Management of XML Documents in Object-Relational Databases

Thomas Kudrass, Matthias Conrad
Leipzig University of Applied Science,
Department of Computer Science and Mathematics, D-04251 Leipzig
kudrass@imn.htwk-leipzig.de matthias@conradius.de

Abstract. Many applications deal with complex XML documents that need to be made persistent. We investigate the use of the object-relational modeling approach that combines both the strengths of object oriented concepts and relational DBMS technology provided by current DBMS vendors. As a prerequisite, we analyze the document definition and generate the equivalent object-relational database schema in SQL. We discuss the main issues when mapping XML documents to an object-relational target schema with the example of Oracle. The paper concludes with a summary of our experiences with the object-relational approach for XML document mangement.

1 Motivation

One of the most important issues of the XML community is the mapping of XML documents to databases. XML provides the required flexibility for the data exchange in heterogeneous information systems and for the development of content management systems. It is undisputed that only database systems as storage platform can guarantee the necessary features such as powerful and efficient search capabilities, transaction-oriented storage und concurrency control. There are different approaches how XML can be represented in database systems. These approaches can be classified according to the data model of the target schema. Therefore, we distuinguish between relational, object-oriented and native XML representation of XML documents in a database [1]. Another aspect to describe the XML-to-database storage approach is the question of how generic the mapping algorithms work. There is a number of relational transformation algorithms, proposed by [5,9], that analyze the document structure only and map the data of a document to generic tables, e.g., edge tables or attribute tables. Although these algorithms are applicable for data-centric documents, they pose some serious limitations. Among them is the high degree of decomposition of the source documents, which turns the upload of a document into a large number of relational insert operations [6]. Another limitation is the information loss as the result of using these algorithms. That affects information that is not considered data of a document - such as comments or processing instructions. Other known transformation problems are the handling of mixed content

and external entities that have to be transformed properly when being transferred into databases and vice-versa. This issue is known as *round-trip* problem.

Content-oriented approaches focus on the data of an XML document, i.e., the schema of the document is the source of the definition of the database schema. Thus, the underlying methods analyze the document schema, typically defined by a DTD or an XML Schema, and transform it according to a general algorithm into the corresponding database structure. Some systems allow the user to intervene into the transformation by individual definitions. Many commercial users hesitate to invest into another DBMS to deal with XML objects. Instead, they are interested in exploiting the full functionality of relational database systems, which have been developed towards object-relational systems. The blend of object-oriented concepts with the established relational database technology is a promising platform for the storage of XML documents with any complexity using the schema information. For it, we have built the prototype of a storage system with Oracle (versions 8i and 9i) to evaluate the feasability of the available object-relational technology for XML document management.

The paper is organized as follows: Section 2 gives an overview about the object-relational techniques as they are used for the modeling of XML documents. Section 3 presents the structure of the utility program *XML2Oracle* we developed to capture the document definition. Section 4 discusses the main issues of the object-relational mapping methodology we had to deal with. Some special issues are treated in section 5. Section 6 introduces the meta-data structures that support the mapping algorithms. In section 7, we give an assessment of our experiences as conclusion about the feasibility of the chosen approach.

2 Object-Relational Concepts - A Short Overview

2.1 Object Types

User-defined datatypes (UDTs) represent the most important object-oriented concept provided by Oracle. They are very similar to the user-defined types as introduced in SQL:1999 [7]. Since the implementation has been done on the Oracle platform, the used SQL syntax from Oracle is a subset of the complete SQL3 standard.

An object type bundles three components: type name, attribute list, method list. The following example shows a simple object type without methods.

```
CREATE TYPE Type_Professor AS OBJECT(
    PName               VARCHAR(80),
    Subject             VARCHAR(120));
```

The statement above defines a datatype that can be used as domain of another attribute. The result is a relational schema that is not in the first normal form because it may contain non-atomic attributes that can be arbitrarily complex.

```
CREATE TYPE Type_Course AS OBJECT(
    Name        VARCHAR(100),    // atomic domain
    Professor   Type_Professor); // non-atomic domain
```

There are two possibilities to create a table using object types:

A table can be defined as object table, i.e., the type of the tuples corresponds with the object type. Object tables are treated like usual tables in SQL queries. The definition of object tables can include constraints. In our example, the attribute `PName` is defined as primary key. Note that the definition of a constraint is bound to the definition of the table - not to the definition of the object type.

```
CREATE TABLE TabProfessor OF Type_Professor(
    PName        PRIMARY KEY);
```

Another way to create tables with object types is to use the object type as attribute domain as the following example shows:

```
CREATE TABLE Course_Offering(
    Department  VARCHAR(120),
    Course      Type_Course);
```

Using object-valued attributes, the SQL queries must use constructors, which have the same name as the type and a list of parameters equivalent to the type attributes. The following sample INSERT statement uses two constructors, `Type_Course` and `Type_Professor`.

```
INSERT INTO Course_Offering
VALUES ('CS', Type_Course ('CAD Intro',
        Type_Professor ('Jaeger','CAD')));
```

2.2 Collection Types

Oracle provides two kinds of collection types: arrays and nested tables. The SQL:1999 standard provides just arrays a the only collecton type.

