `AKA`?, `AKA`$*$, the first occurrence of AKA could be inlined in the table SHOW.

Mapping expressiveness. The mapping strategies *basic inlining*, *shared inlining* and *hybrid inlining* proposed in [11] 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>
```

The XML schema above illustrates an example of mapping which express *basic inlining*. Both, an element TITLE of a MOVIE and an element TITLE of a TV, are mapped to two different tables separately (MovieTitle and TVTitle). To illustrate the *shared mapping* the names of tables have to change. They have to be the same (e.g. Title). This means that they are mapped to the same table and each tuple has an flag whether it corresponds to the Movie or TV. 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.

4.3 Pros and Cons

MXM and ShreX mapping techniques are flexible and platform independent. MXM uses an extra file for mapping information which separate mapping information from the original XML Schema document, but it also force user to use some graphic user interface because it is not obvious which mapping information belong to certain document element, etc.

On the other hand ShreX uses annotated XML Schema documents, which make it easier to pair the mapping information with document node, but the XML Schema document become less readable.

From the technical point of view MXM and ShreX mapping are very similar. They are both using mapping and data repository. The only major difference is the mapping definition.

API Functions	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
isCLOB	attribute or element name	true, false	determines whether the input has been mapped to a CLOB
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
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 4.1: Main functions of ShreX API

Annotation attributes	**Target**	**Value**	**Action**
outline	attribute or element	true, false	If value is true, a rational table is created for the attribute or element. Otherwise, the attribute or element is mapped to one or multiple columns in its containing table.
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.
maptoclob	attribute or element	true, false	If value is true, the element or attribute is mapped to a CLOB column.

Table 4.2: Annotation attributes

Chapter 5

Commercial systems

Major database vendors, Oracle 10i[26], IBM DB2 XML Extender[10] and Microsoft SQL Server[14] offer XML storage and publishing tools on top of their storage system. Due to mismatch of the XML data model and the data model of relational storage, a mapping between this two models is needed. Each vendors offers own proprietary mapping interface to help specify the mapping from XML to relations (and objects) using a declarative mapping schema and special queries. In these systems, document structure is mostly captured by KFO and mixed content is supported in limited way.

This book's main goal is to deal with mapping methods which are independent on relational backend. Thus, these commercial solutions are mentioned to be just and comprehensive. Some interesting features are included in the proposed method described in the second part of this book.

5.1 Oracle 10gR2

As many other database vendors, also Oracle started to explore the usage of relational databases as XML documents storage. In the current version of Oracle database all functionalities which help to maintain XML documents are called Oracle XML DB. It includes tools, packages, object types, etc. which process XML into and out of the database. They include: XML Parser, an XSLT Processor, an XML Schema Processor and XML SQL Utility to generate XML documents, DTDs and schemas from SQL queries. All these tools uses the datatypes for XML (`XMLType`) and for logical pointers (URI-Ref), which were added into database kernel with the previous version.

Oracle XML DB uses annotated XML Schemas as metadata, that is, the standard XML Schema definitions along with several Oracle XML DB-defined attributes. These attributes control how instance XML documents get mapped to the database. Because these attributes are in a different namespace from the XML Schema namespace, such annotated XML Schemas are still legal XML Schema documents.

When using Oracle XML DB with XML Schema, you must first register the XML schema. The XML schema URLs can then be used while creating `XMLType` tables, columns, and views. The XML schema URL, in other words, the URL that identifies the XML schema in the database, is associated with parameter schemaurl of PL/SQL procedure `DBMS_XMLSCHEMA.registerSchema`.

`XMLType` is a datatype that facilitates storing `XMLType` in tables and columns in the database. XML schemas further facilitate storing XML columns and tables in the database and they offer you more storage and access options for XML data along with space- performance-saving options. For example, you can use XML schemas to declare which elements and attributes can be used and what kinds of element nesting, and datatypes are allowed in the XML documents being stored or processed.

Using XML Schema with Oracle XML DB provides a flexible setup for XML storage mapping. For example:

- highly structured data (mostly XML) - each element in the XML documents can be stored as a column in a table
- unstructured data (all or most is not XML data) - the data can be stored in a Character Large Object (CLOB).

Which storage method to choose depends on how the data will be used and depends on the queriability and requirements for querying and updating the data. In other words, using XML Schema gives the user more flexibility for storing highly structured or unstructured data.

As part of registering an XML schema, Oracle XML DB also performs several tasks that facilitate storing, accessing, and manipulating XML instances that conform to the XML schema. These steps include:

- Creating the appropriate SQL object types that enable the structured storage of XML documents that conform to this XML schema. XML-schema annotations can be used to control how these object types are named and generated.
- Creating default `XMLType` tables for all global elements. XML schema annotations can be used to control the names of the tables and to provide column-level and table-level storage clauses and constraints for use during table creation.

All elements and attributes declared in the XML schema are mapped to separate attributes in the corresponding SQL object type. However, some pieces of information in XML instance documents are not represented directly by these element or attributes, such as comments, namespace declarations, prefix information. To ensure the integrity and accuracy of this data, for example, when regenerating XML documents stored in the database, Oracle XML DB uses a data integrity mechanism called DOM fidelity. DOM fidelity refers to how similar the returned and original XML documents are, particularly for purposes of DOM traversals.

In order to provide DOM fidelity, Oracle XML DB has to maintain instance-level metadata. This metadata is tracked at a type level using the system-defined binary attribute `SYS_XDBPD$`. This attribute is referred to as the positional descriptor, or PD for short. The PD attribute is intended for Oracle XML DB internal use only. Users should never directly access or manipulate this column.

The positional descriptor attribute stores all information that cannot be stored in any of the other attributes. PD information is used to ensure the DOM fidelity of all XML documents stored in Oracle XML DB. Examples of such information include: ordering information, comments, processing instructions, and namespace prefixes.

If DOM fidelity is not required, it can be suppressed in the XML schema definition by setting the attribute `maintainDOM=FALSE` at the type level.

Using Oracle XML DB, developers can create `XMLType` tables and columns that are constrained to a global element defined by a registered XML schema. After an `XMLType` column has been constrained to a particular element and a particular XML schema, it can only contain documents that are compliant with the schema definition of that element. An `XMLType` table column is constrained to a particular element and a particular XML schema by adding the appropriate `XMLSCHEMA` and `ELEMENT` clauses to the `CREATE TABLE` operation. Figure 5.1 shows the syntax of creating a `XMLType` table.

```
CREATE [GLOBAL TEMPORARY] TABLE [schema.] table OF XMLType
  [(object_properties)] [XMLType XMLType_storage]
  [XMLSchema_spec] [ON COMMIT {DELETE | PRESERVE} ROWS]
  [OID_clause] [OID_index_clause] [physical_properties]
  [table_properties];
```

Figure 5.1: Syntax of creating a XMLType table

When structured storage is selected, collections (elements which have `maxOccurs` > 1, allowing them to appear multiple times) are mapped into SQL varray values. By default, the entire contents of such a varray is serialized using a single LOB column. This storage model provides for optimal ingestion and retrieval of the entire document, but it has significant limitations when it is necessary to index, update, or retrieve individual members of the collection. A developer may override the way in which a varray is stored, and force the members of the collection to be stored as a set of rows in a nested table. This is done by adding an explicit `VARRAY STORE AS` clause to the `CREATE TABLE` statement. Developers can also add `STORE AS` clauses for any LOB columns that will be generated by the `CREATE TABLE` statement.

The collection and the LOB column must be identified using object-relational notation.

Figure 5.2 shows an example of how to create an `XMLType` table and a table with an `XMLType` column, where the contents of the `XMLType` are constrained to a global element defined by a registered XML schema, and the contents of the `XMLType` are stored using as a set of SQL objects. The example also shows how to specify that the collection of `Action` elements and the collection of `LineItem` elements are stored as rows in nested tables, and how to specify LOB storage clauses for the LOB that will contain the content of the `Notes` element.

