# XML in the World of (Object-) Relational Database Systems

Irena Mlýnková, Jaroslav Pokorný

Charles University
Faculty of Mathematics and Physics
Department of Software Engineering
Malostranské náměstí 25
118 00 Prague 1, Czech Republic

{mlynkova,pokorny}@ksi.ms.mff.cuni.cz

**Abstract.** *Recently XML is quickly becoming a crucial format for representing and exchanging information. However this tendency brings an essential demand for effective storage and management of XML documents. As XML technology has many database features (e.g. storage strategy, query languages, schemes, programming interfaces, etc.), one possible solution can be found in storing XML data in relational or object-relational database systems.*

*At present there is a relatively large number of works and thesis concerning techniques, which somehow combine XML and (object-) relational features. This survey tries to sum these techniques up, to classify them, and to evaluate their advantages and disadvantages.*

## 1 Introduction

Exchanging and especially effective way of processing information has always had a strategic significance for the whole mankind. To enable two objects to communicate with each other it is necessary for both to know the structure of exchanged data and to be able to react flexibly on its changes. Particularly the latter requirement is usually difficult to meet.

In response to this problem XML [9] is quickly becoming a key format for representing and exchanging information. The reason for this tendency is the fact, that XML is not only a language for describing the data itself, but especially for describing its structure. And just this feature simplifies processing XML data considerably.

On the other hand the growing usage of XML technologies brings an essential demand for effective storage and management of XML documents as well as for querying XML data. One possible solution can be found in storing XML data in (object-) relational database systems. This idea results from the way of describing the structure of XML documents, which resembles (object-) relational features in many ways and from other database features (e.g. query languages, schemes, programming interfaces, etc.) XML technologies include. In addition, connecting these two technologies brings another advantage – it enables to provide XML with missing database mechanisms (e.g. indexes, transactions, multi-user access, querying multiple documents, etc.).

Currently there is a relatively large number of works describing techniques, which somehow combine XML and (object-) relational technologies. These techniques have several common characteristics and similar principles according to which they can be classified. This survey tries to sum these techniques up, to discuss possible classifications, and to evaluate their advantages and disadvantages.

The survey is structured as follows: Section 2 briefly describes the basic classification of XML documents. According to this classification Section 3 sums up and evaluates techniques for connecting XML and database technologies. Section 4 contains an overview and possible classifications of methods for transferring data between XML documents and (object-) relational database systems. Section 5 contains an evaluation and discussion of these methods and Section 6 provides conclusions.

## 2 Types of XML Documents

The most general and most usual classification of methods concerning XML data is related to dividing XML documents into two groups according to their content, structure, and supposed use. These two groups are called either *data-centric* and *document-centric* documents [7] or *regular* and *mixed* documents [1]. In this survey will be used the former possibility.

Although the distinction between these two groups is not generally obvious (documents which belong to both groups are sometimes called *hybrid* documents), it works for most XML documents and can be a good start point for classification.

### 2.1 Data-Centric Documents

Data-centric XML documents are sometimes called “documents designed by machines for machines”, what means they are not supposed to be read by people.

```
<Book idNum="12345">
  <Name>All I Really Need to Know I Learned in Kindergarten</Name>
  <Author>
    <FirstName>Robert</FirstName>
    <Surname>Fulghum</Surname>
  </Author>
  <Publisher name="Ivy Books">
    <Year>1989</Year>
    <City>New York</City>
  </Publisher>
  <Publisher name="Ballantine Books">
    <Year>2003</Year>
    <City>New York</City>
  </Publisher>
</Book>
```

Figure 1: An example of a data-centric XML document

As can be seen in Figure 1, these documents have a fairly regular structure, usually do not contain elements with mixed content, comments, processing instructions or CDATA sections, and the order of sibling elements is generally unimportant.

## 2.2 Document-Centric Documents

On the other hand document-centric XML documents are sometimes described as “documents designed by people for people”.

```
<Book idNum="12345">
  <Name>All I Really Need to Know I Learned in Kindergarten</Name>
  <Author>Robert Fulghum</Surname>
  <Sub>Fifteenth Anniversary Edition Reconsidered, Revised and
  Expanded With Twenty-Five New Essays</Sub>
  <Para>Fifteen yeas after publishing <q>his</q> <i>Kindergarten</i>
  Robert Fulghum decided to read the book again, now in <b>2003</b>.
  He wanted to know whether his ideas have changed and if so how and
  why...</Para>
</Book>
```

Figure 2: An example of a document-centric XML document

These documents (see Figure 2) usually contain elements with mixed content as well as comments, CDATA sections, etc. The order of sibling elements is mostly significant and the structure of these documents is generally irregular (sometimes we speak about so-called *semistructured* data).

# 3 Connecting XML and Database Systems

As mentioned above, connecting XML and database systems can bring many advantages, especially in enriching XML technologies with database mechanisms they lack. In recent years many different techniques combining these two technologies with specific purpose and features have appeared. Generally they can be divided according to types of XML documents (as were described in previous section) for which they were primarily designed. These techniques are briefly summed up and evaluated in this section (more detailed description and a list of existing products can be found e.g. in [7] or [8]).

## 3.1 Techniques for Document-Centric XML Documents

As was described in previous section, it is very important for document-centric documents to preserve the order of sibling elements, as well as comments, CDATA sections, etc. – at the very most to preserve the document as a whole, including things such as white spaces. The process of storing an XML document in a database system and retrieving the (same) document back is called *round tripping* [7].

### 3.1.1 LOB

The simplest technique of connecting XML and database technologies is storing the entire XML documents in one table column of CLOB or BLOB data type.

This method ensures the best level of round tripping, enables to manipulate XML document as a whole and provides basic database mechanisms such as transactions, multi-user access, etc. The biggest disadvantage of this method is obvious – the loss of possibility to query the stored XML data. This problem can be partly solved using different kinds of

XML-aware full-text searching, which do not treat XML documents as pure text, but consider the tagging. But neither these techniques do solve the basic disadvantage well.

### 3.1.2 Native XML Database (NXD)

Native XML database is a special kind of database designed especially for storing, querying, and manipulating XML documents. As well as “ordinary” database NXD provides mechanisms like transactions, querying, programming interfaces, etc., but its internal logical model is based on XML. The model is usually general and enables to store any kind of XML document regardless its structure, while the storage strategy provides a good round tripping level. Physical storage model (whether relational database or any other format) is not determined exactly.

Another advantage of NXD expresses the word “native”, which means that NXD supports “natural” ways for accessing the stored XML data – e.g. XML query languages, addressing parts of XML documents, DOM [10] or SAX [18] interfaces, etc.

As NXD uses a certain strategy for storing and ordering XML data, the biggest disadvantage (i.e. performance problems) can encounter in case of retrieving the data in any form other than that in which it is logically stored, such as when inverting the hierarchy or its portions, etc.

### 3.1.3 Persistent Document Object Model (PDOM)

PDOM technology is a special kind of NXD based on the idea of persistent DOM trees. As well as most kinds of native XML databases a database based on PDOM technology is able to return DOM trees of stored XML documents. The persistence in this case means, that changes of returned DOM tree reflect directly in the source database. In other words PDOM technology provides for DOM applications persistent data storage and at the same time an analogy of virtual memory. The latter feature is useful especially while working with large XML documents.

### 3.1.4 Content Management System

Content management system is also a special kind of NXD designed especially for storing and managing human-written XML documents. It provides mechanisms such as multi-user access, version control system, XML editor, etc. and enables users to split XML documents into logical fragments and to work just with them, rather than with whole (possibly large) documents.

## 3.2 Techniques for Data-Centric XML Documents

Techniques for data-centric XML documents have one common idea: XML data is stored and processed in a relational or object-relational database system and using a certain method they are transferred between relations and XML documents and vice versa.

As mentioned above, for data-centric XML documents features like the order of sibling elements, comments, annotations, etc. are usually unimportant. The level of round tripping is hence quite low – the returned XML document can considerably differ from the stored one. They are “same” only at the level of elements, attributes, hierarchical structure and the data itself.

### 3.2.1 Middleware

Middleware is third party software used by data-centric applications for transferring data between XML documents and a certain database system.