The first collection type provided by Oracle is the array of variable length, denoted as VARRAY. The array definition comprises the name of the array, the maximum length and the domain of the elements. Note that there are some restrictions on arrays in Oracle 8. In Oracle 8, the element type must not be another collection type (array or nested table) or a large object type. Oracle 9 eliminates this restriction and accepts any element type in a collection. Although the array type enables the efficient storage of complex values, they can only be retrieved or stored as a whole array in SQL. The following example defines an array type, `TypeVA_Subject`. It consists of at most 5 VARCHAR(200) elements that denote names of subjects taught at university.

```
CREATE TYPE TypeVA_Subject AS
    VARRAY(5) OF VARCHAR(200);
```

The second available type of collection is nested table. Unlike VARRAYs, they enable us to store an unlimited number of elements. A nested table can be defined as in the following example:

```
CREATE TYPE Type_TabSubject AS
    TABLE OF VARCHAR(200);

CREATE TABLE TabProfessor (
    Name        VARCHAR(80),
    Subject     Type_TabSubject)
NESTED TABLE Subject STORE AS TabSubject_List;
```

In this example, a table type, Type_TabSubject, is defined whose elements are strings of no more than 200 characters. The table elements can be of a user-defined object type or a large object type. This table type is used as domain of the attribute Subject in the table TabProfessor. The STORE AS clause determines to store the content of the table-valued attribute Subject in TabSubject_List, a separate table. Nested tables are managed by the DBMS using internal foreign keys.

2.3 Object References

Oracle supports the concept of object identifiers that are managed for row objects, i.e., objects of object tables. Therefore, object identifiers can be used as references to row objects. For that purpose a new data type, REF, has been introduced in Oracle to establish relationships between objects, analogously to the foreign key relationship in the relational model. The following example defines a relationship between course and professor, assuming that a course is offered by (exactly) one professor.

```
CREATE TYPE Type_Course AS OBJECT(
    Name     VARCHAR(200),
    Prof_Ref REF Type_Professor);
CREATE TABLE TabCourse OF Type_Course;
```

All referenced objects must be of the same object type, but can be stored in different tables. Thus, the usage of REF to define relationships is more flexible than foreign keys, which are restricted to two tables. On the other hand, there is a useful construct (SCOPE FOR) that defines the table which a referenced object can belong to. Scoped values can be dereferences. The REF concept of Oracle complies with the SQL:1999 standard.

3 Capturing the XML Document Information

A prerequisite to store an XML document in an object-relational database is the definition of a data structure that is appropriate for the document structure. To capture the XML document information we wrote the program *XML2Oracle* with a graphical user interface [3]. This program analyzes both the XML document and the corresponding DTD using two parsers. One parser analyzes the XML document and checks well-formedness and validity of the document. The second - non-validating - parser analyzes the DTD only and transforms it into a DTD Document Object Model (DOM). The XML parser used in our program is provided in the Oracle XML Developers Kit (XDK) [11]. Among the few available DTD parsers we chose the parser by Wuttka [10].

Figure 1 represents the function of *XML2Oracle* and the use of both parsers. *XML2Oracle* produces two DOM trees that represent the logical structure of the documents. The DOM tree for the XML documents shows the elements and their values as well as the attributes and their values. The DTD is also represented in a tree structure considering constraints, such as occurrence and optionality of elements. The DTD tree representation is the precondition for the definition of the database schema. Appendix A shows a sample document that is used throughout the paper.

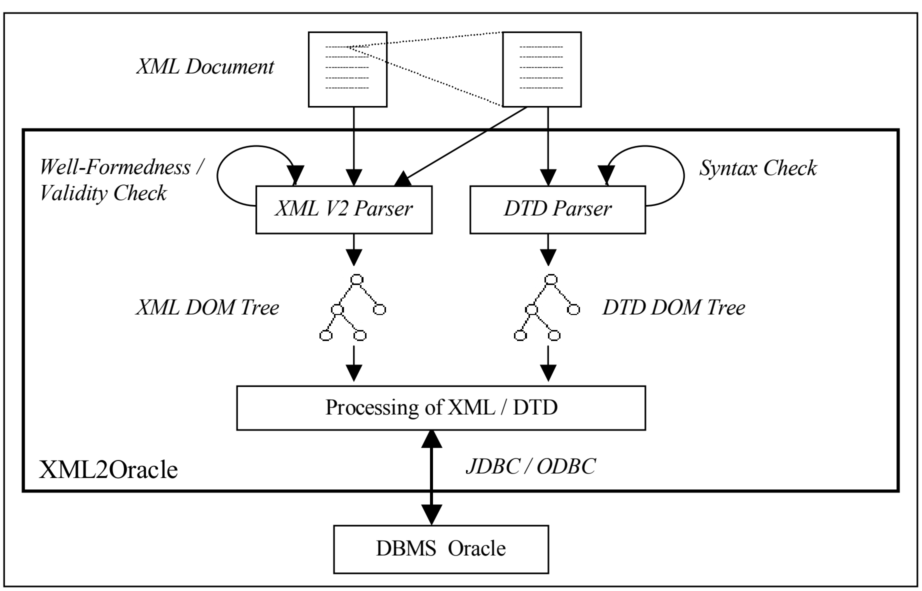


Figure 1 Parsing DTD and XML in *XML2Oracle*

4 Mapping XML Documents to Object-Relational Databases

The DTD tree representation is the input for the generation algorithm producing an SQL script. This script can be executed afterwards without any modification to create and populate the database tables and representing XML documents. The overall structure of the mapping procedure is shown in figure 2.