```
CREATE TABLE purchaseorder_as_table
  OF XMLType (UNIQUE ("XMLDATA"."Reference"),
              FOREIGN KEY ("XMLDATA"."User")
                  REFERENCES hr.employees (email))
ELEMENT
  "http://xmlns.oracle.com/xdb/documentation/
                       purchaseOrder.xsd#PurchaseOrder"
 VARRAY "XMLDATA"."Actions"."Action"
     STORE AS TABLE action_table1
       ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$))
       ORGANIZATION INDEX OVERFLOW)
 VARRAY "XMLDATA"."LineItems"."LineItem"
     STORE AS TABLE lineitem_table1
       ((PRIMARY KEY (NESTED_TABLE_ID, SYS_NC_ARRAY_INDEX$))
       ORGANIZATION INDEX OVERFLOW)
 LOB ("XMLDATA"."Notes")
     STORE AS (TABLESPACE USERS ENABLE STORAGE IN ROW
       STORAGE(INITIAL 4K NEXT 32K));
```

Figure 5.2: Example of specifying storage options for Schema-based XMLType tables and columns

5.1.1 XML Schema annotations

Oracle XML DB gives application developers the ability to influence the objects and tables that are generated by the XML schema registration process by annotation mechanism.

Annotation involves adding extra attributes to the complexType, element and attribute definitions that are declared by the XML schema. The attributes used by Oracle XML DB belong to the special namespace `http://xmlns.oracle.com/xdb`. In order to simplify the process of annotationing an XML schema, it is recommended that a namespace prefix is declared in the root element of the XML schema.

Most commonly used annotations are listed in Table 5.1.

The registered version of an XML schema will contain a full set of XDB annotations. The location of the registered XML schema depends on whether the schema is local or global. This document can be queried to find out the values of the annotations that were supplied by the user, or added by the schema registration process. For instance, the following query shows the set of global complexType definitions declared by the XMLSchema and the corresponding SQL object types.

```
SELECT extractValue(value(ct),
                    '/xs:complexType/@name',
                    'xmlns:xs="http://www.w3.org/2001/XMLSchema"
                     xmlns:xdb="http://xmlns.oracle.com/xdb"')
       XMLSCHEMA_TYPE_NAME,
       extractValue(value(ct),
                    '/xs:complexType/@xdb:SQLType',
                    'xmlns:xs="http://www.w3.org/2001/XMLSchema"
                     xmlns:xdb="http://xmlns.oracle.com/xdb"')
       SQL_TYPE_NAME,
  FROM RESOURCE_VIEW,
    table(
      XMLSequence(
        extract(
          res,
          '/r:Resource/r:Contents/xs:schema/xs:complexType',
          'xmlns:r="http://xmlns.oracle.com/xdb/XDBResource.xsd"
           xmlns:po=
              "http://xmlns.oracle.com/xdb/documentation/
                                                 purchaseOrder"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb"'))) ct
  WHERE
    equals_path(
      res,
     '/sys/schemas/SCOTT/xmlns.oracle.com/xdb/documentation/
                                             purchaseOrder.xsd')
    =1;
```

Annotation attribute	Value	Action
defaultTable	string	Used to control the name of the default table generated for each global element.
SQLName	string	Used to specify the name of the SQL attribute that corresponds to each element or attribute defined in the XML schema
SQLType	Any SQL type name	For complexType definitions, SQLType is used to specify the name of the SQL object type that corresponds to the complexType definitions. For simpleType definitions, SQLType is used to override the default mapping between XML schema datatypes and SQL datatypes.
SQLCollType	Any SQL collection type name	Used to specify the name of the varray type that will manage a collection of elements.
maintainDOM	true, false	Used to determine whether or not DOM fidelity should be maintained for a given complexType definition.
storeVarrayAsTable	true, false	Used to force all collections to be stored as nested tables. The nested tables are created with system-generated names.

Table 5.1: Commonly used annotations

5.2 DB2

Users can annotate a simplified XML Schema with mapping information. The resulting schema is referred to as the XML Extender Document Access Definition (DAD). DADs are used both for publishing relational data in XML and for storing XML. A DAD mapping defines RDB Nodes. A primary key is needed for each table and column types. Two functions are provided: dxxShredXML() to decompose an incoming XML document and dxxGenXML() to compose a shredded XML. A number of stored procedures are provided for handling XML columns. XMLVarCharFromFile() is used for type conversion. Cast functions Varchar(XMLVarChar) for retrieval. Update functions such as Update(xmlobj, path,value) and selection functions using XPath such as Extractvarchar(). The example below shows a portion of the DAD used to map the HL7 example into DB2.

```
<DAD> <Xcollection> <root_node> <element_node name = "HL7">
    <RDB_node>
        <table_name = "hl7_tab"/>
        <element_node name=''PATIENT''>
            <RDB_node>
                <table name=''Patient_tab''/>
                <condition>IDNum > "635"</condition>
                <attribute_node name=''IDNum''>
                    <RDB_node>
                        <table name=''Patient_tab''/>
                        <column name=''Patient_key''/>
                    </RDB_node>
                </attribute_node>
            </RDB_node>
        </element_node>
    </RDB_node>
</element_node> </root_name> </Xcollection> </DAD>
```

This mapping creates a table Patient tab to store PATIENT elements and a column name Patient key using the attribute IDNum of each PATIENT element. More complex mappings, e.g., using join conditions and vertical partitioning of an element into multiple tables could be provided. In addition, XML columns can be registered with the types: XMLCLOB for large XML documents; XMLVARCHAR for small XML documents and XMLFile for XML documents stored outside DB2.

XML Extender provides an XML DTD repository. Each XML database contains a DTD reference table called DTD REF, which is used to store meta information on users mappings. Users can access this table to insert their own DTDs. These DTDs can be used to validate XML documents.

Given a mapping, the system reads an arbitrary XML document and loads it into a DB2 database. Users do not have to write loading programs by hand.

Mixed content is handled using CLOBs (Character Large OBjects) and side tables for indexing structured data contained in text. Side tables are automatically updated when new documents are inserted. This method of handling mixed content is the most advanced among the solutions provided by database vendors.

5.3 MS SQL Server

Microsoft provides extensions to SQL to publish relational data as XML documents using the FOR XML clause. There are three publishing modes:

RAW, AUTO and EXPLICIT. RAW creates flat XML documents by converting each row in the SQL result into an XML element and each non-NULL column value to an attribute (column name becomes the attribute name). In the AUTO mode, query results are used to build nested documents where each table in the FROM clause is represented as an XML element. The columns listed in the SELECT clause are mapped to attributes or sub-elements. EXPLICIT mode provides more flexible publishing of relational data. It defines a SQL view to assemble relevant rows. Special column names such as Tag and Parent are used. Nesting is explicitly specified as part of the query.

Microsoft adopts three solutions for storing XML documents. It implements the generic Edge technique described in Section 3.4.1 It allows users to annotate an XML schema in order to determine the XML-to-relations mapping. Finally, it provides OpenXML.

Annotated schemas are created using the XML Schema Definition (XSD). The XSD language is the successor to the XML-Data Reduced (XDR) schema definition language. This solution is implemented in SQL-XML that enables XML support for SQL Server 2005 Databases. SQL-XML includes an XDR to XSD converter tool that is designed to help convert annotated XDR schemas to equivalent XSD schemas.

An XSD schema is enclosed in a <xsd:schema> element. Additional attributes that define the namespace in which the schema resides and the namespaces that are used in the schema can be defined for that element. Below is an annotated XSD schema that describes a mapping to a relational database [5]. The view specification contains embedded SQL references. Similarly to IBM, this schema is used both for publishing and for storage.

