A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases

R. Bourret, C. Bornhövd, A. Buchmann
Department of Computer Science
Darmstadt University of Technology
Darmstadt, Germany, D-64283

{*rbourret, bornhoev, buchmann*}*@dvs1.informatik.tu-darmstadt.de*

Abstract

XML is rapidly gaining momentum in e-commerce and Internet-based information exchange, where its simplicity and custom-defined tags make it usable as a semantics-preserving data exchange format. However, to realize this potential, it is necessary to be able to extract structured data from XML documents and store it in a database, as well as to generate XML documents from data extracted from a database. Although many DBMS vendors are scrambling to extend their products to handle XML, there is a need for a lightweight, DBMS- and platform-independent load/extract utility as well. In this paper we describe such a utility, that solves the following problems: 1) loading data from XML documents into relational tables with a known schema, 2) creating XML documents according to a known DTD from data extracted from a database, 3) generating relational schemas from XML DTDs for on-the-fly storage of XML documents, and 4) generating XML DTDs from relational schemas for on-the-fly extraction of relational data. We introduce a language to describe a mapping between an existing XML DTD and an existing relational schema and discuss some of the interesting issues arising from such a mapping.

1. Introduction

The Internet has evolved into a global platform for e-commerce and information exchange. XML supports this by allowing the definition of semantically meaningful tags. Generic applications, such as search engines, can make limited decisions about the data in XML documents simply by comparing tag names and analyzing document structure. Applications that use a domain-specific language can recognize a particular set of predefined XML tags, and can perform more sophisticated tasks, such as automated billing, subject-specific searches, and summarizing and comparing information from multiple, independent providers of similar services.

Because reliability is crucial in e-commerce transactions and a significant amount of today's business data is stored in relational databases, an essential part of many applications that use XML as a data exchange format is transferring data between relational databases and XML documents. In this context, we identify four problems:

1. loading data from XML documents into relational tables with a known schema,
2. creating XML documents with a known DTD from data extracted from a database,
3. generating relational schemas from XML DTDs for on-the-fly loading of XML documents,
4. generating XML DTDs from relational schemas for on-the-fly extraction of relational data.

This paper describes an implemented utility, XML-DBMS, that solves these problems, exploiting and expanding on principles previously developed for mapping between object models and relational schemas. The utility is based on a view of the data in an XML document as a tree of objects, which allows us to use known object-relational mapping techniques modified to handle problems unique to XML. The utility is based on widely accepted standards – JDBC for database access, SAX and DOM for XML document access, and DDML for XML schema information[1] – so it can be used to build applications independent of

[1] JDBC is a java API for SQL-based access to relational databases. The API is implemented separately by each DBMS. SAX (Simple API for XML) provides a Java API for event-based parsing. Applications implement methods for events such as the start and end of an element, and register these methods with a parser. The parser calls the methods when the events occur. DOM (Document Object Model) specifies an object model for XML documents, with objects for elements, PCDATA, entity references, and so on. DDML (Document Definition Markup Language) is an XML schema language that was submitted to the W3C and is being incorporated in the W3C's XML Schema language.

DBMS, XML parser, and DOM implementations. It includes a simple language with which users can specify mappings between existing XML DTDs and relational schemas (cases 1 and 2 above), as well as components to dynamically generate relational schemas and mappings from XML DTDs and vice versa (cases 3 and 4).

Through its custom mapping language, it also accommodates the extreme flexibility with which data can be represented in an XML document. This flexibility distinguishes XML-DBMS from utilities that are being offered by DBMS vendors, which are mostly DBMS specific and make limiting assumptions about the XML documents and the required DBMS (for details see Section 8).

The main contribution of the work presented in this paper is the integration of previously developed concepts into a practical and flexible data transfer utility that realizes the mapping between two of the most important data handling technologies: XML documents and relational databases.

Section 2 presents a motivating example and application scenario, Section 3 briefly describes XML and its use as a data exchange format, Section 4 describes the object view, Section 5 describes the mapping and the mapping language, Section 6 describes how to generate XML DTDs from relational schemas and vice versa, Section 7 describes the software for transferring data, Section 8 discusses related work, and Section 9 provides conclusions.