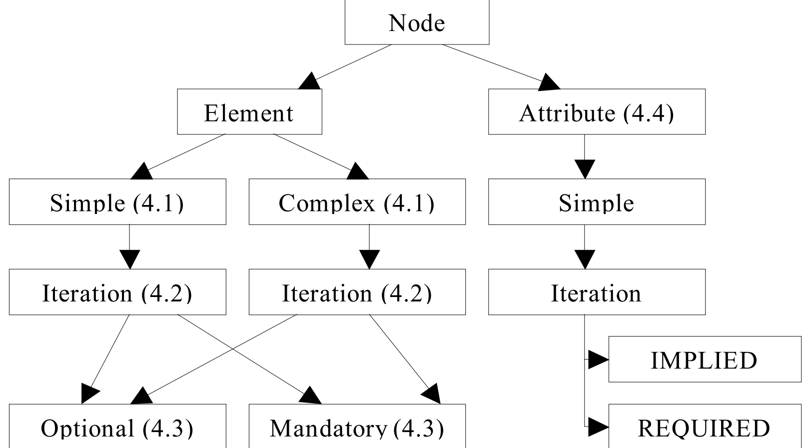


Figure 2 Overall Structure *of the Mapping Algorithm*

Each case to be treated is marked with the corresponding subsection number. A node of the tree represents either an element or an attribute in the XML document. We distinguish simple elements and elements that can be decomposed into subelements (complex elements). The multiple occurrence and the optionality of the elements have to be considered as well when deriving the equivalent database definition. The algorithm works for all possible combinations of the cases mentioned above.

4.1 Mapping of Elements

Simple Elements

Simple elements contain character data; they are assigned the #PCDATA type in a DTD. An element that consists of simple subelements only can be represented in an object-relational database as follows: First, an object type is defined with a name derived from the parent element, using some predefined naming conventions (see section 5). All subelements of type #PCDATA are represented as attributes of the object type. The attribute domain is defined as VARCHAR. The name of an attribute is determined by the name of the corresponding subelement. Secondly, that object type is used as the tuple type for the table that is created with it. Due to the lack of element datatypes in a DTD there is no way to restrict the type of the table attributes. For that reason, our mapping schema generates VARCHAR(4000) as default attribute type in order to avoid value assigment conflicts. However, the limitation of the string length is a serious restriction regarding XML documents with chunks of unstructured text, particularly document-oriented documents.

Complex Elements

It has been shown that the relational mapping of complex elements leads to an enormous decomposition of the documents into tables, sometimes termed *shredding*. The relational approach becomes worse if the elements are nested repeatedly. Therefore, we pursue the idea of a one-to-one mapping of an XML document into a complex database object representing its content and logical structure. The basic idea of our mapping algorithm is as follows: For each element type in a DTD, a corresponding object type is created in the database. Hence, subelements of an element are represented as attributes of the equivalent object type. If a subelement is of type #PCDATA, the domain is created as for simple elements. If the subelement contains further subelements, a new object type is defined that is used as domain for the attribute that represents the parent subelement.

The aggregation of SQL object types enables an XML document of any nesting depth to be mapped to an object-relational database. The result is a relation that is composed of object types that may be composed of other object types. It reflects the logical structure of the source XML document.

The advantage of the object-relational schema becomes obvious when executing INSERT and SELECT statements. In our example, the insertion of an XML document would produce three separate INSERT statements, which would also require the crea-

tion of primary keys. Using an object-relational approach requires a single INSERT query for one document as the following example shows:

```
INSERT INTO TabUniversity
VALUES ('Computer Science',
 Type_Student('23374','Conrad','Matthias',
  Type_Course('CAD Intro',
    Type_Professor('Jaeger','CAD','Computer Science'),
    '4'))));
```

The element data to be stored are passed to the appropriate constructor method.

The advantages of the object-relational representation also become evident when information has to be retrieved from the document. The object structure can be traversed using the dot notation without executing join operations.

```
SELECT S.attrLName
FROM TabUniversity S
WHERE S.attrStudent.attrCourse.attrProfessor.attrPName
      = 'Jaeger';
```

The sample query above retrieves the family names of students who have subscribed to a course of Professor Jaeger.

4.2 Iteration Operators

The multiple occurrence of XML elements can be marked with a '*' or a '+' operator in the DTD, denoted as set-valued elements. The representation of set-valued elements can be implemented by collection types in an object-relational DBMS. However, some workaround solution is necessary in systems which only provide limited set type constructors, such as Oracle 8.

Collection type constructors can easily be applied to set-valued elements with text-valued subelements as the following example shows. There are two alternatives: nested tables and arrays.

```
<!ELEMENTProfessor(PName,Subject+,Department)>
<!ELEMENTSubject (PCDATA)>
```

```
CREATE TYPE Type_Subject AS
TABLE OF VARCHAR(4000);

CREATE TABLE TabProfessor(
...attrSubject Type_Subject,
...)
NESTED TABLE attrSubject ...;
```

```
CREATE TYPE TypeVA_Subject AS
VARRAY(9) OF VARCHAR(4000);

CREATE TABLE TabProfessor(
...attrSubject TypeVA_Subject,
...);
```

The usage of the Oracle8 collection types is not possible for set-valued subelements in Oracle 8 due to the type restrictions (cf. section 2.2).