```
<xsd:schema xmlns:xsd="http://www.w3.org/XMLSchema"
    xmlns:sql="urn:schemas-microsoft:mapping-schema">
    <xsd:element name="PATIENT" sql:relation="Patients">
        <xsd:complexType>
            <xsd:sequence name="PaNa">
                <xsd:element name="FaNa"
                    sql:field="LastName"
                    type="xsd:string" />
                <xsd:element name="GiNa"
                    sql:field="FirstName"
                    type="xsd:string" />
            </xsd:sequence>
            <xsd:attribute name="IDNum"
                sql:field="PatientID"
                type="xsd:integer"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:annotation>
```

```
            <xsd:appinfo>
                <sql:relationship name="Patient_OBX"
                    parent="Patients" parent-key="PatKey"
                    child="OBX" child-key="OBXKey" />
            </xsd:appinfo>
        </xsd:annotation>
    </xsd:schema>
```

The mapping describes the names of the tables and columns used to store XML documents. Mappings can also describe the names of the KFO fields that capture document structure. Mapping schemas are parsed to generate the corresponding relational schema.

The third solution, OpenXML, compiles XML documents into an internal DOM representation using sp_xml_preparedocument. The generic syntax is

```
OPENXML($<$XML doc handler$>$, $<$path expression$>$,
$<$flags$>$) WITH (schema - Table).
```

The T-SQL function is provided to build rowsets from a XML stream. OpenXML can be attribute-centric or element-centric. An example of an attributecentric mapping is given below:

```
Select * from OpenXML(@pat, ‘/HL7/PATIENT’, 1)
WITH (IDNum int GiNa varchar(20))
```

In order to load the XML data in the underlying database, users provide a decomposition of XML documents into multiple tables in a programmatic way which can make this task tedious.

1. The XML document is first parsed into a DOM tree.

2. Users must write XPath expressions to specify XML values to map into tuples and attribute values.

As an example, the user could define a table containing patient records. The user can then specify how tuples in this table are computed using the query: /PATIENT → row in Table Patient specifies that each distinct patient node corresponds to a distinct row in the Patient table. The query /PATIENT//FaNa → LastName in Table Patient specifies how to compute the value of the LastName column in the Patient table.

Microsoft supports storing XML documents in CLOBs. However, unlike IBM, no side tables are provided to index mixed content data.

Templates are used to query the relational database that stores XML data. Templates are XML documents that provide a parameterized query and update mechanism to the database. In a template, elements in the urn:schemas-microsoft-com:xml-sql namespace are processed by the template processor and used to return database data as part of the resulting XML document.

5.4 Pros and cons

Commercial systems and their support for XML documents have one big advantage. They can use database specific features to make XML documents mapping process more powerful. They use their own amended SQL language and therefore use specific database objects and procedures. It is common that XML support is already built-id the database core system. This means that XML documents mapping and subsequent querying is a little bit quicker than other non-commercial approaches.

The price for this performance is platform dependence. This means that each system has its own mapping language.

Chapter 6

Solution design

In this chapter it is described how new proposed mapping method *XRM* (XML to Relation Mapping) reuses some existing features of mapping techniques, which have already been developed (mentioned in the Chapter 4 and 5), and how it integrates new features into those concepts.

The Section 6.1 gives a big picture of the *XRM* system. It is high-level description of how the *XRM* system works. The non-detachable part of this section are the reasons that led to the creation of *XRM* system.

More detailed specification of each module and communication between them can be found in the Section 6.2.

6.1 Analysis

In the previous chapters some of the existing mapping techniques are described. Each of them has number of pros, however also several cons. The *XRM* system tries to inherit some of these advantages and beware of all disadvantages. Three basic conditions which are necessary to comply are:

➢ Flexibility

➢ Platform independence

➢ Easy-to-use

➢ Enriched by some new interesting feature(s)

It is assumed that non-commercial methods (systems) are platform independent. It means that the theoretical method could be implemented in almost any actual programming language and as relational core could be used any current relational database system. Database system independence is the biggest advantage over commercial systems.

On the other hand commercial systems can exploit the specific features of the individual system (database objects, XML support integrated into database core, etc.). However, this individual approaches force users to learn specific mapping language and also specific operating principles.

XRM tries to fulfill platform independence by using standardized SQL (database system independence) and only basic program functions like I/O, DB connections, etc. (program language independence).

Lately, in the chapter 7 it can be found that for the prototype implementation Java is used as an program language and Oracle database system for testing and benchmarking.

The *XRM* system is mostly based on ShreX mapping system. It uses annotated XML Schema documents. This combination provide user an easy way to define mapping rules. Annotated XML Schema can be used to validate XML documents because annotations use different namespace. Document structure definition is stored together with mapping definition in a single file so there is no need of any extra file. 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 an result, *XRM* not only supports all the mapping strategies proposed in the literature, but also new useful strategies that had not been considered previously. For more information about annotations see Section 6.1.1.

6.1.1 Mapping definition

Mappings are expressed by annotating an input XML schema document. Annotations define how specific XML document fragment should be mapped into the relational model. Annotations are added into the XML schema using attributes from a namespace called `xrm`, and can be associated to any attribute, element or group. All available annotations are listed in the Table 6.1.

Annotations can be divided into two groups - local and global. Local annotations as `outlined`, `outlinedMethod`, etc. have impact only on an element they correspond to. For example, if annotation `outlined` is used on element TITLE it means that element TITLE is outlined from its parent element, but it does not necessarily mean that all its descendants have to be outlined. However, global annotations as `outlinedAttributes`, `outlinedElements` and `XPathQueries` have impact on the whole mapping document. This means that their functionality can be expressed by using several corresponding local annotations. Priority of annotation types is:

1. Local annotation
2. Global annotation
3. Default rules

Annotation attributes	**Target**	**Value**	**Description**
outlinedAttributes	root element	true, false	If value is true, all attributes are outlined by default. Only outlined within the attribute definition can override this setting.
outlinedElements	root element	true, false	If value is true, all the elements are outlined by default. Only outlined within the element definition can override this setting.
XPathQueries	root element	string	Comma-separated list of most frequently used XPath queries. If used properly, it speeds up the system.
outlined	attribute, element	true, false	If value is true, a relational table is created for the attribute or element. Otherwise, the attribute or element is inlined.
outlinedMethod	root element	KFO, Dewey, Path, Interval, Edge, etc.	Specifies structure mapping.
datatype	attribute, element of simple type	varchar, number, date, char, CLOB, etc.	The string overrides the SQL type of a column.
name	attribute, element, group	string	The string used as table eventually column name.
queryByValue	attribute, element of simple type	true, false	If value is true, it indicates that many queries by value will be queried on this element or attribute.

Table 6.1: Annotations attributes

The only exceptions are annotations `name` and `queryByValue`. The annotation `name` cannot have neither global nor default value. If there is no local annotation value, this annotation is automatically generated by system (element or attribute name + random number). The annotation `queryByValue` cannot have global value or default rule because of system performance. If this annotation is set to true it means that an index is created for the column in which this element or attribute is stored. Setting this annotation to true for all the elements and attributes leads to the system overload.

Figure 6.1 presents an example of annotated XML Schema document. *XRM* annotations are shown bold. The schema is partly annotated. Initial processing by Annotation processor (described later) completes partly annotated schema into fully annotated schema. It is shown in the Figure 6.2. Fully annotated XML Schema document is used to prepare document repository DB schema and subsequent mapping of XML documents.

6.1.2 IXRM

XRM system stores mapping information into mapping repository. Mapping data are loaded to the repository by using simple SQL INSERT statement. On the other hand there has to be some interface defined to provide ability to query these data. Since *XRM* mapping information 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. In the *XRM* system a simple interface API called *IXRM* (Interface for XML to Relational Mapping) is designed. *IXRM* provides several methods to acquire all the information from the mapping document.

The Table 6.2 lists all methods provided by *IXRM*. All methods get one of input parameter ID of the mapping document (XML Schema document) they want to query. This allows several mapping documents to be stored in a single mapping repository. Based on return values of these methods XML documents are uploaded to data repository and the XPath queries are translated into SQL statements, which returns the result of them.

6.2 Solution overview

The Figure 6.3 depicts the whole architecture of *XRM* system. It consists of three main processes. The first one (with prefix "`1.`") handles input XML Schema document, its completing and storing into mapping repository. The second one (with prefix "`2.`") is in charge of accepting XML documents which are tied to the XML schema already stored in the system. These documents are parsed and divided into tuples according