2. Motivating Example and Scenario

WE-trade is an international B2B electronic trading center that is being funded by the European Commission to foster the trade between East and West Europe. One of the core components of WE-trade is the multilingual broker. The broker can match offers and requests both for products and for services that are formulated in the user's native language. Offers are submitted as XML documents that conform to a given DTD and the essential information is extracted and stored as structured data in a relational database. Offers and requests are matched with the help of underlying ontologies, that make it possible to match requests and offers not only based on direct concept matches but also based on sub- and super-categories. Concepts of the ontologies are translated and serve for the multilingual matching. The brokering service is intended to offer hints in addition to performing the match between offers and requests.

If a cherry grower from Bulgaria wants to sell 5 tons of cherries, the broker makes use of the product ontology to find a proper match, for example with an importer from Italy requesting fruit, but also alerts the seller with the help of the service ontology about the need to contact a customs agent familiar with European agricultural tariffs, a lawyer familiar with the appropriate laws, etc. The brokering service may even suggest candidates to contact based on WE-trade's database. The result of any matches are placed into XML documents in the appropriate language.

XML is particularly well suited for use in WE-trade because it allows us to structure documents with well-defined text portions in multiple languages into which we can place data that is identified through the appropriate tags. Since XML uses Unicode, all the necessary alphabets that must be supported in WE-trade can be handled.

A second major concern of WE-trade is the support of negotiations and the elaboration of the corresponding documents. XML is also well suited for this function. Structured documents can be expressed in XML and stored either as combinations of standard text elements and data or simply as BLOBs.

The delays that major DBMS vendors were experiencing in releasing their XML-enabled products motivated the development of a platform-independent load and extract utility that is based on standard structures and interfaces, such as DOM, JDBC and SAX. The need for a platform-independent, light-weight yet flexible utility becomes more evident as limiting assumptions about document structure and DBMS-specific implementations become apparent as XML-enabled DBMSs are released.

3. XML as a Common Data Exchange Format

XML [6] is a subset of SGML [14] defined for use on the Web. It defines a meta-language for defining XML markup languages as well as the rules that documents conforming to these languages must follow.

Although XML was originally conceived as a replacement for HTML [22], it has emerged as a generic data exchange format. Its hierarchical structure and user-defined tags can be adapted to a wide variety of structured and semi-structured data, and many operations on XML documents – parsing, editing, validation, transformation, and so on – can be performed independent of the actual tags in the document, which has led to a growing market of standard components from which XML applications can be assembled.

XML has several advantages over other data exchange formats. Its support of Unicode makes it highly portable and capable of representing most of the world's string data. Its use of tags to label data means that documents are self-describing, makes them readable by humans and allows them to be used by applications not initially intended as targets. Its predictable format means that any application can read a document, even if it doesn't understand the semantics of the tags.

XML also has disadvantages, the most obvious being size and lack of data types. Although the repetitive nature of tags lends itself to compression, documents with large amounts of numeric or binary data represented as strings will always be larger than if they used a binary format. The lack of data types is expected to be resolved in the future by the introduction of data type attributes, but conversion

to and from strings will always be slower than using binary formats.

XML does not enforce unique tags. Although this problem will never completely be solved, many industry groups are standardizing tags and the W3C is working on technologies to simplify the reuse of standard tags, for example, by encoding UN/EDIFACT and STEP standards in XML.

```
<Orders>
  <SalesOrder SONumber="12345">
    <Customer CustNumber="543">
      <CustName>Blue River Fruits</CustName>
      <Address>
        <Street>24 Templerd.</Street>
        <City>Dublin</City>
        <Country>Ireland</Country>
        <PostCode>0802871</PostCode>
      </Address>
    </Customer>
    <OrderDate>150999</OrderDate>
    <Line LineNumber="1">
      <Product Name="Rainier Cherries">
        <Description>
          <P><B>Rainier cherries:</B><BR />
          Large, golden, and honey-sweet,
          best quality.</P>
        </Description>
        <Price Currency="EUR">310</Price>
      </Product>
      <Quantity Unit="ton">2</Quantity>
    </Line>
    <Line LineNumber="2">
      <Product Name="Gala Apples">
        <Description>
          <P><B>Gala apples:</B><BR />
          Good-quality, firm with smooth and clean skin.</P>
        </Description>
        <Price Currency="EUR">170</Price>
      </Product>
      <Quantity Unit="ton">5</Quantity>
    </Line>
  </SalesOrder>
</Orders>
```

Figure 1. Sales order example.