For each complex object that contains a '+' or '*' operator, an individual object type is created with the same name as the corresponding element type in the DTD. Its subele-

ments are represented as attributes of the object type provided that they are not set-valued and complex. A relationship between the parent element and its set-valued complex subelement can be expressed as reference, which appears as REF attribute in the object type definition that represents the subelement. The reference to another object type requires the existence of an object table of that type because the reference relationship is established between OIDs, which are properties of objects. Therefore, a one-to-many relationship can be mapped to a REF-valued attribute pointing to the parent element, analogously to the foreign key relationship between relations. The approach of using REF attributes proves weak when dealing with many-to-many relationships because that would require the introduction of additional object types - analogously to the relationship table.

Example

Another drawback of the modeling alternative above appears when the database is populated from XML documents: Since XML elements do not have a user-defined identifier, it is hard to generate the appropriate INSERT statements. For them, the identifier of the object to be referenced has to be retrieved. This may require querying *all* attributes of an object to identify the object that takes part in the relationship. We introduced an additional unique attribute for the sole purpose of simplifying the generation of INSERT operations that establish relationships with other elements.

With Oracle 9i it is now possible to create arbitrarily nested collection types, which would solve the problems of set-valued elements as described above[4].

```
CREATE TYPE TypeVA_Course AS
  VARRAY(100)OF Type_Course;
CREATE TYPE TypeVA_Professor AS
  VARRAY(100)OF Type_Professor;
CREATE TYPE TypeVA_Subject AS
  VARRAY(100)OF VARCHAR(4000);
CREATE TYPE Type_Professor AS OBJECT(
    attrPName       VARCHAR(4000),
    attrSubject     TypeVA_Subject,
    attrDept        VARCHAR(4000));
CREATE TYPE Type_Course AS OBJECT(
    attrName        VARCHAR(4000),
    attrProfessor   TypeVA_Professor,
    attrCreditPts   VARCHAR(4000));
CREATE TYPE Type_Student AS OBJECT(
    attrStudNr      VARCHAR(4000),
    attrLName       VARCHAR(4000),
    attrFName       VARCHAR(4000),
    attrCourse      TypeVA_Course);
CREATE TABLE TabUniversity(
    attrStudyCourse VARCHAR(4000),
    attrStudent     TypeVA_Student);
```

In our prototype, we chose the VARRAY collection type; nested tables work in nearly the same manner. Thus, the implementation of complex elements works as follows: For each set-valued element of a DTD, a VARRAY collection type is created and named according to predefined conventions - as our example shows. Complex elements and their children are represented by object types and their attributes. The resulting object type can be used as domain of elements in an array type. For example, an array type `TypeVA_Professor` with elements of type `Type_Professor` is defined for the document element `Professor`. This array is assigned to the attribute `attrProfessor` within the object type `Type_Course`. Thus, it is possible to express a many-to-many relationship between professor and course by using set-valued attributes in both objects. The use of nested collection types saves the work with explicit references to represent relationships. The effect is a more natural modeling of an XML document in an object-relational system. The intermediate tree representation of the document type definition produced after the analysis step can be represented as the definition of only one table with simple, complex and set-valued attributes. The data of complex subelements that will be stored in objects within the table are passed via constructor methods called within the INSERT statement. Thus, an INSERT operation to fill the database table from the document's top element looks as following:

```
INSERT INTO TabUniversity VALUES('Computer Science',
 TypeVA_Student(
  Type_Student('23374','Conrad','Matthias',
   TypeVA_Course(
    Type_Course('Database Systems II',
     TypeVA_Professor(
      Type_Professor('Kudrass',
       TypeVA_Subject('Database Systems',
                    'Operat. Systems'),
        'Computer Science')),'4'),
    Type_Course('CAD Intro',
     TypeVA_Professor(
      Type_Professor('Jaeger',
       TypeVA_Subject('CAD','CAE'),
        'Computer Science')),'4'),
     ...)),
  Type_Student('00011','Meier','Ralf',...)...)
  ...);
```

Note that the SQL:1999 standard - like Oracle 8 - excludes the nesting of arrays, which would aggravate a solution as shown above.

4.3 Not-Null Constraints

XML supports the concept of null values through optional element types and attributes. If the value of an optional element type or attribute is null, it is not included in the document. Optional elements are defined with the operators '?' (zero or one) or '*' (zero or many). XML attributes can be defined as #IMPLIED in the DTD. In those cases a document is still valid when the element or attribute value does not occur. Such ele-

ments or attributes have to be mapped to nullable columns - the default for every non-prime attribute of a relation.

Elements that are not characterized as optional or set-valued elements defined with the '+' operator are considered mandatory in the DTD. The same applies to attributes that are defined as #REQUIRED. The corresponding attributes in the database have to be defined as NOT NULL attributes. Note that constraints (such as NOT NULL) can only be defined in the object table - not in the definition of the object type. The current restrictions for constraints affect the definition of set-valued and object-valued attributes. Set-valued attributes cannot be defined as NOT NULL altogether. It is only possible to prohibit null values in inner attributes of complex attributes using CHECK constraints. However, also CHECK constraints can only be defined in a table or object table.