```
<element name="SHOW" xrm:outlinedMethod="KFO" />
    <sequence>
        <element name="TITLE" type="string"
            xrm:outlined="true"
            xrm:name="ShowTitle" />
        <element name="YEAR" type="integer"
            xrm:outlined="false"
            xrm:name="Showyear"
            xrm:datatype="NUMBER(4)" />
        <element name="REVIEW" type="ANYTYPE"
            minOccurs="0" maxOccurs="unbounded" />
        <element name="AKA" type="string"
            minOccurs="0" maxOccurs="unbounded" />
    </sequence>
</element>
```

Figure 6.1: Example of annotated XML Schema

```
<element name="SHOW" xrm:outlinedAttributes="false"
xrm:outlinedElements="false" xrm:XPathQueries=""
xrm:outlinedMethod="KFO" name="SHOW1"
xrm:queryByValue="false" >
    <sequence>
        <element name="TITLE" type="string"
            xrm:outlined="true" xrm:name="ShowTitle"
            xrm:datatype="varchar"
            xrm:queryByValue="false" />
        <element name="YEAR" type="integer" xrm:outlined=
            "false" xrm:datatype="varchar" xrm:datatype=
            "NUMBER(4)" xrm:name="Showyear" />
        <element name="REVIEW" type="ANYTYPE" minOccurs="0"
            maxOccurs="unbounded" xrm:outlined="true"
            xrm:datatype="CLOB" xrm:name="REVIEW2"
            xrm:queryByValue="false" />
        <element name="AKA" type="string" minOccurs="0"
            maxOccurs="unbounded" =xrm:outlined="false"
            xrm:datatype="varchar" xrm:name="AKA3"
            xrm:queryByValue="false" />
    </sequence>
</element>
```

Figure 6.2: Example of fully annotated XML Schema

Method name	**Return value**	**Description**
isAttributesOutlined	true, false	Returns true if annotation *outlinedAttributes* is set to true, false otherwise.
isElementsOutlined	true, false	Returns true if annotation *outlinedElements* is set to true, false otherwise.
isOutlined	true, false	Returns true if annotation *outlined* is set to true, false otherwise.
getOutlinedMethod	Mapping method name	Return the value of the annotation *outlinedMethod*.
getDatatype	Datatype name	Returns the value of the annotation *datatype*.
getName	String	Returns the value of the annotation *name*.
isSetQueryByValue	true, false	Returns true if annotation *queryByValue* is set to true, false otherwise.

Table 6.2: IXRM methods

to the mapping rules stored in mapping repository. XML documents and its data are stored in data repository. The last process (with prefix "`3.`") parses XPath query into SQL query and runs it on the data repository. It returns the result in XML format.

Each process and its subsystems are described in more details in the subsections below.

6.2.1 XML schema loading process

Before XML documents can be uploaded into the *XRM* system their annotated XML Schema document has to be stored in the system. This is done by XML schema loading process. Steps of this process are marked in the Figure 6.3 by labels with prefix "`1.`".

The annotated XML Schema document is put as an input (`1.1a`) to the Annotation processor (see Section 6.2.4). It parses this document, adds (`1.1b`) missing annotations according to the Default rules file (see Section 6.2.5) and transforms XPath expressions to the form of appropriate annotations. The generated schema document is provided (`1.2`) to the user to make (`1.3`) some changes if any. It is an original XML Schema document with all the mapping information that are needed to successfully map XML documents to the relational system. After user's confirmation (`1.4`), this file is delegated to the DB schema generator (`1.5a`) and Mapping DB input interface (`1.5b`). The DB schema generator (see Section 6.2.7) creates an object model of an appropriate DB

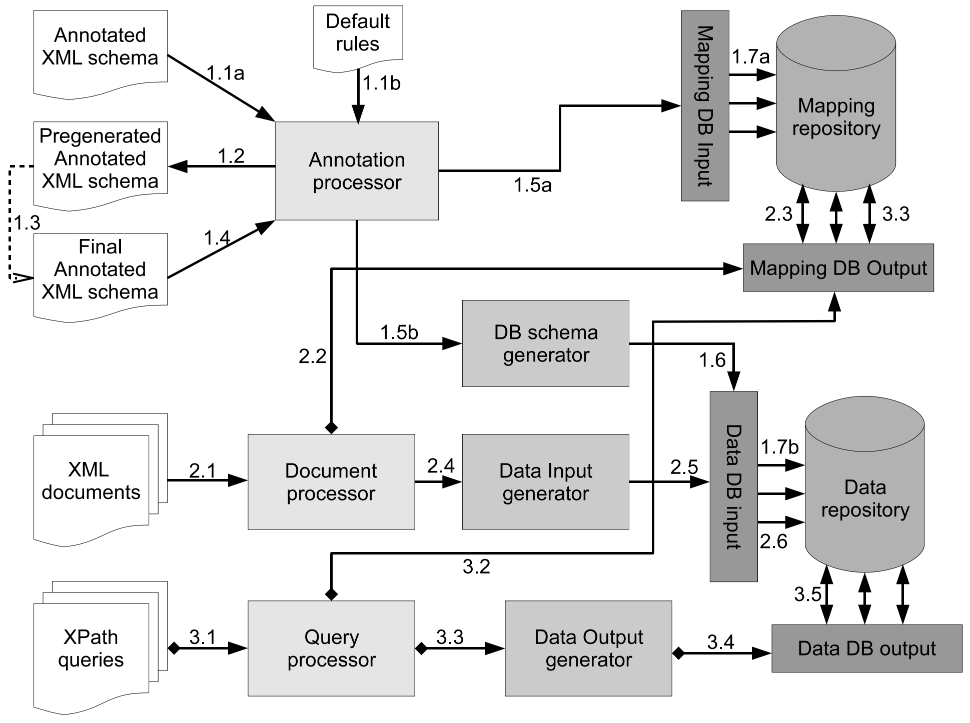

Figure 6.3: XRM architecture

schema for Data repository (see Section 6.2.10) and call (`1.6`) Data DB input (see Section 6.2.10) to create (`1.7b`) this model to prepare Data repository for documents uploading. The Mapping DB input (see Section 6.2.6) stores (`1.7a`) XML Schema document to the Mapping Repository. The Mapping repository is used during XML document loading process to access mapping informations.

After successful execution of this process XML documents tied on this XML schema document can be loaded into the *XRM* system. Mapping information from Mapping repository are used to map these documents to the Data repository. This can be done by XML document loading process (see Section 6.2.2).

6.2.2 XML document loading process

During this process XML documents, whose XML schema is already uploaded in system, can be stored in the *XRM* system.

The document is parsed (`2.1`) in Document processor (see Section 6.2.8). Appropriate mapping rules are loaded (`2.2`) from Mapping repository and internal document (see Appendix C for XML Schema of this file) with all mapping information is passed (2.4) to the Data Input generator (see Section 6.2.9). This module transforms mapping information document into object model of DB schema with all data stored, which is used to physically create (2.5) data in Data repository (see Section 6.2.10) through Data DB input.

6.2.3 XPath query executing process

All XML documents stored in the *XRM* system can be queried by XPath expressions.

The XPath query (3.1) and mapping information (3.2) from Mapping repository are combined and internal query document (see Appendix D for XML Schema of this file) is created in the Query processor (see Section 6.2.11). Data Output generator (see Section 6.2.12) converts (3.3) internal query document into object model of DB schema with conditions loaded. This model is executed (3.4) through Data DB Output, which creates the SQL script for aquiring data. The result is packed in the Data Output generator into query result XML document.

6.2.4 Annotation processor

All documents, which are stored in the *XRM* system, must have assigned an XML Schema document. This schema document and especially annotations used in this schema are used to generate fully annotated XML Schema document. This document holds all information needed to store corresponding XML document.

Annotation processor implements two interfaces - Annotated schema generation and Fully annotated schema processing.

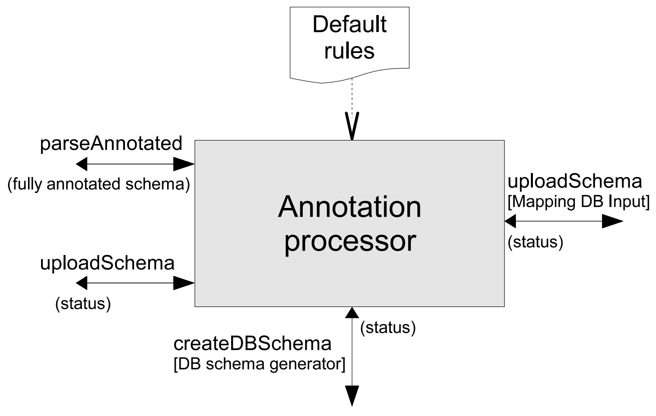

Figure 6.4: Annotation processor

The first one parses incoming annotated XML Schema document and because not every single element has to be annotated by user it adds these missing ones according to the Default mapping rules stored in the *XRM* system. The `XPathQueries` annotation is parsed and each expression listed is transformed into local annotations.

The transformation of XPath expressions affects local annotations (`outlined`, `outlinedMethod`, `queryByValue`). For example, by providing this XPath expression