By defining domain-specific tags, special-purpose markup languages can be defined. These fall into two rough categories. *Data-centric languages* describe discrete pieces of data and are typically used to transfer data between applications and data stores. They are characterized by fairly regular structure, fine-grained data (the smallest independent unit of data is usually at the attribute or PCDATA-only element level), and little or no mixed content[2]. The order in which sibling elements and PCDATA occurs is usually not significant. For example, a sales order written in a "sales order language" might look like the example in Figure 1.

Document-centric languages are used to create documents for human consumption, such as contracts, books, and advertisements. They are characterized by less predictable structures, coarser-grained data, and large amounts of mixed content. The order in which sibling elements and PCDATA occurs is usually significant. For example, a lease written in a simple "contract language" might look like the following:

```
<LeaseContract>
  <Lessee>ABC Industries</Lessee> agrees to lease the property
  at <Address>123 Main St., Dublin, Ireland</Address> from
  <Lessor>XYZ Properties</Lessor> for a term of not less than
  <LeaseTerm TimeUnit="Months">18</LeaseTerm> at a cost of
  <Price Currency="EUR">1000</Price>.
</LeaseContract>
```

Figure 2. Contract example.

XML-DBMS is capable of handling the full spectrum of XML markup languages. Because it stores the necessary linkage and order information it is able to store and reconstruct XML documents written in document-centric as well as data-centric languages.

4. An Object View of an XML Document

To simplify the mapping process, we view the *content* of a document as a tree of objects to which existing object-relational mapping techniques can be applied. This view is not the DOM (Document Object Model), which models a documents's *structure* with standard tags (e.g., PCDATA, entity references, etc.) as opposed to the *content* of the document that is represented in our object tree. For example, the XML document shown in Figure 1 might be modeled with sales order, line, customer, and part objects. The remainder of this section describes how the parts of an XML document can be viewed.

4.1. Element Types

Element types can be viewed either as classes or properties. Element types with element or mixed content, such as the *SalesOrder* element type or the *LeaseContract* element type are generally viewed as classes. Element types with PCDATA-only content, such as the *OrderDate* and *PostCode* element types are generally viewed as properties.

The view of element types as properties is not limited to element types with PCDATA-only content. To see why this is useful for element types with mixed or element content, consider the *Description* element type, which contains an HTML description of a product. Although such an element can contain subelements such as <B> and <P>, its content is useful only as a whole. That is, its subelements and PCDATA cannot be meaningfully interpreted on their own and it makes sense to view it as a property.

When an element type is viewed as a property, it belongs to the class of its parent, i.e., containing element type. Because element types can have more than one parent type, an element type-as-property can be viewed as a property of more than one class. The value of an element whose type is viewed as a property is its content; for storage in the database, this is serialized as XML, using

[2]Technically, *mixed content* means any content that contains PCDATA, including PCDATA-only content. In common usage, as in this paper, *mixed content* refers to a mixture of elements and text.

markup for subelements and their attributes. For example, the content of the second *Description* element in Figure 1 is:

```
<P><B>Gala apples:</B><BR />
Good-quality, firm with smooth and clean skin.</P>
```

4.2. PCDATA and Attributes

When an element type is viewed as a class, its PCDATA and attributes are viewed as properties of that class. When an element type is viewed as a property, its PCDATA is viewed as part of its value and its attributes are viewed as properties of the element type's parent class. The reason for this latter view is that the element type, being viewed as a property, cannot itself have properties.

4.3. Hierarchy

Elements in an XML document are organized in a hierarchy. How the parent/child relationship between two elements in an XML document is viewed depends on how their respective element types are viewed. If both element types are viewed as classes, then the relationship is viewed as an inter-class relationship. If the child element type is viewed as a property, then the relationship is viewed as a class-property relationship. If the parent element type is viewed as a property, then the child element is part of the value of that property.