The following example illustrates the problems when using CHECK clauses for NOT NULL constraints:

```
<!ELEMENT Course (Name,Address?)>
<!ELEMENT Address (Street,City?)>

CREATE TYPE Type_Address AS OBJECT
    attrStreet  VARCHAR(4000),
    attrCity    VARCHAR(4000));
CREATE TYPE Type_Course AS OBJECT
    attrName    VARCHAR(4000),
    attrAddress Type_Address);
CREATE TABLE TabCourse OF Type_Course(
    attrName    NOT NULL,
CHECK (attrAddress.attrStreet IS NOT NULL));
```

According to the DTD, the subelement `Address` is optional in the XML document. If the `Address` element exists in the document, then its subelement `street` must exist as well. The following INSERT statement produces a desired error message because it is not allowed to create a new address with a city but without a street.

```
INSERT INTO TabCourse
VALUES('CAD Intro',Type_Address(NULL,'Leipzig'));
```

Let's assume a new course is inserted into the `TabCourse` table without any address data:

```
INSERT INTO TabCourse
VALUES('Operating Systems',NULL);
```

The second INSERT statement conflicts with the declaration in the DTD and produces also an error message. Since the element `Address` does not exist, the corresponding object attribute is assigned a NULL value. However, the DBMS checks for the existence of the attribute `attrStreet` within the attribute `attrAddress`, which results in a non-desired error message. Therefore, the use of CHECK constraints for optional complex element types is not recommendable. The provided modeling features of Oracle do not allow to define NOT NULL constraints for subelements of complex element types that are optional. Likewise, NOT NULL constraints cannot be applied to collection types.

4.4 Mapping of Attributes

Attributes can be defined with some additional information in XML. The main difference between attributes and elements is that attributes cannot be nested. Instead, they are assigned string values only. Possible types of attributes are: ID, IDREF, CDATA, and NMTOKEN.

XML attributes are treated like simple elements when being stored in a database. Since an XML attribute is just a string, it is mapped to an attribute of a table or an object type with the VARCHAR datatype assigned. Attributes that are defined as #REQUIRED are represented as NOT NULL columns in the database.

```
<!ATTLIST StudentNr SNumber CDATA #REQUIRED> //DTD Def.

CREATE TYPE Type_StudentNr AS OBJECT(       //DB Def.
    attrSNumber     VARCHAR(4000),
    ...);

CREATE TABLE TabStudent OF Type_Student(
    attrSNumber     NOT NULL,
    ...);
```

In order to keep the relationship between element and attribute in the generated database schema, an object type is defined that stores both the element and the attribute. Thus, the resulting object type is assigned the simple element - unlike simple elements without any attributes (cf. section 4.1). The mapping methodology is illustrated with the following example:

```
                        CREATE TYPE TypeA AS OBJECT(
<!ELEMENT A (B,...)             attrB   TypeB,
<!ELEMENT B (PCDATA)>           ...);
<!ATTLIST B             CREATE TYPE TypeB AS OBJECT(
    C CDATA IMPLIED             attrB    VARCHAR(4000),
    D CDATA IMPLIED>            attrListBTypeAttrL_B);
                        CREATE TYPE TypeAttrL_B AS OBJECT(
                                attrC    VARCHAR(4000),
                                attrD    VARCHAR(4000));
```

If an element is described in more detail by attributes in the DTD, then an object type is created for the attribute list with a name according to our naming conventions (see section 5). The attributes of the object type are derived from the XML attribute list.

Elements can reference other elements identified by an ID attribute through IDREF attributes. A mapping of those attributes into simple VARCHAR database columns would ignore their semantics. Instead, IDREF attributes must be represented as REF-valued columns in the database pointing to the referenced element. The target element is stored in an object table; its ID attribute is mapped to a VARCHAR column. This mapping rule requires determining in advance which ID attribute is referenced by an IDREF value. This kind of information cannot be captured from the DTD, rather from the XML document.

5 Meta-Data about XML Documents

The object-relational mapping algorithms, as they are used in our system, cause some loss of information. So it cannot be determined if a table column or an object type has been derived from an element or an attribute in the source XML document. Another problem is the generation of names in the target database. It must be avoided that element names may conflict with SQL keywords (e.g., `ORDER`). Further, the uniqueness of the generated names of database objects has to be guaranteed. The introduction of naming conventions for the generation of the database schema helps to distinguish between identical names stemming from different document types. Table 1 shows the naming conventions used in *XML2Oracle*.

Naming Convention	Object Semantics
Tab*Elementname*	Name of a table
attr*Elementname*	Name of a DB attribute derived from a simple XML element (table or object type column)
attr*Attributename*	Name of a DB attribute derived from an XML attribute (table or object type column)
attrList*Elementname*	Name of a database attribute that represents an XML attribute list
ID*Elementname*	Name of a primary key or foreign key attribute
Type_*Elementname*	Name of an object type derived from an element name
TypeAttrL_*Elementname*	Name of an object type generated for an attribute list
TypeVA_*Elementname*	Name of an array
OView_*Elementname*	Name of an object view

Table 1: Naming Conventions in *XML2Oracle*

In addition, *XML2Oracle* maintains a meta-table during the transformation to capture information about the source XML document. Each XML document to be stored is assigned a unique `DocID` to identify it in the database. Further meta-information regard document name, `DocName`, document location, `URL`, and prolog information, such as the character set. The structure of the meta-table looks as follows:

```
TabMetadata (DocID: INTEGER, DocName: VARCHAR,
URL: VARCHAR, SchemaID: VARCHAR, NameSpace: VARCHAR,
XMLVersion: VARCHAR, CharacterSet: VARCHAR,
Standalone: CHAR, DocData: TypeVA_DocData, Date: DATE)
```

The attribute `DocData` represents an array of `DocData` objects:

```
Type_DocData (XML_Type: String, XML_Name: String,
DB_Name: String, DB_Type: String, NameSpace: String)
```

The meta-table solves the naming issue for elements by inserting a schema identifier that is generated for each newly created schema by *XML2Oracle*. That `SchemaID` is combined with the naming conventions to generate unique names of tables and object types. `SchemaID`s are necessary to deal with identical element names from different DTDs. Those elements may have different subelements, which would result in errors when generating the database schema. Note the restriction imposed by a DBMS regarding the maximum name length of identifiers (e.g., Oracle accepts only 30 characters).