```
/element1/element2[@attribute2="abc"]/element3
```

on the attribute "attribute2" the annotation `queryByValue` and `outlined` is set to "true", on elements "element2" and "element3" is the annotation `outlined` set to "true" and `outlinedMethod` set to "KFO". Additionally "element3" has `datatype` set to "CLOB". The attribute "attribute1" has `queryByValue` set to true because in the expression query on specific value of this attribute is used. The value of this attribute could be any string and redundancy can not be omitted, therefore the attribute "attribute2" is outlined. "element1" and "element2" are elements with complex type. To avoid huge tables and redundancy their children should be outlined. The most common mapping method for outlined elements is "KFO". While the expression queries on the whole content of "element3" the `datatype` is set to "CLOB". It is the easiest way to return the whole fragment of XML document.

As mentioned before, `XPathQueries` belongs to the group of global annotations, therefore it has lower priority then local annotations. So there do not have to be all the XPath queries which will be ever queried. These queries should only simplify process of defining annotation. Detailed mappings can be defined only by setting local annotations.

As an output of this operation the fully annotated XML Schema document is called back.

The second one accepts fully annotated XML Schema document and passes it to the Mapping DB input module to store this document and DB Schema Generator to create an appropriate schema in Data repository.

6.2.5 Default rules

Annotation processor needs some default rules to generate a fully annotated XML Schema document. In order to avoid hard-coding the semantics of default mapping rules into each application, these rules are specified in an XML document, which is parsed along with other mapping information. The XML Schema document of the default rules XML document can be found in an Appendix B. Almost every type of annotation have an default value. The exception is only the name of relational table in which corresponding element or attribute should be stored. If this annotation is omitted, it means that this name should be automatically generated.

The main benefit of writing default rules as a separate specification and subsequent adding to the XML Schema document is that they could be made queryable and thus applications build on top of the mapping information could be abstracted from any hard-coded choice.

As an extreme case not-annotated XML Schema document can be passed to the Annotation processor and it generates all annotations according to the default mapping.

6.2.6 Mapping repository

Mapping information generated by Annotation processor is stored in Mapping repository. It consists of DB schema in some relational system and two interfaces - Mapping DB input and Mapping DB output.

DB schema in relational system is a physical repository of mapping information. In this repository there are stored fully annotated XML Schema documents, their unique identifiers and DB schema name in which all data in Data repository is stored.

Mapping DB input is an interface to database which provides methods for storing annotated XML Schema documents. Practically it is a simple module which gets an fully annotated XML Schema document and loads it into DB repository.

On the other hand Mapping DB output is an interface to acquire the mapping information. This interface called *IXRM* has been described in the Section 6.1.2. It provides complete access to mapping information. These information is then used to map XML document into Data repository and to evaluate XPath queries.

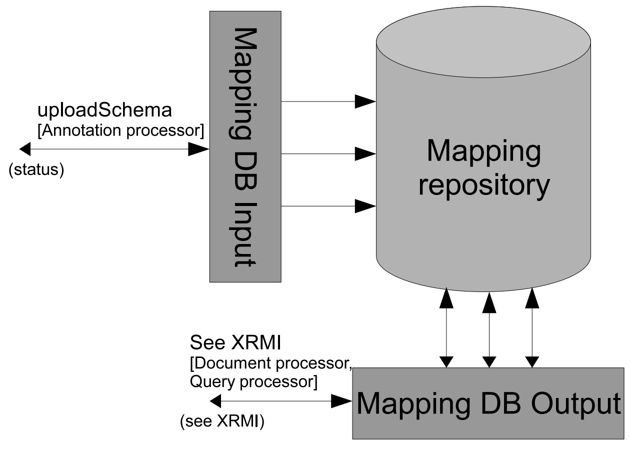

Figure 6.5: Mapping repository

6.2.7 DB schema generator

XML documents are stored in relational database system. All documents, which are tied to the same XML Schema document, are stored in the specific database schema. This schema is created according to the XML schema and its annotations. The DB schema generator converts fully-annotated XML Schema document into an object model of the schema, which is used to create the database schema for this XML schema. It creates relational tables objects according to the outlined annotation and the specifics of storage technique used. For example, if the KFO storage technique is used, there must be created a new table with three or more columns and a foreign key between this new table and the table where the parent is stored. The new tables columns are ID (unique identifier), PID (foreign key to the parent table) and the VALUE column in case it is simple type element or more columns for its attributes and subelements (in case they are not outlined).

This procedure is done for every element and attribute in the XML Schema document. Of course, other storage techniques can be used.

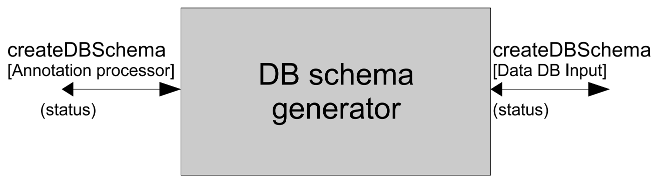

Figure 6.6: DB schema generator

6.2.8 Document processor

Document processor accepts XML documents with known XML Schema document assigned. It uses Mapping DB output and *IXRM* interface to gather mapping information.

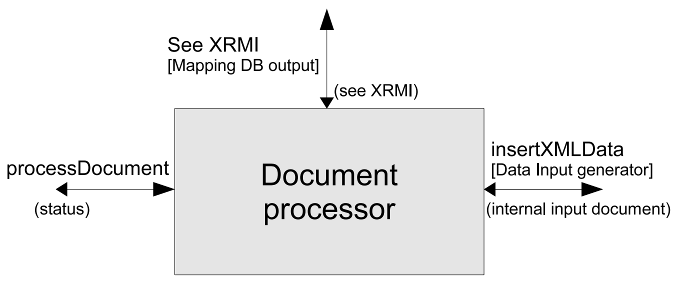

Figure 6.7: Document processor

Using this information the XML document is converted into the internal form which consists of all the XML data and mapping information. It means that each element and each attribute have uniquely determined where it should be stored in the Data repository. This is done by inserting annotations from its XML Schema document into the XML document. This information can be redundant if the element has several subelements of the same type. The mapping information is stored for each of these subelements. However, this redundancy helps DB Input generator to create an object model for document loading into Data repository.

6.2.9 Data Input generator

The internal XML document generated by Document processor is converted into an object model which is used to create the data in the Data repository. Document processor creates an internal form of input XML document which holds all the information needed to insert XML document data into Data repository. It consists of table name or column name, mapping method, flag whether it is outlined or not, flag whether the index should be created, etc.

All this information is processed and object model with all the data is created. The Data DB Input module is then called to insert data from this model into Data repository.

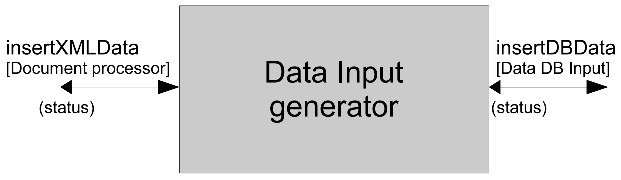

Figure 6.8: Data Input generator

6.2.10 Data repository

As well as repository for mappings data exists, the repository for the data itself exists. It consists of the relational database system and two interfaces - Data DB input and Data DB output. It is similar to the Mapping repository. Data DB input is used to create DB schema according to the XML schema and its annotations and stores XML document data, whereas the Data DB output is used to gather the information to create query output.

The Data DB input accepts two kinds of SQL scripts. One is from DB Schema generator, which is an SQL DDL script. This script creates DB schema according to which it inserts the XML document data into Data repository. It is simple module for storing new XML documents.

On the other hand, the Data DB output accepts XPath query object model. This model consists of all information needed to gather all the data for the XPath query result. It converts this model into a set of SQL SELECT statements and executes them against Data repository.

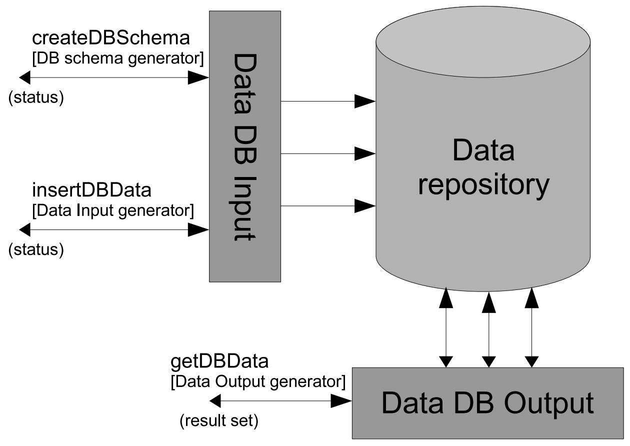

Figure 6.9: Data repository

6.2.11 Query processor

Storing XML documents would have no sense if there existed no means to query these documents. Query processor accepts XPath queries. It uses Mapping DB output interface to acquire necessary information about the location of demanded data. These information and the query are combined into an internal query representation (XML Schema document can be found in Appendix D). This document is similar to the internal document representation. There are missing all the values and some conditions are added. Internal query representation document is passed to the Data Output generator which converts this input document into object model which is sent to the Data DB output.

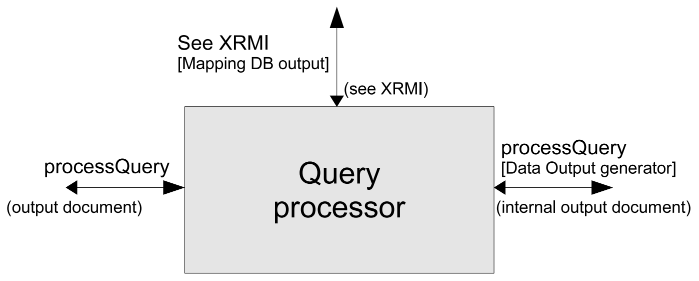

Figure 6.10: Query processor

6.2.12 Data Output generator