### 3.2.2 XML-Enabled Database

XML-enabled database is a DBMS, usually relational or object-relational, which contains additional functions and extensions for transferring the data between XML documents ant its internal structures. The main difference between NXDs and XML-enabled databases is that NXDs usually use generic structures, which can hold any XML document, whereas the storage strategy of XML-enabled databases is usually driven by previously known schema of stored XML documents.

### 3.2.3 XML Server

XML servers are XML-aware servers (e.g. J2EE servers, web application servers, custom servers, etc.) designed and/or suited particularly for processing XML data. They can be used for creating distributed XML applications (such as e-commerce or B2B applications), publishing XML documents on the Internet, etc. XML servers usually work with both data and document-centric XML documents.

### 3.2.4 XML Query Engine

XML query engine is a standalone application created especially for querying XML documents. XML query engines are usually able to process both data and document-centric XML documents.

### 3.2.5 XML Data Binding

XML data binding technology is based on the idea of mapping XML data to classes and objects of an object-oriented programming language (usually Java or C++). The structure of these classes is designed particularly for each XML document (or a collection of XML documents). This technique tries to enable applications to work with XML data using structures, which can be more suitable and natural for them than, e.g., structures of a DOM tree.

# 4 Methods of Storing XML Data

Section 3 contained summaries of techniques for connecting XML and database technologies. Most of the mentioned ones can use or can be based on relational or object-relational database systems and thus require a suitable method for transferring the data between XML documents and (object-) relational structures (so-called *mapping method*). The best-known representatives of these methods (mostly for data-centric approach) are described and classified in this section.

Existing mapping methods have many common features and characteristics according to which they can be classified differently. A basic classification [1] includes following three classes:

- *generic methods* – mapping methods which do not use any schema of stored XML documents,
- *schema-driven methods* – mapping methods based on existing schema of stored XML documents, and
- *user-defined methods* – methods based on user-defined mapping.

## 4.1 Generic Mapping Methods

Generic mapping methods do not use (possibly) existing XML schema[1] of stored XML documents. They are usually based on one of these approaches:

- to create a general (object-) relational schema into whose relations can be stored any XML document regardless its structure or
- to create a special kind of (object-) relational schema into whose relations can be stored only a certain collection of XML documents having a similar structure.

This section describes several representatives of each approach. In the former case it is a group of four methods which view XML document as tree – for the purposes of this survey they are called generic-tree mapping, structure-centred mapping, simple-path mapping, and Monet mapping method. In the latter case it is a method called table-based mapping.

### 4.1.1 Generic-Tree Mapping

A typical representative of generic mapping is a group of methods called generic-tree mapping (see [12] or [13]). These methods view XML document as a directed tree and define a general target relational schema into whose relations it can be stored.

The XML document tree has a firm structure: internal nodes are labelled with unique identifiers; leaves of the tree are labelled with either attribute or element values. Edges connecting internal nodes are labelled with element names, others with attribute names or names of elements without subelements. Outgoing edges of a node represent subelements or attributes of the element represented by the ingoing edge. An example of an XML document and its generic tree is depicted in Figure 3.

There are several methods for storing the above-described tree – so-called edge, attribute[2], universal, and normalized universal mapping. All of these approaches are briefly described in following subsections.

#### Edge Mapping

This method stores all edges of the tree in one table with the following structure:

```
Edge(source, ordinal, name, flag, target)
```

[1] It is necessary to distinguish between words “XML schema” (i.e. schema of XML document expressed in any language, e.g. DTD, XML Schema, etc.) and “XML Schema” (i.e. one of the languages).
[2] Names of graph edges are called “attributes”. The term has nothing in common with attributes in XML.

```
...
<person id=1 age=23>
  <name>Irena</name>
  <surname>Mlýnková</surname>
  <address id=2>
    <street>Podlesí 4943</street>
    <city>Zlín</city>
  </address>
</person>
<person id=3 age=30>
  <name>Jim</name>
  <surname>Beam</surname>
</person>
...
```

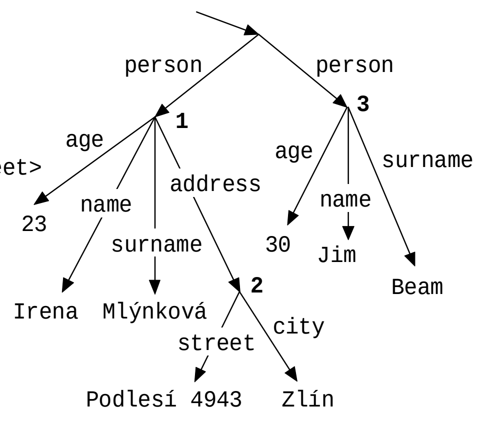


Figure 3: An example of a generic-tree

The table contains identifiers of nodes connected by the edge (`source` and `target`), name of the edge (`name`), a flag that indicates whether the edge is internal or points to a leaf (`flag`), and an ordinal number of the edge within sibling edges (`ordinal`).

### Attribute Mapping

In this kind of mapping an extra table for each edge name (so-called *attribute*) is established. Except for missing `name` column the structure of these tables is similar to the previous case:

$\texttt{Edge}_{\texttt{name}}\texttt{(source, ordinal, flag, target)}$

### Universal Mapping

This method stores edges of the tree in so-called *universal table*, which contains columns for all the attribute names described in previous method. In other words, a universal table corresponds to the result of an outer join of all tables from attribute mapping. If $a_1$, ... $a_k$ are all the attribute names in the XML document, the universal table can have the following structure:

$\texttt{Uni(source, ordinal}_{a1}\texttt{, flag}_{a1}\texttt{, target}_{a1}\texttt{, ... ordinal}_{ak}\texttt{, flag}_{ak}\texttt{, target}_{ak}\texttt{)}$

Obviously the universal table contains many null values and hence it has a great deal of redundancy.

### Normalized Universal Mapping

This method tries to solve the main disadvantage of universal mapping. The main idea is to store multi-valued attributes (i.e. edges which occur multiple times) in separate, so-called *overflow tables*. An overflow table is established for each attribute name while its structure is the same as in attribute mapping. The universal table then contains only one row per each attribute name, others are stored in corresponding overflow tables.

There is also a plenty of variations of these methods. First, in all described approaches there are two possibilities of storing values in leaves. They can be stored either in separate *value tables* (each table holds all values of a certain data type) or in additional columns of existing tables. In the latter case each table must contain columns for each possible data type and thus contains many null values.

Other, so-called *hybrid methods* can be created using combinations of the four described approaches – e.g. establishing attribute tables for frequent attributes and storing all other attributes in an edge table, etc.

### 4.1.2 Structure-Centred Mapping

Another representative of generic mapping methods, which enables to store any kind of XML document, is so-called structure-centred mapping [16]. It also views XML document as a kind of directed tree. All nodes of the tree have the same structure defined as a tuple $\nu = (t, l, c, n)$, where t is the type of the node (e.g. ELEM, ATTR, TXT...), l is the node label, c is the node content and $n = \{\nu_1, \dots \nu_n\}$ is the list of successor nodes.

The paper considers the problem how to realize mapping of the lists of successor nodes. It proposes three kinds of storage strategies focusing on speeding up the access performance. The three approaches are briefly described in following subsections.

#### Foreign Key Strategy

The foreign key storage strategy is obvious and fits plain relational databases well. Each tree node $\nu$ is simply mapped to a tuple with a unique identifier and a foreign key reference to the parent node.

The method is quite simple and the stored tree can easily be modified. Nevertheless, its disadvantage is evident – the retrieval of the data involves many self-join operations.

#### Depth First (DF) Strategy

In this storage strategy each node of the graph is given an *index value* (a couple of so-called *minimum* and *maximum DF values*), which represents its position in the graph. The values are determined in the following way: When traversing the tree in a depth first manner a counter is increased each time another node is visited. If a node is visited the first time its minimum DF value is set to the current counter value. When all child nodes have been visited, the maximum DF value is set to the current counter value. An example of the DF indexing is depicted in Figure 4.