Another choice to deal with synonymous elements is the usage of namespaces. A namespace can be defined for a single element or a whole document. Accordingly, the namespace definitions are stored in the meta-table as well. The attribute `DocData` has been introduced to capture whether an attribute in the database has been derived from an XML element or an XML attribute (attribute `XML_Type`). Furthermore, information about its naming (`DB_Name`) and its type (`DB_Type`) are stored.

6 Special Issues

6.1 Representation of Entities

Entity references can be inserted into documents. *XML2Oracle* expands them at their occurrences so that the expanded entities are stored in the database. Therefore, the information about the original entity definition has been lost. Also, a problem arises with how to store markup characters that are not used for markup. These are stored using the lt, gt, amp, quot, and apos entities. The XML parser (e.g., the parser used in *XML2Oracle*), transforms those entity references into the corresponding character literals that are stored in the database. So it is not possible to retrieve the original document from the database. A solution for that issue is to extend the meta-database by information about the entities. For them, an object type could be defined which comprises both entity reference and substitution text. The information about internal entities defined in the DTD can be captured from the output of the DTD parser used in *XML2Oracle*. When the document is retrieved from the database the characters can be replaced by the original entity references that can be found in the meta-table.

6.2 Non-hierarchical and Recursive Relationships

The usage of a tree as an intermediate data structure implies restrictions for some documents. Recursive relationships between document elements cannot be adequately represented in a tree. The same applies to elements with multiple parents as the example in figure 3 shows:

An element that occurs more than once with different parent elements in the DTD is represented repeatedly as node in the generated DTD tree, for example the element `Address`. In such cases a graph should be the preferred data structure.

A DTD can be designed in such a way that an element can be part of any other element. Hence, recursive relationships between elements may occur. The schema generation algorithm applied in *XML2Oracle* is not appropriate for this kind of recursion because it would execute infinite loops. Therefore, we describe a methodology to cope with recur-

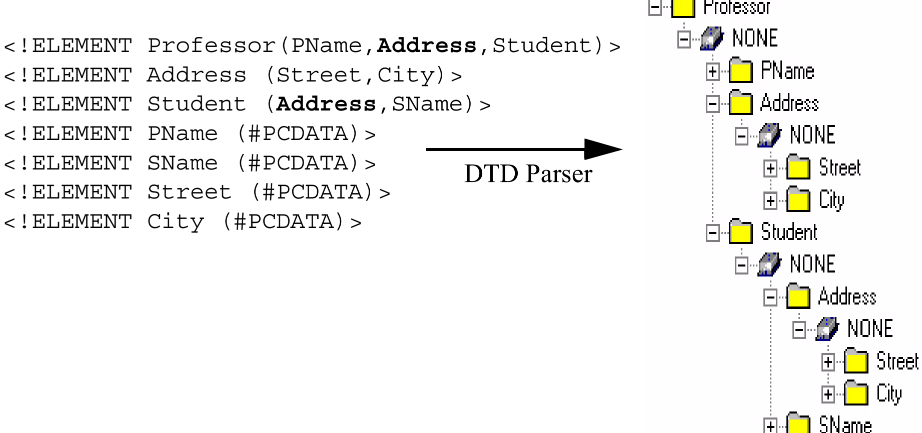


Figure 3 Mapping of Non-Hierarchical Relationships

sive relationships. The basic idea is to use REF-valued attributes to treat recursion as illustrated here:

```
<!ELEMENT Professor
          (PName,Address,Course*,Subject?,Dept)>
...
<!ELEMENT Dept (DName,Professor*)>
```

The complex element `Professor` comprises - among others - the subelement `Dept`. This element is complex itself and contains the element `Professor` as subelement. To implement this structure in an object-relational DBMS, an object type has to be defined for the element `Professor`. It comprises an attribute `attrDept` (derived from the subelement `Dept`) that is assigned the object type `Type_Dept`. That object type itself contains a collection-typed attribute `attrProfessor` (e.g., nested table) because `Professor` is a set-valued subelement of `Dept`. The nested table stores only references to the object type. Note that an object table has to be created for the object type `Type_Professor` because references can only point to objects of an object table.

```
CREATE TYPE Type_Professor;
CREATE TABLE TabRefProfessor AS
    TABLE OF REF Type_Professor;
CREATE TYPE Type_Dept AS OBJECT(
    attrDName     VARCHAR2(4000),
    attrProfessor TabRefProfessor);
CREATE TYPE Type_Professor AS OBJECT(
    attrPName     VARCHAR2(4000),
    attrDept      Type_Dept );
```

Since all types are related, the deletion of any type must be propagated to all dependents by using DROP FORCE statements in SQL.

6.3 Using Object Views

Besides supporting the creation of tables with object types as structured column values, Oracle also supports the creation of database views that can deliver structured rows of data. Database views can be used in combination with user-defined object types to create structured logical views based on one or more tables or views [8]. Let's assume a relational schema has been generated from the DTD as it has been described in known mapping algorithms [2].