4.4. Order

The order in which elements and PCDATA appear in an XML document may be significant, as can be seen in our contract example. If this case, inter-class relationships (between two element types-as-classes) are considered to be ordered, as are the properties corresponding to element types-as-properties and PCDATA. That is, each inter-class relationship and each of these properties is considered to have a parallel property that tracks its order in its parent. For a given parent, all of these order properties share a single order space.

The order in which values in a multi-valued (IDREFS, NMTOKENS, or ENTITIES) attribute occur may also be significant. In this case, the properties of such attributes are ordered. Each attribute has its own order space.

5. Mapping Data Between XML and Relational Databases

We use an object-relational mapping to map an object view to a relational schema: we map classes to tables, properties to columns, and inter-class relationships to candidate key/foreign key pairs. However, to provide more flexibility, we add some features not found in most object-relational mappings.

First, single-valued properties can be mapped to a column in the class table or a column in a separate table; the latter is useful for storing BLOBs separately. Multi-valued properties must currently be mapped to multiple rows in a separate table. We are considering one additional mapping to a fixed set of columns in the class table (when the number of property values has a known maximum).

Second, we allow the mapping to specify whether object IDs (keys) and order information are generated by the system or use existing data. For example, we might designate that the *SONumber* attribute in our sales order language contains the key for the sales order table, but state that we want the system to generate a unique ID for each contract in our contract language. Similarly, we might use the *LineNumber* attribute to specify the order of *Line* elements in their parent, but have the system generate order information about the PCDATA and subelements of the *LeaseContract* element.

Third, we provide two different ways to ignore structure that exists in an XML document but not in the database. The first is to ignore the root element. This is useful when the root element type exists only to satisfy XML's requirement that there be a single root element, such as the *Orders* element in our sales order language, which exists only so we can place more than one sales order in a single document. The second is to "pass through" an element type-as-class, treating its properties as properties of its parent class. For example, the *Address* element type in our sales order language exists only for clarity. We can treat its properties (*Street*, *City*, *Country*, and *PostCode*) as properties of its parent class (*Customer*). When an element type is ignored or passed-through, its structure is removed when transferring data to the database and recreated when extracting data from the database.

Finally, we do not require that any given class or property (or table or column) be mapped at all. Anything that is not mapped is simply not transferred; in the case of an element whose type is not mapped, its children are also not transferred because there is no way to link them to higher level elements.

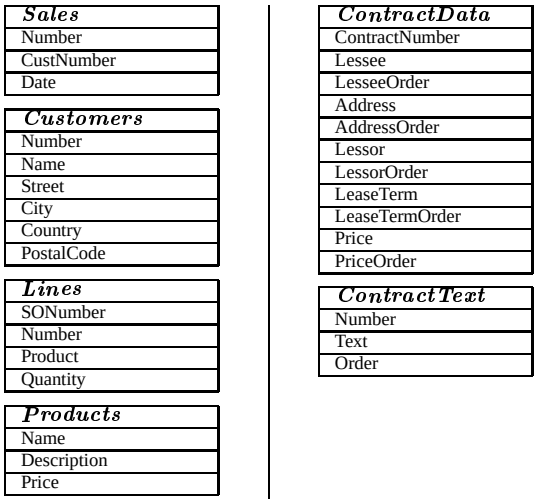
We enforce the usual inferential integrity notion. Ordering information is stored on the "many-side" of a 1:N relationship.

The mapping between an object view and a relational schema can be specified by the user or generated dynamically by the utility. In the former case, both the XML DTD and the relational schema must exist and the user specifies the object view, as well as the mapping, with the mapping language we have developed. The latter case applies when either the XML DTD or the relational schema, but not both, exists. In this case, the utility generates the missing schema (relational or DTD), the object view, and the mapping according to predefined rules.

5.1. The XML-DBMS Mapping Language

Our mapping language is a simple, XML-based language that describes both how to construct an object view for an XML document and how to map this view to a relational

schema. This requires both the XML DTD and the relational schema to exist. We introduce the main parts of the language in a series of examples, which use the sample XML documents shown in Section 3 and the following tables; the full DTD for the language is given in [2].