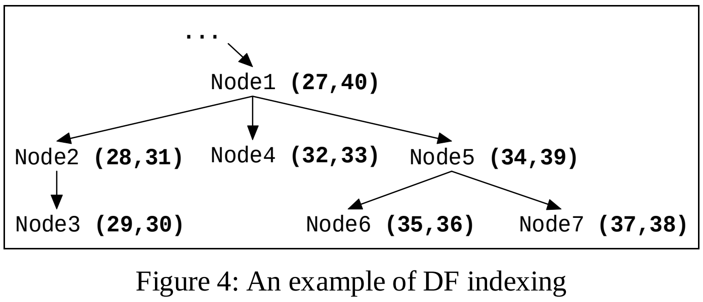


Figure 4: An example of DF indexing

Using DF values relationships of arbitrary nodes (e.g. sibling order, element-subelement relationship, etc.) can easily be determined just by comparisons – for example a node $\nu$ is contained within the subtree of node $\mu$, if the minimum DF value of $\nu$ (denoted as $\nu_{min}$) is greater than $\mu_{min}$ and $\nu_{max} < \mu_{max}$. Also retrieving a part of a document is an advantage of this method since a linear scan is possible (the nodes can be totally ordered according to DF values). The weak point of this strategy is document update – in the worst case it requires to update DF values of all nodes of the tree.

**Simple Continued Fraction (SICF) Strategy**

In this strategy each node of the graph is also given by an identification of its position. In this case the identification is called *simple continued fraction* (SICF) and is of the form

$$s = \cfrac{1}{q_k + \cfrac{1}{\cfrac{\dots}{q_2 + \cfrac{1}{q_1}}}}$$

where $q_i \in N$ ($i = 1, \dots k$) are called *partial quotients* of $\sigma$ and the expression $<q_1, \dots q_k>$ is called *partial quotient sequence*. Partial quotient sequences uniquely determine fractions and vice versa.

The SICF values are determined the following way: The root node gets a *seed value* $s \in N$, $s > 1$ (so its SICF value is $<s>$). If a node $\nu$ has SICF value $<q_1, \dots q_m>$ and has n ordered child nodes $\nu_1, \dots \nu_n$, than the SICF value for i-th child node is defined as $<q_1, \dots q_m, i>$.

The advantages and disadvantages of this strategy are similar to the previous one.

### 4.1.3 Simple-Path Mapping

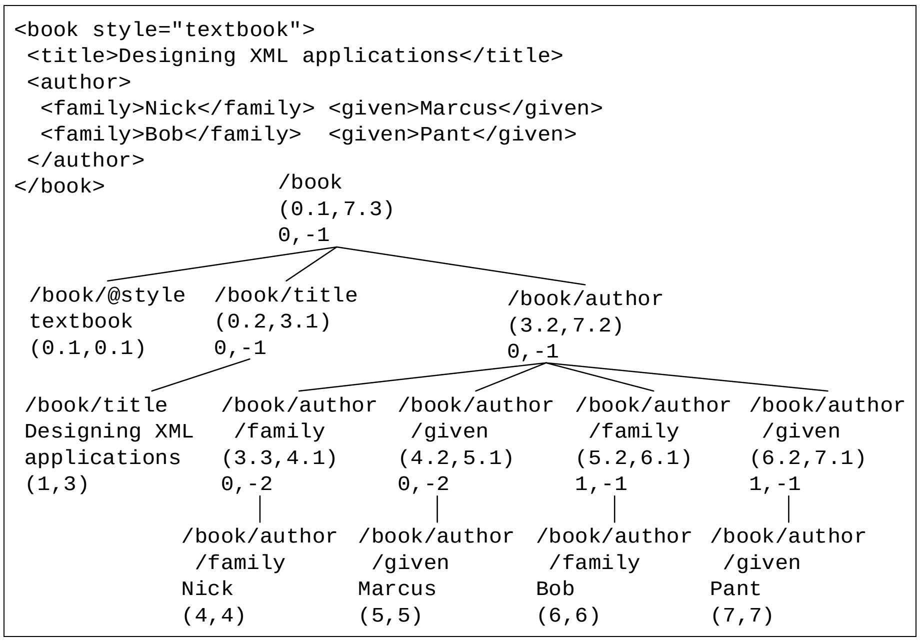
For the purpose of this paper, another example of generic mapping methods is called simple-path mapping [25]. The method also views XML document as a tree and enables to store any kind of XML document. In addition it assumes that queries over the stored XML data are expressed as path queries of XML query languages (in this case of XQL [22]).

The tree consists of three kinds of nodes – Element, Attribute, and Text nodes. Element nodes have an element type name as a label and can have zero or more children with one of the three mentioned types. Attribute nodes have an attribute name and an attribute value as a label and have no child nodes. Text nodes have character data as a label and have no child nodes too.

The main idea of the method is to decompose XML documents into so-called simple paths and to store them in the database. Each simple path expresses a basic path query of XQL and can be defined as a SimpleAbsolutePathUnit as follows:

```
<SimpleAbsolutePathUnit> ::= <PathOp> <SimplePathUnit> |
                             <PathOp> <SimplePathUnit> '@' <AttName>
<PathOp>                 ::= '/'
<SimplePathUnit>         ::= <ElementType> |
                             <ElementType> <PathOp> <SimplePathUnit>
```

Consequently, each node in the graph retains its simple path. But as a simple path contains neither order nor hierarchy information within the graph, these two are stored in the graph too. The position information (called *region*) is a pair of a start and an end value, which are assigned as follows: Each word occurrence is assigned an integer number corresponding to its position within the document. Each tag is assigned a real number – its integer part indicates the position of the preceding word and its decimal part indicates the position of the tag being concerned in the current sequence of tags. The order information is composed of occurrence plus and occurrence minus order information, which expresses the index number of the node within its parent node. An example of an XML document and corresponding simple-path tree is depicted in Figure 5.



```
<book style="textbook">
 <title>Designing XML applications</title>
 <author>
  <family>Nick</family> <given>Marcus</given>
  <family>Bob</family>  <given>Pant</given>
 </author>
</book>
```

/book (0.1,7.3) 0,-1
/book/@style textbook (0.1,0.1)
/book/title (0.2,3.1) 0,-1
/book/author (3.2,7.2) 0,-1
/book/title Designing XML applications (1,3)
/book/author /family (3.3,4.1) 0,-2
/book/author /given (4.2,5.1) 0,-2
/book/author /family (5.2,6.1) 1,-1
/book/author /given (6.2,7.1) 1,-1
/book/author /family Nick (4,4)
/book/author /given Marcus (5,5)
/book/author /family Bob (6,6)
/book/author /given Pant (7,7)

Figure 5: An example of a simple-path tree

Relations for storing XML documents (i.e. their simple-path trees) are called Element, Attribute, Text, and Path relation and have the following structure:

```
Element(docID,pathID,index,reindex,pos)
Attribute(docID,pathID,attvalue,pos)
Text(docID,pathID,textvalue,pos)
Path(pathexp,pathID)
```

The four relations store all the information about the graph. The Element, Attribute, and Text relations store information about each node type, i.e. document identifiers (`docID`), path identifiers (`pathID`), plus and minus occurrence order (`index` and `reindex`), regions (`pos`), attribute and text values (`attvalue` and `textvalue`). The relation Path stores information about simple paths, i.e. simple path strings (`pathexp`) and path identifiers (`pathID`).

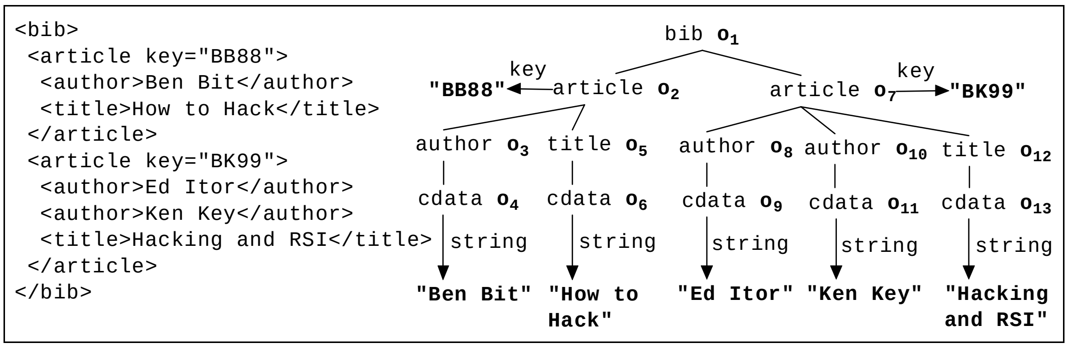
The main advantage of this method is apparent – storing simple paths of elements and attributes simplifies and speeds up processing path queries of XML query languages. Particularly processing the queries which contain "//" operator can be simply solved using SQL LIKE operator on corresponding simple paths.