The Data Output generator converts the internal query representation document with query and location information into an object model. This document contains all the information needed to find demanding data. This information is transformed similarly as in Data Input generator. The difference is that the Data Output generator creates a model that has several values missing. These missing values are then aquired from the Data repository through the Data DB output interface. The result is transformed into the XML document and it is returned as an query result.

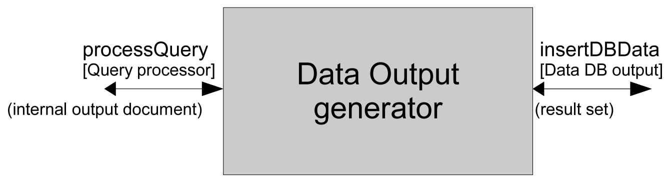

Figure 6.11: Data Output generator

6.3 Mapping completeness

The *XRM* system generates complete mappings. This statement can be supported by the fact that every annotated XML Schema document is automatically completed by global annotations and default rules. The completeness is ensured because every single element and attribute has to have all necessary annotations to correctly map it to the Data repository. Therefore there is uniquely determined where the element or attribute should be stored and how it can be queried back.

Chapter 7

Implementation

As part of this book a prototype implementation of mapping method is designed. It is called like the method itself, *XRM*. In this chapter details of the implementation, used technologies and libraries and control can be found.

7.1 Used technologies

For prototype implementation of mapping method Java language (concretely the application is tested with SDK version 5.0 Update 12) was chosen. The Java language was chosen because of its platform independence of application and because of many libraries for work with XML documents that exist. For XML documents processing SAX parser (concretely its Xerces implementation) is used.

As a relational backend is used Oracle 10g Release 2 database. This database server was chosen because it is one of the most popular databases. *XRM* system uses standardized SQL, therefore it is not significant which database is used.

7.2 Architecture

The architecture of application consists of eight basic and several auxiliary packages. These packages correspond to eight modules of proposed mapping method.

- ➢ **schema.processor**: This package contains classes that implements functionality of Annotation processor
- ➢ **schema.generator**: This package contains classes that implements functionality of DB Schema generator
- ➢ **schema.repository**: This package contains classes that implements functionality of Mapping repository

- **document.processor**: This package contains classes that implements functionality of Document processor
- **query.processor**: This package contains classes that implements functionality of Query processor
- **data.outputgenerator**: This package contains classes that implements functionality of Data Output generator
- **data.inputgenerator**: This package contains classes that implements functionality of Data input generator
- **data.repository**: This package contains classes that implements functionality of Data repository
- **techniques**: This package contains classes which implements functionality of individual storage techniques (such as KFO, PATH)
- **helper**: This package contains helper classes for work with XML documents and database connections
- **execute**: This package contains classes for input and output processing

Each package can contain these subpackages:

- **entity**: This package contains entity Java beans. For example, beans used to create object model of DB schema (`Table`, `Column`, `Row`, `ForeignKey`, etc.)
- **handlers**: This package contains event handlers for XML processng by using SAX parser
- **input**: This package contains classes of an input of the module
- **output**: This package contains classes of an output of the module

Package document.processor

This package implements functionality of Document processor. The main class is `DocumentProcessorImpl` and its interface `DocumentProcessor`. It is dependent on package `schema.repository`, especially *IXRM* part.

Package schema.processor

This package contains classes which implement Annotation processor functionality. The main class is `AnnotationProcessorImpl` and its interface - `AnnotationProcessor` (groups all methods which Annotation processor provides).

The other important class is `DefaultRules` which stores information from Default rules XML document.

Package query.processor

This package implements functionality of Query processor. The main class is `QueryProcessorImpl` and its interface `QueryProcessor`. It is dependent on package `schema.repository`, especially *IXRM* part.

Package schema.generator

This package contains classes which implements functionality of DB Schema generator. The main class is `DBSchemaGeneratorImpl` and its interface - `DBSchemaGenerator`.

DB Schema generator needs classes from package `techniques` since it uses used technique principles to generate DB schema.

Package schema.repository

This package implements functionality of Mapping DB Input and Mapping DB output modules. Since Mapping DB output is basically implementation of *IXRM*, this package contains classes that implements *IXRM* interface.

This is the only package that should work with Mapping repository database. Any other mapping data manipulation could cause system failure.

Package data.outputgenerator

This package implements functionality of Data Output generator. Its main classes are `DataOutputGeneratorImpl` and its interface `DataOutputGenerator`.

Package data.inputgenerator

This package implements functionality of Data input generator. Its main classes are `DataInputGeneratorImpl` and its interface `DataInputGenerator`.

Package data.repository

This package implements functionality of Data DB Input and Data DB output modules.

This is the only package that should work with Data repository database. Any other data manipulation could cause system failure.

7.3 Control

The prototype implementation can be run using command prompt. There exist three subapplications which starts three different *XRM* processes. XML schema input process accepts one input parameter - annotated XML Schema in first step and fully-annotated schema in second step. XML document loading process accepts one input parameter - XML document which has assigned an XML Schema that is already loaded in the system. XPath query executing process accepts XPath expression.

Results of each subapplication can be stored in file or displayed on the screen depending on the input parameter. There exists an application log stored in `/log` directory. In this log are written all information about program execution and an error states and their causes of course.

7.4 Implementation limitations

The enclosed implementation is only prototype implementation and there should be made several modifications and completions to be able to deploy this application in production environment. There are three places where it can be enriched.

There are currently implemented just two storage techniques that can be used - KFO and PATH. The *XRM* is, however, ready to add some new ones.

XPath expressions that can be used in *XRM* are restricted to use only child and attribute axes. The predicate part can contain element value, attribute value and position. The Query processor can be upgraded to allow more axes.

`XPathQueries` annotation translation into local annotations can be improved to use some more complex algorithm. Current implementation uses just few rules how these queries modify local annotations.

The prototype implementation is designed to be illustrating the *XRM* system architecture at the expense of performance and memory complexity.

Chapter 8

Experimental results

In this chapter there are described several experimental executions of prototype implementation. They should present basic implementation features and principles. As prototype is not implemented to fulfill all the performance requirements, performance issues are neither tested nor compared with other mapping techniques implementations. There are described behavior of *XRM* system, how the data DB schema changes according to the annotations alternations, etc.

Every test presented here consists of four steps:

➢ Annotate XML Schema document and upload it to the Mapping repository

➢ Insert a sample document into Data repository

➢ Query this document with an XPath expression

After each step there are described all the changes that were taken place in the system, especially in the Mapping and Data repositories. The XML Schema document used in these examples is shown in the Figure 8.1. The XML document used as an data example is stored on the enclosed CD-ROM. There were queried two XPath expressions:

```
/shiporder[@orderid="123"]
/shiporder/item[title="PC"]
```

In the first case the XML Schema document was less annotated. Only three elements were annotated - `shiporder`, `shipto` and `item`. Fully-annotated XML Schema document is stored on the enclosed CD-ROM. Most of the annotations were set according to the Default rules (stored on enclosed CD-ROM). There were created three tables. One for each outlined element.

By querying the first XPath expression *XRM* has to join three tables to acquire the result. On the other hand the second query does not have

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="shiporder">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="orderperson"
                  type="xs:string"/>
            <xs:element name="shipto">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="name"
                        type="xs:string"/>
                     <xs:element name="address"
                        type="xs:string"/>
                     <xs:element name="city"
                        type="xs:string"/>
                     <xs:element name="country"
                        type="xs:string"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element name="item" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="title"
                        type="xs:string"/>
                     <xs:element name="note" type="xs:string"
                        minOccurs="0"/>
                     <xs:element name="quantity"
                        type="xs:positiveInteger"/>
                     <xs:element name="price"
                        type="xs:decimal"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
         <xs:attribute name="orderid" type="xs:string"
            use="required"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Figure 8.1: XML Schema used in experiments

to join any tables. Its result can be constructed from one table - table for the element `item`.

In the second case the XML Schema document was already fully-annotated. Every element and attribute of this schema was annotated by `outlined` annotation set to true. This means that there were created one table for every element and attribute - 13 tables.

By querying the first XPath expression *XRM* has to join all 13 tables. This is not optimal solution since table joins are expensive database operations. The second query does not make any difference. There are 5 table joins needed.

In the third case the XML schema document was annotated like in the first case, the only difference was added `XPathQueries` annotation added to the `xs:schema` element. This alternation changed the DB schema. There were created 8 tables. Every element has its own table only the whole element `shipto` is stored in one CLOB value. This reduces the number of table joins while keeping the data granularity.

This examples shows that *XRM* system fulfill all basic functionality of XML-to-Relations mapping system and moreover provides features to simplify annotating process.

Chapter 9

Conclusion

The goal of this thesis was to explore the opportunities and limitations of mapping techniques which map XML documents into relational system and using this analysis propose and implement new solution.

The first part of this thesis analyzes already proposed mapping techniques, such as MXM, ShreX and commercial ones (by Oracle, IBM and Microsoft). The result of this analysis was formulation of basic criteria for the proposed solution.

On the basis of this analysis, especially on the basis of realized features and pros and cons of the existing approaches, there was proposed a new mapping solution. The annotated XML Schema was selected because of XML Schema flexibility, annotated schema transparency and also because XML Schema is standardized by W3 Consortium. New proposed annotations helps an unfamiliar user to use this mapping technique without learning every annotation type meaning and subsequent meaning of their effects. On the other hand they also gives more possibilities for expert user to optimize system performance.

The benefit of this thesis consists in selecting XML Schema for defining document structure and adding some new annotations which simplifies process of annotating the XML Schema documents. The annotation `XPathQueries` helps an unfamiliar user to annotate XML Schema more optimally without hard-studying XML and relational system details. The annotation `queryByValue` helps an expert user to more precisely define where should be created database indexes and where not. The next benefit is the modularity of solution system, therefore it is simple to extend the system with new storage techniques, default rules definition or `XPathQueries` annotation transformation.

Though there were completed essentially all thesis goals, in the proposed solution there are places which can be further improved and extended. One of the places can be defining more detailed rules for translating `XPathQueries` annotation into local annotations. Especially support for more XPath axes. The next area of improvement is prototype implementation. There are some places where it can be upgraded to support more storage techniques and more XPath axes for executing XPath expressions. The last opportunity for improvement is the graphical user interface. Controlling the program through command prompt is not comfortable. An GUI with visualizing the annotations could be very useful.

Bibliography

[1] G. Weikum A. Theobald. *Adding Relevance to XML.* WebDB'00, 2000.

[2] Sihem Amer-Yahia. *Storage Techniques and Mapping Schemas for XML.* AT&T Labs Research, 2003.

[3] E. W. Brown. *Fast Evaluation of Structured Queries for Information Retrieval.* SIGIR, 1995.

[4] D. J. DeWitt C. Zhang, J. F. Naughton and Q. Luo. *On Supporting Containment Queries in Relational Database Management Systems.* SIGMOD, 2001.

[5] Microsoft Corporation. Microsoft support for xml. http://msdn.microsoft.com/sqlxml.

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

[7] Anders Berglund et al. Xml path language (xpath) version 2.0. http://www.w3c.org/TR/xpath20, 2002.

[8] M. J. McGrill G. Salton. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[9] Health level seven. http://www.hl7.org, 2003.

[10] IBM. Ibm xml extender. http://www-4.ibm.com/software/data/db2/extenders/xmlext/docs/v71wrk/english/index.htm.

[11] G. He J. Shanmugasundaram, K. Tufte and C. Zhang. Relational databases for querying xml documents: Limitations and opportunities, 1999.

[12] The library of congress. http://www.loc.gov, 2003.

[13] J. Haritsa M. Ramanath, J. Freire and P. Roy. *Searching for afficient XML-to-Relational mappings.* Proceedings of the International XML Database Symposium, 2003.

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

[15] M.Klettke and H. Meyer. *XML and object-relational database systems - enhancing structural mappings based on statistics.* Proceedings of the Workshop on Web and Databases, 2000.

[16] K. Grossjohann N. Fuhr. *XIRQL - An Extension of XQL for Information Retrieval.* SIGIR, 2001.

[17] Introduction to the dewey decimal classification. http://www.oclc.org/dewey/about/about_the_ddc.htm.

[18] P.Roy P. Bohannon, J. Freire and J. Simeon. From xml schema to relations: A cost-based approach to xml storage, 2002.

[19] G. Navarro R. Baeza-Yates. *Integrating Contents and Structure in Text Retrieval.* ACM SIGMOD Record, 1996.

[20] W3C Recommendation. Extensible markup language (xml). http://www.w3.org/TR/REC-xml, 2004.

[21] W3C Recommendation. Xml schema. http://www.w3.org/TR/xmlschema-1, 2004.

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

[23] N. Koudas S. Al-Khalifa, H. V. Jagadish and J.M. Patel. *Structural joins: A Primitive for Efficient XML Query Pattern Matching.* ICDE, 2002.

[24] D. Srivastava S. Amer-Yahia, M. Fernandez and Y. Xu. *Exact and Approximate Phase Matching in XML.* CIKM, 2002.

[25] R. Greer S. Amer-Yahia, M. Fernandez and D. Srivastava. *Logical and Physical Support for Heterogenous Data.* CIKM, 2002.

[26] M. Krishnaprasad S. Banerjee, V. Krishnamurthy and R. Murthy. *Oracle 8i - The XML Enabled Data Management System.* ICDE, 2000.

[27] M. Fernandez et al S. Boag, D. Chamberlin. *XQuery 1.0: An XML query language.* W3 Consortium, 2004.

[28] C. Hara S. Davidson, W. Fan and J. Qin. Propagating xml constraints to relations, 2003.

[29] Divesh Srivastava Sihem Amer-Yahia. *A Mapping Schema and Interface for XML Stores.* AT&T Labs Research, 2002.

[30] Juliana Freire Sihem Amer-Yahia, Fang Du. *A Comprehensive Solution to the XML-to-Relational Mapping Problem.* AT&Labs Research, OGI/OHSU, 2004.

[31] S. Uemura T. Shimura, M. Yoshikawa. *A Path-Based Approach to Storage and Retrieval of Documents using Relational Databases.* TOIT, 2001.

[32] Xerces java parser 1.4.3. http://xml.apache.org/xerces-j.

[33] Xsl transformation (xslt). http://www.w3.org/TR/xslt.

Appendix A

XRM annotations - XML Schema document

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://ksi.mff.cuni.cz/xrm">

    <xs:attribute name="outlinedAttributes"
        type="xs:boolean" default="false"/>

    <xs:attribute name="outlinedElements"
        type="xs:boolean" default="false"/>

    <xs:attribute name="outlined"
        type="xs:boolean" default="false"/>

    <xs:attribute name="outlinedMethod"
        type="xs:string" default="KFO"/>

    <xs:attribute name="datatype"
        type="xs:string" default="varchar"/>

    <xs:attribute name="name"
        type="xs:string"/>

    <xs:attribute name="queryByValue"
        type="xs:boolean" default="false"/>

    <xs:attribute name="XPathQueries"
        type="xs:string"/>

</xs:schema>
```