The following example shows an object view representing data of a relational schema - without considering set-valued elements.

```
CREATE VIEW OView_University AS
    SELECTType_University(u.attrStudyCourse,
     Type_Student(s.attrStudNr, s.attrLName,
                 s.attrFName,
      Type_Course(c.attrName,
       Type_Professor
         (p.attrPName,p.attrSubject,p.attrDept))))
     AS University
    FROM tabUniversity u, tabStudent s,
         tabCourse c, tabProfessor p
    WHEREs.IDStudNr = c.IDStudNr AND
         c.IDCourse = p.IDCourse;
```

We begin by creating user-defined types from the given DTD according to the methodology described in section 4. Next, we create an object view, `OView_University`, to superimpose the correct logical structure on top of a join of four physical tables with information about students, courses and professors. It uses the `Type_University()` constructor to create instances of `Type_University()` objects. In the same way, it uses the `Type_Student()` constructor to create instances of `Type_Student` objects within `Type_University`. Nested as an attribute to the `Type_Student()` constructor is the `Type_Course()` constructor to create an instance of `Type_Course`. The same applies to the `Type_Professor()` constructor that is nested within the `Type_Course()` as well. Object views can be applied in template-driven mapping procedures, i.e., SELECT queries on the object view can be embedded into XML template documents. This can be exploited by software utilities that transfer data from object-relational databases to XML documents.

In order to transform a simple set-valued element represented as a separate table in the relational model into a collection object type, the collection is dynamically computed using the keywords CAST and MULTISET. The following example shows a piece of the resulting view for the set-valued element `Subject` that is assigned to each `Professor` element in the DTD.

```
...Type_Professor (p.attrPName,
CAST (MULTISET (SELECT s.attrSubject
               FROM  tabSubject s
               WHERE p.IDProfessor = s.IDProfessor)
     AS TypeVA_Subject), p.attrDept), ...
```

7 Conclusions and Future Work

We have presented a number of mapping techniques that can be deployed to represent an XML document in an object-relational DBMS. The algorithms use the document schema information as they are stored in a DTD. We discovered advantages and drawbacks of using the object-relational approach for the transformation of DTDs.

Advantages:

- user-defined datatypes as adequate representation of document elements
- allows non-atomic domains, more natural representation of XML documents of any complexity
- multiple nesting of XML elements
- simple database queries by using dot notation, tight correspondence with XPath expressions
- uniform identity of every element in the database by object identifiers
- relationships between elements via object references (REF-valued attributes)

Drawbacks:

- set-valued complex elements cannot be mapped to collection types due to system limitations (Oracle 8i only), the same applies to the SQL:1999 standard
- NOT NULL constraints cannot be adequately expressed
- usage of references does not preserve the order of elements
- distinction between element and attribute requires additionally the maintenance of metadata
- loss of document information: comments, processing, instructions,entity references, prolog
- little flexibility in case of changes to the DTD, any change implies the adaptation of the database schema
- no type concept in DTDs -> simple elements and attributes can only be assigned the VARCHAR datatype in the database
- restricted maximum length of the VARCHAR datatype

Our work has shown that it is not necessary to purchase another - native XML - DBMS in order to manage XML documents in databases, provided that the available relational DBMS is enhanced by the necessary object-relational extensions. Our prototype implementation has revealed the strengths of the object-relational approach regarding the structural complexity, but also some weaknesses that could be overcome by further efforts in the future. One of the main limitations is caused by the lack of definition capabilities in a DTD. Hence, one of the next tasks is to start with the analysis of documents with XML Schema, which provides more advanced concepts (such as element types). For the intermediate representation of XML documents a graph structure should be preferred in order to cope with recursive relationships. To store text elements in the database some more flexibility is required beyond the available VARCHAR datatype. Large text elements should be assigned the CLOB type. Our approch as we presented in this

paper can be developed further by enhancing the meta-database to consider comments, processing instructions, entity references and their location within the document.

The use of an object-relational DBMS as storage engine for XML documents supports the coexistence of different storage models. XML datatypes currently provided by RDBMS vendors focus mainly on the implementation of XML documents as CLOBs (*Character Large Objects*). Therefore, we have to investigate the extended XML support announced for Oracle 9i; the Release 2 (XML DB) uses the object-relational approach in a very similar way as we have proposed here. Our work has proved that there are more alternatives to make use of object-relational database technology with a broad range of storage choices and query capabilities. In particular, data-centric applications that exchange structured data can benefit from our work.

Acknowledgement

This work has been funded by the Saxonian Department of Science and Art (Sächsisches Ministerium für Wissenschaft und Kunst) through the HWP program.

References

[1] R. Bourret: XML and Databases, 2000. http://www.rpbourret.com/xml/XMLAndDatabases.html

[2] R. Bourret: Mapping DTDs to Database, 2001. http://www.rpbourret.com/xml/index.htm (on: http://www.xml.com).

[3] M. Conrad: Speicherung von XML-Dokumenten mit bekanntem Schema in objektrelationalen Systemen am Beispiel Oracle, Diplomarbeit (German), HTWK Leipzig, 2001.

[4] B. Chang, M. Scardina, S. Kiritzov: Oracle 9i XML Handbook. Maximize XML-enabled Oracle 9i, Oracle Press Osborne/Mc Graw Hill, 2001.