### 4.1.4 Monet Mapping

For the purpose of this paper, another representative of generic mapping methods, which enables to store any kind of XML document, is called Monet mapping [24].

This method also views XML document as a tree, defined as a rooted tree d = (V, E, r, $label_E$, $label_A$, rank) with node set V, edge set $E \subseteq V \times V$ and a root node $r \in V$. The function $label_E$: $V \rightarrow$ string assigns labels to nodes (i.e. elements); $label_A$: $V \rightarrow$ string $\rightarrow$ string assigns pairs of strings (attributes and their values) to nodes. Character data are modelled as special “string” attributes of “cdata” nodes and function rank: $V \rightarrow$ int establishes a ranking to allow for an order among sibling nodes.

An example of an XML document and corresponding Monet tree is depicted in Figure 6. Identifiers $o_i$ denote object identifiers, whose assignment is arbitrary – e.g. in this case according to the depth-first traversal order.



```
<bib>
 <article key="BB88">
  <author>Ben Bit</author>
  <title>How to Hack</title>
 </article>
 <article key="BK99">
  <author>Ed Itor</author>
  <author>Ken Key</author>
  <title>Hacking and RSI</title>
 </article>
</bib>
```

Figure 6: An example of a Monet tree

The main idea of this method is based on a complete binary fragmentation of the document tree to binary associations defined as pairs $(o, .) \in oid \times (oid \cup int \cup string)$, which describe different parts of the tree (oid $\times$ oid associations represent edges, oid $\times$ string associations represent attributes, oid $\times$ int associations preserve the topology of the document). The associations, which have semantically related information, are stored in relations together. The semantically related information is in [24] related to definition of a path(o) as a sequence of (vertex and edge) labels along the path from the root node to a given node o (where $\rightarrow_e$ denotes edge to an element, $\rightarrow_a$ denotes edge to an attribute) – for example:

```
path(o3) = bib →e article →e author
path(“Ben Bit”) = bib →e article →e author →e cdata →a string
```

Each path then describes the position of an element in the graph relative to the root node. At the same time, path(o) is used to denote the type of binary association ( . , o). The paper proposes storing all associations of the same type in the same binary relation.

The advantage of this method is, that it avoids large and expensive scans over irrelevant data. On the other hand, the disadvantage is the high degree of fragmentation, which can increase efforts to reconstruct the original document or its parts.

### 4.1.5 Table-Based Mapping

A typical representative of the latter above-mentioned generic approach (i.e. the approach which enables to store only a certain collection of XML documents having similar structure) is called table-based mapping [5]. This method is based on the assumption, that the structure of XML document matches a set of tables – for example:

```
<database>
  <table1>
    <row1>
      <column1> ... </column1>
      <column2> ... </column2>
      ...
    </row1>
    ...
  </table1>
  ...
</database>
```

The mapping between elements and relations is in this case exactly defined by the structure of the XML document. Apparently, this method is suitable especially for transferring the data between two database systems.

## 4.2 Schema-Driven Mapping Methods

Schema-driven mapping methods are based on existing schema (e.g. DTD [9], XML Schema [11] [27] [2], etc.) of stored XML documents, which is mapped to relational or object-relational database schema. Into relations of this schema the data from XML documents *valid* against initial XML schema are then stored.

The purpose of these methods is to create optimal (object-) relational schema, which consists of reasonable amount of relations and whose structure corresponds to the structure of initial XML schema as much as possible. All of these methods try to improve the basic mapping idea "to create one relation for each element composed of its attributes and to map element-subelement relationships using keys and foreign keys". The main disadvantage of this idea is namely redundant and useless amount of relations, which require many join operations for data retrieval.

### 4.2.1 Common Characteristics

As mentioned in [1], schema-driven mapping methods have several common basic principles resulting from information stored in the XML schema (e.g. element-subelement relationships, minimum and maximum allowed number of occurrences, etc.). The most important ones are as follows:

- Subelements with maximum occurrence of one are (instead of to separate tables) mapped to tables of parent elements (so-called *inlining*).
- Elements with optional occurrence (i.e. with minimum occurrence of zero) are mapped to nullable columns.

- Subelements with multiple-occurrence are mapped to separate tables. Element-subelement relationships are mapped using keys, foreign keys, and referential integrity.
- Alternative subelements are mapped to separate tables (analogous to previous case) or to one universal table (with many nullable fields).
- If it is necessary to preserve the order of sibling elements, the information is mapped to a special column.
- Elements with mixed content are not usually supported, since their mapping would require many columns with nullable fields.
- Despite previous optimalizations a reconstruction of an element requires joining several tables.

### 4.2.2 Possible Classifications

The group of existing schema-driven mapping methods is bigger than in the previous case. These methods have several common features according to which they can be classified quite differently.

In this section, a few of possible classifications are discussed and corresponding representatives just mentioned. The more detailed descriptions of the mentioned methods are listed in following sections 4.2.3 to 4.2.8.

**Source XML Schema**

An obvious classification of these mapping methods is based on the type of the source XML schema. Most of these methods (e.g. Basic, Shared, and Hybrid algorithms, etc.) are based on DTD. The reason for this is the fact, that although the DTD is quite simple, it is still sufficient for most applications.

On the other hand, although the XML Schema is much more complex and thus difficult for learning, it contains useful features (e.g. data types) that DTD lacks and gives users more powerful tool for describing the allowed structure of XML documents. At present, there are also several methods (e.g. XMLSchemaStore mapping or LegoDB mapping), which try to exploit these features.

**Target Database Schema**

As in the previous case of source XML schema, the methods differ also according to the target database schema. In this paper two possibilities are concerned – relational or object-relational approach. Most of the methods are based on the former one, since the relational databases and their features managed to gain more focus than others (including object-relational ones). Despite of this fact there are several methods, which try to take the advantage of object-relational features, such as NF$^2$-relations (e.g. Hybrid object-relational mapping) or user defined data types and references (e.g. XMLSchemaStore mapping).

**Fixed and Flexible Methods**

Another classification (see [1] or [3]) according to the basic principles of schema-driven approaches includes two classes – fixed and flexible methods. *Fixed methods* (e.g. Basic, Shared, and Hybrid algorithms, Object-relational mapping, etc.) are those, which do not use any other information than the source schema itself and whose mapping algorithm is straightforward. On the other hand, *flexible methods* (e.g. LegoDB mapping or Hybrid object-relational mapping) are methods, which do use the additional information (usually query statistics, element statistics, etc.) and focus on creating an optimal schema for a certain application.

### 4.2.3 Algorithms Basic, Shared, Hybrid, and Derived Algorithms

The best-known representative of fixed schema-driven mapping methods is a group of three algorithms called Basic, Shared, and Hybrid [23]. These algorithms describe possible ways of mapping a DTD to relational schema. Their main idea is based on a definition of a directed graph, so-called *DTD graph*, which represents the structure of processed DTD. Nodes of the graph are elements (which appear in the graph exactly once), attributes, and operators (which appear in the graph as many times as they appear in the DTD). Edges of the graph represent element-attribute, element-subelement or element-operator and operator-subelement relationships.

An example of a DTD and corresponding DTD graph is depicted in Figure 7.

```
<!ELEMENT author(name?,surname)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT surname(#PCDATA)>
<!ELEMENT book(author*,title)>
<!ATTLIST book published CDATA>
<!ELEMENT title(#PCDATA)>
<!ELEMENT article(author)>
<!ATTLIST article paper CDATA>
```

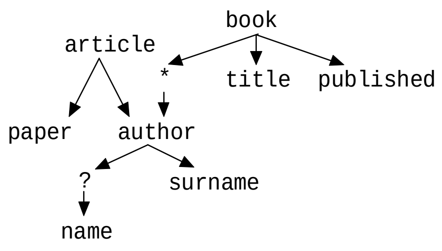


Figure 7: An example of a DTD graph

One important fact is, that each DTD is first pre-processed and simplified to contain only "?" and "*" operators and flat expressions (see [23]).