Appendix B

Default rules - XML Schema document

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://ksi.mff.cuni.cz/xrm">
   <xs:element name="DefaultRules">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="outlinedAttributes"
               type="xs:boolean" default="false"/>
            <xs:element name="outlinedElements"
               type="xs:boolean" default="false"/>
            <xs:element name="outlinedMethod"
               type="xs:string" default="KFO"/>
            <xs:element name="datatype" type="xs:string"
               default="varchar"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Appendix C

Internal document representation - XML Schema document

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:attributeGroup name="XRMAttributeGroup">
      <xs:attribute name="tableName" type="xs:string"
         use="required"/>
      <xs:attribute name="columnName" type="xs:string"
         use="optional"/>
      <xs:attribute name="datatype" type="xs:string"
         use="optional"/>
      <xs:attribute name="outline" type="xs:string"
         use="optional"/>
      <xs:attribute name="queryByValue" type="xs:boolean"
         use="required"/>
   </xs:attributeGroup>

   <xs:complexType name="attributeType">
      <xs:attribute name="value" type="xs:anySimpleType"/>
      <xs:attribute name="attributeName" type="xs:string"/>
      <xs:attributeGroup ref="XRMAttributeGroup"/>
   </xs:complexType>

   <xs:complexType name="elementType">
      <xs:sequence>
         <xs:element name="element" type="elementType"
            minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name="attribute" type="attributeType"
            minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="value" type="xs:anySimpleType"/>
      <xs:attributeGroup ref="XRMAttributeGroup"/>
      <xs:attribute name="elementName" type="xs:string"/>
   </xs:complexType>

   <xs:element name="root">
      <xs:complexType>
         <xs:choice>
            <xs:element name="value" type="xs:anySimpleType"/>
            <xs:sequence>
               <xs:element name="element" type="elementType"
                  minOccurs="0" maxOccurs="unbounded"/>
               <xs:element name="attribute" type="attributeType"
                  minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
         </xs:choice>
         <xs:attribute name="elementName" type="xs:string"/>
         <xs:attribute name="outlinedMethod" type="xs:string"/>
         <xs:attribute name="XMLSchemaNamespace"
            type="xs:string"/>
         <xs:attributeGroup ref="XRMAttributeGroup"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Appendix D

Internal XPath query representation - XML Schema document

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:attributeGroup name="XRMAttributeGroup">
      <xs:attribute name="tableName" type="xs:string"
         use="required"/>
      <xs:attribute name="columnName" type="xs:string"
         use="optional"/>
      <xs:attribute name="datatype" type="xs:string"
         use="optional"/>
      <xs:attribute name="outlineMethod" type="xs:string"
         use="optional"/>
      <xs:attribute name="queryByValue" type="xs:boolean"
         use="required"/>
   </xs:attributeGroup>

   <xs:complexType name="attributeType">
      <xs:sequence>
         <xs:element name="value" type="xs:anySimpleType"/>
      </xs:sequence>
      <xs:attribute name="attributeName" type="xs:string"/>
      <xs:attributeGroup ref="XRMAttributeGroup"/>
   </xs:complexType>

   <xs:complexType name="elementWhereType">
      <xs:sequence>
         <xs:element name="value" type="xs:anySimpleType"
            minOccurs="0"/>
         <xs:element name="attribute" type="attributeType"
```

```
            minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name="position" type="xs:integer"
            minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="elementType">
      <xs:sequence>
         <xs:element name="where" type="elementWhereType"/>
         <xs:choice>
            <xs:element name="value" type="xs:anySimpleType"/>
            <xs:sequence>
               <xs:element name="element" type="elementType"
                  minOccurs="0" maxOccurs="unbounded"/>
               <xs:element name="attribute"
                  type="attributeType" minOccurs="0"
                  maxOccurs="unbounded"/>
            </xs:sequence>
         </xs:choice>
      </xs:sequence>
      <xs:attributeGroup ref="XRMAttributeGroup"/>
      <xs:attribute name="elementName" type="xs:string"/>
   </xs:complexType>

   <xs:element name="root">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="where" type="elementWhereType"/>
            <xs:choice>
               <xs:element name="value"
                  type="xs:anySimpleType"/>
               <xs:sequence>
                  <xs:element name="element" type="elementType"
                     minOccurs="0" maxOccurs="unbounded"/>
                  <xs:element name="attribute"
                     type="attributeType" minOccurs="0"
                     maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:choice>
          </xs:sequence>
          <xs:attributeGroup ref="XRMAttributeGroup"/>
          <xs:attribute name="elementName" type="xs:string"/>
          <xs:attribute name="XMLSchemaNamespace"
             type="xs:string"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Appendix E

CD-ROM content

A part of this book is an enclosed CD-ROM which contains the text of this book and the first of all the source code including the documentation and compiled files of the prototype implementation of *XRM*. CD-ROM contains following files and directories:

- ➢ content.txt - file with the description of CD-ROM content
- ➢ /text - directory with the text of this book
- ➢ /src - directory with the source code of prototype implementation
- ➢ /doc - directory with the documentation of prototype implementation (generated by javadoc application)
- ➢ /release - directory with compiled application *XRM*
- ➢ /data - directory with the data used to verify prototype implementation in the Chapter 8