[5] D. Florescu, D. Kossmann: Storing and Querying XML Data using an RDBMS, Data Engineering, Sept. 1999, Vol.22, No.3.

[6] T. Kudrass: Management of XML Documents without Schema in Relational Database Systems, OOPSLA Workshop on Objects, <XML> and Databases, Tampa, Oct. 2001.

[7] J. Melton, A. Simon: SQL:1999 - Understanding Relational Language Components. Morgan Kaufmann, 2001.

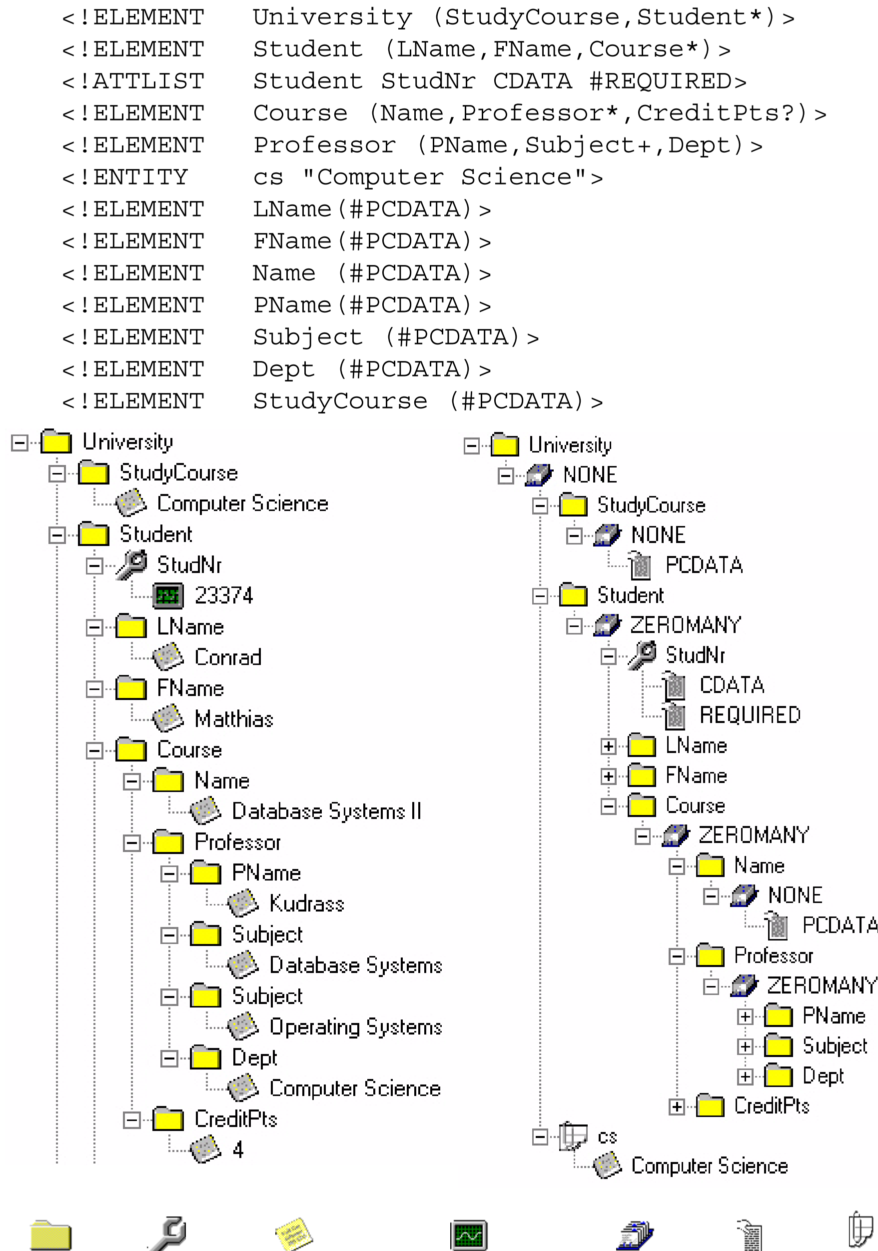
[8] S. Muench: Building Oracle XML Applications, O'Reilly & Associates, 2000.

[9] J. Shanmugasundaram et. al: Relational Databases for Querying XML Documents: Limitations and Opportunities, Proc. 25th VLDB conference, 1999.

[10] M. Wutka: DTD Parser, 2001, http://www.wutka.com/dtdparser.html

[11] Oracle Corp. XML Developers Kit, http://www.oracle.com/xml

Appendix A: Sample Document



```
<!ELEMENT   University (StudyCourse,Student*)>
<!ELEMENT   Student (LName,FName,Course*)>
<!ATTLIST   Student StudNr CDATA #REQUIRED>
<!ELEMENT   Course (Name,Professor*,CreditPts?)>
<!ELEMENT   Professor (PName,Subject+,Dept)>
<!ENTITY    cs "Computer Science">
<!ELEMENT   LName(#PCDATA)>
<!ELEMENT   FName(#PCDATA)>
<!ELEMENT   Name (#PCDATA)>
<!ELEMENT   PName(#PCDATA)>
<!ELEMENT   Subject (#PCDATA)>
<!ELEMENT   Dept (#PCDATA)>
<!ELEMENT   StudyCourse (#PCDATA)>
```


Element   Attribute   Element Value   Attribute Value   Occurrence   Property   Entity

Figure 4 Sample Document