These algorithms try to gradually improve the idea "to create one relation for each element" using different kinds of inlining approaches. They differ according to the amount of redundancy they may cause.

#### Basic Algorithm

The Basic algorithm combines two approaches:

- to inline as many descendants of an element as possible into the corresponding relation and
- to create a relation for each element in the DTD graph.

In the former case only two kinds of element-subelement relationships are solved using keys and foreign keys – subelements with multiple occurrence (indicated by the use of "*" operator) and recursion (indicated by cycles in the DTD graph). The reason for the latter approach is the fact, that a valid XML document can be rooted by any element in the DTD.

The main disadvantages of this algorithm are obvious – a huge amount of unnecessary relations and a great deal of redundancy since an element node can be represented in several relations.

**Shared Algorithm**

The Shared algorithm tries to avoid the drawbacks of Basic ensuring that an element is represented in exactly one relation. The idea is to identify elements that are in Basic represented in multiple relations and to share them by creating separate relations for them and using keys and foreign keys. The mapping rules are as follows:

- Nodes with an in-degree of one are inlined to parent relations.
- Nodes with an in-degree of zero are stored in separate relations, because they are not reachable from any other node.
- Repeated elements are stored in separate relations.
- Of all mutually recursive elements having an in-degree one, one of them is stored in a separate relation.
- The problem of inlined elements, which can become roots of an instance XML document, is solved using a flag for each element that indicates this situation.

Apparently the main advantage of the Shared algorithm is the reduced amount of relations and redundancy. Its main disadvantage is the number of join operations necessary for restoring an element, which can be worse than in Basic.

**Hybrid Algorithm**

The Hybrid algorithm tries to combine the join reduction properties of Basic with the sharing features of Shared. The algorithm is similar to Shared except for additional inlining. In addition, Hybrid inlines elements with an in-degree greater than one, that are neither recursive nor reached through a “*” node.

Order of elements can be easily incorporated into all three approaches by storing a position field for each element.

**CPI Algorithm**

Another representative of fixed schema-driven mapping, which also describes a way of mapping a DTD to relational schema is a method called CPI (Constraints-Preserving Inlining) algorithm [17]. This algorithm can be based e.g. on previously mentioned Hybrid algorithm while its main purpose is to capture not only the structure of the given DTD but the semantic constraints which can be found in the DTD as well.

The considered constraints are e.g. domain constraints (i.e. restrictions to a specific set of values), cardinality constraints (i.e. cardinality relationships between an element and its subelements expressed using “+”, “*”, “?” or no operator), referential integrity (i.e. ID, IDREF, IDREFS types), etc. These constraints are represented using corresponding SQL constraints e.g. [NOT] NULL, CHECK, UNIQUE, PRIMARY/FOREIGN KEY, etc.

### 4.2.4 Object-Relational Mapping

Object-relational mapping (see [4] [5] [6]) is another example of fixed schema-driven mapping method. The word “object-relational” is a bit confusing, since it does not denote the target schema but the two steps of the algorithm. The source XML schema is expressed either in DTD or XML Schema; the target database schema is relational in all cases.

The two mentioned steps are:

1. The source XML schema is mapped to an object schema expressed in some object-oriented language. (This step apparently results from the idea of XML data binding technology.)
2. The object schema is mapped to a database schema.

Obviously, if the object schema is not essential, the two mappings can be joined and the intermediate object schema eliminated. The object schema models the source XML schema as a tree of objects. It is important to understand, that these objects are specific for each XML schema and thus have nothing in common with objects in the DOM tree model. Moreover, although we speak about classes and objects, the referred-to classes are closer to C structs rather than C++ or Java classes because they have no behaviour (i.e. no methods).

**Mapping DTD to Object Schema**

In this step element types with PCDATA-only content and attribute types are considered as *simple* types. Element types with element or mixed content, or element types with attributes are considered as *complex* types. The mapping rules can be summed as follows:

- Simple types are mapped to scalar data types.
- Complex types are mapped to classes with each element type in the content model mapped to a property of the class.
- The data type of each property is either the scalar data type to which the referenced element is mapped or a pointer/reference to the corresponding object.
- Attributes are mapped to properties.
- Subelements in a sequence or a choice are mapped to properties (whereas in the latter case the corresponding columns in the relational schema will be nullable).
- Repeated subelements are mapped to multi-valued properties of (un)known size.
- Mixed content is mapped to a multi-valued property of unknown size for storing PCDATA-values together with additional *order columns* for each property sharing the same order space.

An example of a DTD, its object schema (and corresponding relational schema) is depicted in Figure 8.

```
<!ELEMENT A(B,C)>         class A {          Table A
<!ELEMENT B(#PCDATA)>        String b;          Column b
                             C      c; }        Column c_fk

<!ELEMENT C(D,E)>         class C {          Table C
<!ELEMENT D(#PCDATA)>        String d;          Column d
<!ELEMENT E(#PCDATA)>        String e;          Column e
<!ATTLIST E F CDATA)>        String f; }        Column f
                                                Column c_pk
```

Figure 8: An example of object-relational mapping for DTD

**Mapping XML Schema to Object Schema**

The first step for the case of XML Schema is similar to the previous one. The differences are related to additional features XML Schema in comparison to DTD has. The most interesting ones are mapped as follows:

- Simple data types are mapped to corresponding scalar types. Default and fixed values are mapped to default and constant properties of corresponding classes.
- Abstract complex types are mapped to abstract classes; extension of complex types is mapped to class inheritance.
- Model groups and attribute groups can be mapped either to classes or can be considered as a macro-like replacement mechanisms and thus eliminated.
- Wildcard nodes are generally mapped to arbitrary types (such as Object in Java or pointer to void in C++).
- Identity constraints can be in special cases mapped to keys, foreign keys, and unique values; generally their mapping is difficult or impossible.
- Substitution groups are handled by pre-processing the source XML schema – each occurrence of a substitution group is replaced with corresponding choice group.

An example of an XML Schema document, its object schema (and corresponding relational schema) is depicted in Figure 9.

```
<schema>
 <complexType name="A">                class A {        Table A
  <attribute name="b" type="int"/>        Int b;           Column b
  <sequence>                              C   c; }         Column c_fk
   <element name="c" type="C"/>
  </sequence>
 </complexType>

 <complexType name="C">                class C {        Table C
  <sequence>                              String    d;      Column d
   <element name="d" type="string"/>      String(5) e;      Column e
   <element name="e" type="E"/>           Date      f; }    Column f
  </sequence>                                               Column c_pk
  <attribute name="f" type="date"/>
 </complexType>

 <simpleType name="E">
  <restriction base="string">
   <maxLength value="5"/>
  </restriction>
 </simpleType>
</schema>
```

Figure 9: An example of object-relational mapping for XML Schema

**Mapping Object Schema to Relational Schema**

The second step of the mapping is in both cases obvious:

- Classes are mapped to tables (known as *class tables*).

- Scalar properties of the classes are mapped to columns of class tables.
- Pointer/reference properties are mapped to primary key/foreign key relationships.
- Multi-valued properties are mapped to separate tables (known as *property tables*).

Furthermore, in the case of XML Schema mapping:

- Simple data types are mapped to corresponding database data types.
- Superclasses and subclasses can be mapped either to two tables with one-to-one relationship or to one common table.
- Arbitrary types are mapped to a general data type, such as BLOB.

An example for the case of DTD is depicted in Figure 8, an example for the case of XML Schema in Figure 9.

### 4.2.5 Constraints Preserving Mapping

Another representative of fixed schema-driven mapping methods, which also describes mapping of XML Schema structures to relational schema is called constraints preserving mapping algorithm [26]. The main focus of this method is to preserve not only the structure of the given XML schema but also the variety of semantic constraints XML Schema enables to express.

First, the paper proposes a formal representation of XML Schema structures based on the regular tree grammar called FD-XML (for exact definition see [26]). Second, an extension of ER model, so-called EER model, is proposed and the FD-XML is converted into EER schema. Then, the EER schema is simplified and optimized, preserving the correct data structure as well as the semantic constraints and finally, the simplified EER schema is converted to relational schema.

The EER model extends ER model in following three features:

1. Arrowheads starting from the parent element and ending at the subelement are used to express parent-child relationships.
2. Cardinalities (n, m) are used to represent the allowed interval of occurrences.
3. Accessories are given to EER model in order to preserve the data constraints.

The rules for converting XML Schema structures to EER model are as follows:

1. Represent every element in the set of non-terminals of the FD-XML (e.g. elements, complex types, etc.) as an entity, which is denoted by a rectangle box.
2. Build up the relationships between entities according to the set of element production rules of the FD-XML (e.g. rules expressing type assignments, parent-child relationships etc.), which are denoted by diamond boxes.
3. Build up the entities' attributes according to the set of attribute production rules of the FD-XML (e.g. rules expressing element-attribute relationship, etc.) and add the corresponding occurrence times.
4. Keep the set of terminal symbol data types and their constraints, the set of primary keys and unique constraints and the set of foreign keys as accessories of the EER model.

An example of XML Schema document and corresponding EER model is depicted in Figure 10.

```
<schema>
 <element name="E1" type="T1"/>
 <complexType name="T1">
  <sequence>
   <element name="E2" type="string"/>
   <element name="E3" type="int"/>
   <element name="E4" type="decimal"
            minOccurs="0"/>
  </sequence>
  <attribute name="A1" type="string"/>
 </complexType>
</schema>
```

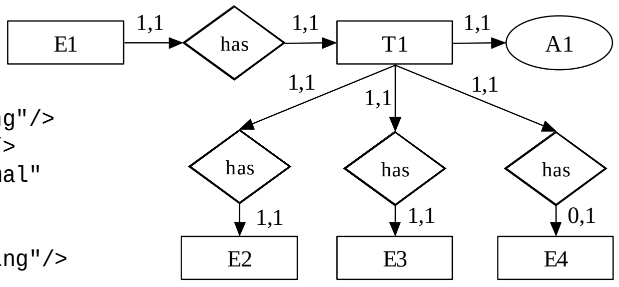


Figure 10: An example of an EER diagram

The rules for simplification of the EER schema are as follows:

1. An entity is converted to its parent entity's attribute if it satisfies the following:
   a. The entity has a unique parent entity.
   b. The entity possesses no subentity.
   c. The parent-children entities meet the qualification of 1,1→1,1 or 1,1→0,1.
2. The subentity is removed from its parent entity that satisfies the following:
   a. The entity possesses no attribute.
   b. The entity has no or just one parent entity.
   c. The entity possesses only one subentity and meets the qualification of 1,1→1,1 or 1,1→0,1.

The rules for transferring the simplified EER schema to relational schema are as follows:

1. Map an entity to a relation and the entity's attributes to the relation's attributes.
2. For every relation add a primary key attribute called <entity name> + "ID".
3. Map a relationship between entities to a relation and all entities' keys as well as attributes of the EER relationship to the relation's attributes.
4. Merge the relations possessing same primary keys.
5. According to the primary key constraint set establish the primary key constraints.
6. According to the foreign key constraint set establish the foreign key constraints.
7. Establish all the attribute data types as well as their constraints.

### 4.2.6 XMLSchemaStore Mapping

Another example of fixed schema-driven mapping method proposing mapping XML Schema structures to object-relational schema, particularly the one defined in SQL:1999 standard [19] [20], is (according to the experimental implementation of the method) called

XMLSchemaStore mapping method [21]. The method results from the idea of XML data binding technology and from many object-oriented features XML Schema has. It tries to preserve the structure as well as semantic constraints of the source XML schema in the target schema and to exploit object-relational features of the SQL:1999 standard. The mapping rules are as follows:

- Both built-in and user-defined simple types are mapped to corresponding database simple types (eventually) together with corresponding integrity constraints.
- Complex types and model groups are mapped to object-relational user-defined types (UDT), whereas:
    - XML attributes are mapped to UDT attributes with corresponding simple type.
    - Simple element content is mapped to UDT attribute with corresponding simple type.
    - Element-subelement relationship is mapped to UDT attribute, whose type is (according to the allowed occurrence and the type of the subelement) either the UDT of the subelement or the reference/array of references to the UDT.
- Deriving of complex types is mapped to UDT inheritance.
- Elements are (according to the allowed occurrence and the type of the element) mapped either to own type table or to a type column of the type table which corresponds to their parent element.

The resulting object-relational schema can be then described as a set of type tables "interconnected" using references.

The mapping algorithm is based on traversing a directed graph called ***DOM graph***, whose ordered edges determine the "order" in which the UDTs and corresponding type tables should be created to follow reference properties. The DOM graph results from the structure of a DOM tree of the given XML Schema file. It can be created the following way:

- The original edges of the DOM tree are directed to express the "direction" of element-subelement or element-attribute relationship.
- New edges expressing the "direction" of the usage of globally defined items (e.g. elements, complex types, etc.) are added.

An example of an XML Schema file and corresponding DOM graph is depicted in Figure 11. The solid lines correspond to original edges of the DOM tree; dash-and-dot lines are the additional ones.

The mapping is done while traversing the graph starting in the `schema` node. First, all descendants of a current node are processed, e.g. the corresponding UDTs and type tables are created. Second, the current node can be processed, since all necessary object-relational items already exist.

```
<schema>
 <complexType name="T1">
  <sequence>
   <element ref="E1"/>
  </sequence>
  <attribute name="A1" type="T2"/>
 </complexType>
 <element name="E1" type="string">
 <simpleType name="T2">
  <restriction base="string">
   <length value="5"/>
  </restriction>
 </simpleType>
</schema>
```

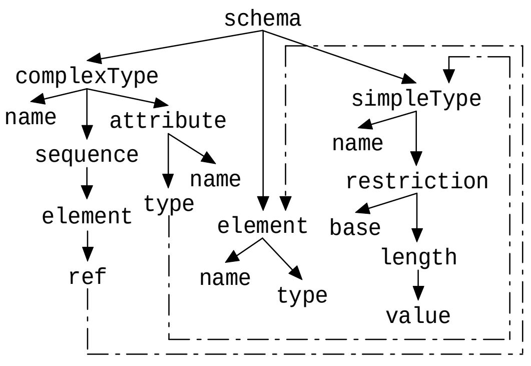


Figure 11: An example of a DOM graph

### 4.2.7 LegoDB Mapping

A representative of flexible schema-driven mapping methods is an algorithm proposed in LegoDB system [3]. First the method defines fixed mapping of XML Schema structures (for processing simplicity rewritten into syntactically simpler, but semantically equivalent so-called *p-schemas*) to relations. The flexibility is based on the idea to explore a space of possible XML-to-relational mappings and to select the best one according to given statistics. The statistics include both data information about a sample set of XML documents (such as sizes of data types, minimum and maximum values, number of distinct values, etc.) and query workloads (i.e. a set of sample queries and their importance).

In order to select the best mapping, the system in turns applies the following two steps to the source p-schema, until a good result is achieved:

1. Any possible XML-to-XML transformation is applied to the p-schema resulting in a new one.
2. XML-to-relational transformations are applied to the new p-schema and against the resulting relational schema the given queries are estimated.

As the space of possible p-schemas can be relatively large (possibly infinite), the paper also proposes a greedy evaluation strategy that explores only the most interesting subset.

The XML-to-XML transformations used in the algorithm are:

- *Inlining/outlining* – nesting/unnesting a subelement within its parent element
- *Union factorization/distribution* – possible partitioning of a union of (sub)elements into groups
- *Repetition merge/split* – exploitation of the relationship between sequencing and repetition defined by the rule: `(a+)=(a,a*)`
- *Wildcards rewriting* – exploitation of distributing the (implicit) union in the wildcard defined by the rule: `~=(e1|(~!e1))`, where ~ indicates “any element” and `~!e1` indicates “any element except `e1`“
- *From union to options* – exploitation of the fact, that a union is contained in a sequence of optional parts, defined by the rule: `(t1|t2)`$\subset$`(t1?,t2?)`

Unlike the first four transformations, the last one does not exactly preserve the semantics of the source schema. On the other hand it enables inlining of an union, which might improve the performance for certain queries, although it results in tables with large number of null values.

The XML-to-relational transformations are similar to those described in the previously mentioned methods:

- Create one relation for each element.
- For each relation create a key, that will store the identifier of corresponding element.
- For each relation create foreign keys to the identifiers of all parent elements.
- For each relation create an attribute for each inlined subelement.
- If the element is optional then the corresponding column can contain a null value.

### 4.2.8 Hybrid Object-Relational Mapping

Another example of flexible schema-driven mapping methods concerning mapping DTDs to object-relational schemes is called hybrid object-relational mapping (see [14] or [15]). The main idea of this method is connected with the word "hybrid" – it tries to improve the straightforward mapping of all elements and attributes in a DTD to relations, which can in case of semistructured data lead to large database schemes, by storing structured parts of the DTD in relations and semistructured parts in a more natural way. For the latter case the method assumes the existence of so-called *XML data type*, which stores XML fragments and supports the evaluation of path expressions as well as fulltext operations.

The main concern of this approach is to determine the optimal structure of the target database (i.e. to decide which parts of the DTD are structured and which semistructured). The suggested algorithm is as follows:

1. A graph (similar to above-described DTD graph) representing the hierarchy of elements and attributes of the DTD is built.
2. A *measure of significance* $v$ is determined for each element/attribute in the graph.
3. The resulting database design is derived from the graph.

The measure of significance can be expressed as

$$v = \frac{1}{2} v_S + \frac{1}{4} v_D + \frac{1}{4} v_Q$$

while the used variables have the following meaning:

- $v_S$ (weight derived from the DTD structure) – the combination of values expressing the position of the element/attribute in the graph together with possible alternatives and quantifications within the element/attribute,
- $v_D$ (weight derived from the existing XML data) – the ratio of the number of documents containing the element/attribute and the absolute number of documents,
- $v_Q$ (weight derived from the queries) – the ratio of the number of queries containing the element/attribute and the absolute number of queries.

According to a given limit (which influences the level of detail of the resulting database schema) the algorithm determines the nodes in the graph, each of which fulfils the following conditions:

- It is not a leaf of the graph.
- The node and all its descendants are below the given limit.
- There exists no predecessor of the node that fulfils the same conditions.

All subgraphs consisting of these nodes and their descendants are replaced by an XML attribute.

The resulting graph is finally mapped using a fixed mapping method to object-relational schema. The mapping focuses on the use of structured or nested attributes in NF$^2$-relations, assuming the existence `set-of` and `tuple-of` constructors. The mapping rules are as follows:

- Simple elements and attributes are mapped to relational attributes.
- Sequences of elements are mapped to `tuple-of` relational attributes; optional elements are mapped to nullable attributes.
- Repeatable elements and set-valued attributes are mapped to `set-of` relational attributes.

An example of a DTD and its corresponding object-relational mapping is depicted in Figure 12. It uses the following notation: <> represents a tuple constructor, {} represents a set constructor, and a key of nested relation is in bold face.

```
<!ELEMENT chapter(ctitle, section+)>     chapter=<id,ctitle,{section}>
<!ATTLIST chapter id ID REQUIRED>
<!ELEMENT ctitle(#PCDATA)>

<!ELEMENT section(stitle, paragraph+)>   section=<id,stitle,{paragraph}>
<!ATTLIST section id ID REQUIRED>
<!ELEMENT stitle(#PCDATA)>

chapter = <id,ctitle,{<id,stitle,{paragraph}>}>
```

Figure 12: An example of object-relational mapping

## 4.3 User-Defined Mapping Methods

User-defined mapping methods are most often used in commercial systems. This approach requires that the user first defines a target relational or object-relational schema and then expresses required mapping using a system-dependent mechanism, e.g. a special query language, a declarative interface or using annotations added into stored XML document. At present, most of existing systems (e.g. Oracle9i, IBM DB2 or Microsoft SQL Server) support some kind of user-defined mapping.

Obviously, this approach is the most flexible one, as it allows users to specify the schema they really need. On the other hand, as mentioned in [3], it has also disadvantages – it requires large development effort and moreover mastering of two complex and distinct technologies (XML and database systems). It might be also quite difficult even for an expert to determine a good mapping for complex applications.

The description of existing mechanisms for expressing the required mapping is not in the focus of this paper.

## 5 Discussion of Mapping Methods

As can be seen from the above summary the storage strategies have many common features and just few but thus more striking differences. These are discussed and evaluated in this section.

Obviously there is probably no reasonable argument for comparing generic and schema-driven methods together. Generally these two approaches are too different seeing that they result from quite distinct basic ideas. Apparently we can say that generic methods are independent on the XML schema of stored XML documents and thus enable to store any document. But on the other hand, a similar argument can be used for schema-driven methods, which try to exploit the information from the existing XML schema and hence enable the user to influence the structure of the resulting (object-) relational schema. That is why the methods of these two approaches are in this section discussed separately.

### 5.1 Generic Methods

Omitting the extreme case of table-based mapping (and resembling techniques), which is proposed only for a certain group of XML documents, all generic methods follow the same idea – to store any kind of document regardless its structure. All of these methods also quite naturally view XML document as a tree whose nodes correspond to single items of XML documents and (ordered) edges express the relationships between the nodes.

The main disadvantage of all mentioned methods is, that despite of the general view on the XML document they omit such document-centric items as comments, CDATA sections, etc. This feature is probably adopted only for simplicity since the graph-view on the document as a set of items with a certain hierarchy could probably include them as well.

Second point to discuss is a mixed content of elements. The ability to support this feature essentially depends on the basic structure of the graph – e.g. in case of Generic-tree mapping method this type of element content can be supported quite hardly, whereas, e.g., the Structure-centred mapping method should have no problems with mixed content at all. In the former case, there is probably no other solution to this problem than to treat the mixed content elements as elements with text content, preferably together with some XML-aware fulltext enhancement.

Another point to discuss concerns data types of element and attribute values. In general all values in XML documents are considered as texts – the idea of data types is related to XML schemes (especially to those using XML Schema structures) and the validity of XML documents. On the contrary, for some purposes it can be useful to distinguish at least the basic ones. This feature is for example included in Generic-tree mapping methods.

On the other hand, all of these techniques support preserving the order of sibling elements; furthermore the ordering can be total and thus can be used for indexing the data. This advantage apparently results from the basic design of these methods.

All of the mentioned features are summarized in Table 1.

| Described methods | Considered features | | | |
|---|---|---|---|---|
| | **Comments, CDATA sections...** | **Mixed content elements** | **Different data types** | **Preserving the element order** |
| Generic-tree mapping | Not supported | Not supported | Supported | Sibling order |
| Structure-centred mapping | Not supported | Supported | Not supported | Total ordering |
| Simple-path mapping | Not supported | Supported | Not supported | Sibling order |
| Monet mapping | Not supported | Supported | Not supported | Sibling order |

Table 1: Summary of generic mapping features

Generally speaking, these methods are most suitable in cases when no XML schema exists. All of the above-described methods are primarily determined for data-centric XML documents, but probably with extensions related to the above-mentioned document-centric items they could be used for document-centric documents as well.

## 5.2 Schema-Driven Methods

More discussible methods are apparently the schema-driven ones. Regardless the kind of source or target schema, there are XML structures whose mapping is quite obvious and thus in most cases similar since they have a suitable equivalent in (object-) relational world (e.g. attributes, sequences of elements, etc.), whereas others (e.g. alternatives, mixed content, etc.) are quite difficult to express using database features. Thus the methods can be evaluated according to the ability to handle these problematic structures. Another evaluation factor can result from the basic classification of XML documents and the corresponding level of round tripping. Although this problem is mainly related to document-centric XML documents, whereas all of the mentioned techniques are designed especially for data-centric ones, they usually consider some of the document-centric features (e.g. the sibling order) that can be interesting for data-centric approach as well.

As mentioned above, the main point is again the mixed content of elements. Several of the described methods support this feature using one of two possible approaches. The first one is the same as was mentioned in the previous section – to treat mixed content as a text content together with some XML-aware text enhancements (e.g. the XML data type mentioned in Hybrid object-relational mapping method). The second approach (used in Object-relational mapping method) is to store the mixed content element as any other element and to store the text parts in a separate table or multi-valued property (depending on the target schema) together with the ordering information for both the texts and the subelements.

Another point to discuss is preserving the order in the XML document. The information does not have to be stored for all items in the document. The order is significant for sibling elements within their parent element but not for attributes of one element. Hereafter it is significant for multi-valued data types (i.e. for both attributes and elements) and as was already mentioned for mixed content elements. All of the above-mentioned methods support (or can be easily extended to support) preserving the order of XML items.

Next important issue is the idea of flexible mapping methods. There is no reason for asking whether flexible methods are better than the fixed ones – apparently they are since the resulting schema suits the given statistics at least as well as corresponding fixed method. Indeed these methods can be obviously used only if it is possible to obtain necessary statistic information. The interesting point is how to determine the “best” schema. Just two but nevertheless quite different representatives (i.e. LegoDB mapping method and Hybrid object-relational method) were mentioned, whereas both are somehow based on a sample set of XML documents and typical queries. As was already mentioned, the most flexible mapping is provided by user-defined mapping methods. But this approach is quite extreme since it retains the whole problem and the right decisions on the user.

Last but not least is the matter of target database schema. Most of the mentioned methods focus on relational schema and thus must quite "unnaturally" solve the problems of multi-valued properties, type derivation (in case of XML Schema structures), etc. These are the main reasons why several methods (i.e. XMLSchemaStore mapping method or Hybrid object-relational mapping method) focus on object-relational schema and its object-oriented features.

All of the mentioned features are summarized in Table 2.

| **Described methods** | **Considered features** | | | | |
|---|---|---|---|---|---|
| | **Source schema** | **Target schema** | **Mixed content elements** | **Preserving the element order** | **Flexibility** |
| Basic, Shared, Hybrid, and CPI mapping | DTD | Relational | Not considered | Can be extended | Fixed |
| Object-relational mapping | DTD/XML Schema | Relational | Supported using additional fields | Supported | Fixed |
| Constraints preserving mapping | XML Schema | Relational | Not considered | Not considered | Fixed |
| XMLSchemaStore mapping | XML Schema | Object-relational | Not supported | Supported | Fixed |
| LegoDB mapping | XML Schema | Relational | Not considered | Not considered | Flexible |
| Hybrid object-relational mapping | DTD | Object-relational | Supported using XML-aware data type | Can be extended | Flexible |

Table 2: Summary of schema-driven mapping features

To sum up, schema-driven mapping methods try to exploit the information in the given XML schema as much as possible. Although they can preserve some document-centric features (e.g. document order or mixed content elements), they are primarily designed for data-centric XML documents.

## 6 Conclusion

This survey was trying to offer a general and clear summary of existing strategies for connecting XML and database technologies, especially those related to relational and object-relational systems. Several possible classifications were mentioned and discussed and the best-known representatives of the classes were briefly described. Finally the general common features, advantages, and disadvantages of the described methods were discussed.

According to the overview it is possible to say, that there are several areas, which will probably be in the main focus of future works. The first will apparently concern semantic constraints (especially those, which can be expressed using XML Schema structures) that should be preserved in the target (object-) relational schemes. Several of the mentioned methods particularly focused on this area, but the current features of database systems and relational languages still limit these approaches in many ways.

The second interesting point is connected with flexible mapping methods, which try to optimize the fixed schema according to its probable future use. As there are no rules, which define a "good" XML schema (such as, e.g., normal forms for relational schemes), the fixed mapping of a "bad" one can result in a "bad" relational schema as well. Thus an important task may be to determine a definition of a "good" XML schema and ways how to establish it.

Although, apparently not all existing methods could be mentioned and the described classifications are not the only possible ones, we hope that the survey is broad enough and gives a clear overview of most of the existing approaches.

## References

[1] S. Amer-Yahia, M. Fernàndez: *Overview of Existing XML Storage Techniques*. AT&T Labs, 2001.

[2] P. V. Biron, A. Malhotra: *XML Schema Part 2: Datatypes*. W3C Recommendation, 2001. `www.w3.org/TR/xmlschema-2/`.

[3] P. Bohannon, J. Freire, P. Roy, J. Siméon: *From XML Schema to Relations: A Cost-Based Approach to XML Storage*. In Proc. of Int'l Conf. on Data Engineering, 2002.

[4] R. Bourret, C. Bornhövd, A. P. Buchmann: *A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases*. WECWIS 2000, Milpitas, CA.

[5] R. Bourret: *Mapping DTDs to Databases*. XML.com, 2001.

[6] R. Bourret: *Mapping W3C Schemas to Object Schemas to Relational Schemas*. 2001. `www.rpbourret.com`.

[7] R. Bourret: *XML and Databases*. 2003. `www.rpbourret.com`.

[8] R. Bourret: *XML Database Products*. 2003. `www.rpbourret.com`.

[9] T. Bray, J. Paoli et al.: *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, 2000. `www.w3.org/TR/REC-xml/`.

[10] L. Wood, V. Apparao, S. Byrne et al.: *Document Object Model (Core) Level 1*. W3C Recommendation, 1998. `www.w3.org/TR/REC-DOM-Level-1/`.

[11] D. C. Fallside: *XML Schema Part 0: Primer*. W3C Recommendation, 2001. `www.w3.org/TR/xmlschema-0/`.

[12] D. Florescu, D. Kossmann: *A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database*. Technical Report 3684, INRIA, March 1999.

[13] D. Florescu, D. Kossmann: *Storing and Querying XML Data Using an RDBMS*. IEEE Data Engineering Bulletin, 22(3), 1999.

[14] M. Klettke, H. Meyer: *Managing XML Documents in Object-Relational Databases*. Rostocker Informatik Fachberichte, 24, 1999.

[15] M. Klettke, H. Meyer: *XML and Object-Relational Database Systems – Enhancing Structural Mappings Based on Statistics*. In Informal Proc. WebDB Workshop, pages 151 – 170, 2000.

[16] A. Kuckelberg, R. Krieger: *Efficient Structure Oriented Storage of XML Documents using ORDBMS*. Springer-Verlag Heidelberg, Volume 2590, pages 131 – 143, 2003.

[17] D. Lee, W. W. Chu: *CPI: Constraints-Preserving Inlining Algorithm for Mapping XML DTD to Relational Schema*. Journal of Data & Knowledge Engineering 39, pages 3 – 25, 2001.

[18] D. Megginson: *SAX*. SourceForge, 1998. `www.saxproject.org`.

[19] J. Melton: *Advanced SQL: 1999 - Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann Publishers, 2003.

[20] J. Melton, A. R. Simon: *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann Publishers, 2002.

[21] I. Mlýnková: *XML Schema and its Implementation in Relational Databases*. Master thesis, Charles University, 2003. (In Czech)

[22] J. Robie, J. Lapp, D. Schach: *XML Query Language (XQL)*. In Proc. of QL'98 – The Query Languages Workshop, Cambridge, Mass., 1998.

[23] J. Shanmugasundaram, K. Tufte, G. He, Ch. Zhang, D. DeWitt, J. Naughton: *Relational Databases for Querying XML Documents: Limitations and Opportunities*. In VLDB, pages 302 – 314, 1999.

[24] A. Schmidt, M. Kersten, M. Windhouwer, F. Waas: *Efficient Relational Storage and Retrieval of XML Documents*. In Proc. of WebDB, pages 47 – 52, 2000.

[25] T. Shimura, M. Yoshikawa, S. Uemura: *Storage and Retrieval of XML Documents using Object-Relational Databases*. In Proc. DEXA Conf., 1999.

[26] H. Sun, S. Zhang, J. Zhou, J. Wang: *Constraints-Preserving Mapping Algorithm from XML-Schema to Relational Schema*. Springer-Verlag Heidelberg, Volume 2480, 2002.

[27] H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn: *XML Schema Part 1: Structures*. W3C Recommendation, 2001. `www.w3.org/TR/xmlschema-1/`.