Mapping Classes (Element Types) to Tables. Element types with element content are usually viewed as classes and mapped to a table. For example, the following declares the *SalesOrder* element type to be a class and maps it to the *Sales* table:

```
<ClassMap>
  <ElementType Name="SalesOrder"/>
  <ToClassTable>
    <Table Name="Sales"/>
  </ToClassTable>
  . . . property maps . . .
  . . . related class maps . . .
  . . . pass-through maps . . .
</ClassMap>
```

The *ClassMap* element contains all the information needed to map a single class (element type), including the table to which the class is mapped, the maps for each property in the class, a list of related classes, and a list of passed-through child classes.

The *ElementType* element identifies the element type (class) being mapped and the *ToClassTable* element gives the name of the table to which the class is mapped. The element type name may include a namespace prefix [5], which is declared in a *Namespace* element; prefixes declared in *Namespace* elements are separate from those declared with xmlns attributes.

Mapping Properties (Attributes and Element Types) to Columns. Single-valued attributes and element types with PCDATA-only content are usually viewed as properties and mapped to columns. For example, the following declares the *SONumber* attribute and the *OrderDate* element type (when *SalesOrder* is its parent) to be properties and maps them to the *Number* and *Date* columns, respectively. These maps are nested inside the class map for *SalesOrder*.

```
<PropertyMap>
  <Attribute Name="SONumber"/>
  <ToColumn>
    <Column Name="Number"/>
  </ToColumn>
</PropertyMap>

<PropertyMap>
  <ElementType Name="OrderDate"/>
  <ToColumn>
    <Column Name="Date"/>
  </ToColumn>
</PropertyMap>
```

The *Attribute* and *ElementType* elements identify the properties being mapped and the *ToColumn* elements state that they are being mapped to columns in the table to which the class (*SalesOrder*) is mapped.

Mapping Inter-Class Relationships. When a child element type is viewed as a class, its relationship with its parent element type must be stated in the map of the parent class. For example, the following declares that *Line* is related to the *SalesOrder* class. This map is nested inside the class map for *SalesOrder*; the actual mapping of the *Line* class is separate.

```
<RelatedClass KeyInParentTable="Candidate">
  <ElementType Name="Line"/>
  <CandidateKey Generate="No">
    <Column Name="Number"/>
  <CandidateKey/>
  <ForeignKey>
    <Column Name="SONumber"/>
  </ForeignKey>
  <OrderColumn Name="Number" Generate="No"/>
</RelatedClass>
```

The *ElementType* element gives the name of the related class and the *KeyInParentTable* attribute states that the candidate key used to join the tables is in the parent (*Sales*) table. *CandidateKey* and *ForeignKey* give the columns in these keys, which must match in number and type. The *Generate* attribute of *CandidateKey* tells the system whether to generate the key. This allows us to preserve keys that have business meaning and generate object identifiers when no such keys exist. In this case, we do not generate the key because we have mapped the *SONumber* attribute of the *SalesOrder* element type to the candidate key column (*Sales.Number*).

The (optional) *OrderColumn* element gives the name of the column that contains information about the order in which *Line* elements appear in the *SalesOrder* element. Because this column must appear in the table on the "many" side of the relationship, *Number* refers to the *Lines.Number* column, not the *Sales.Number* column. The *Generate* attribute of the *OrderColumn* element tells the system whether to generate the order value. In this case, we do not generate the order value because we will separately map the *LineNumber* attribute of the *Line* element type to the order column (*Lines.Number*).

Eliminating Unwanted Hierarchy. When element types represent hierarchy that exists in the XML document

but not in the database, they can be eliminated in one of two ways. First, root element types can be ignored. For example, the following states that the *Orders* element type is to be ignored.

```
<IgnoreRoot>
  <ElementType Name="Orders"/>
  <PseudoRoot>
    <ElementType Name="SalesOrder"/>
    <CandidateKey Generate="No">
      <Column Name="Number"/>
    </CandidateKey>
  </PseudoRoot>
</IgnoreRoot>
```

The first *ElementType* element gives the element type to be ignored. The *PseudoRoot* element introduces an element type (*SalesOrder*) to serve as a root in its place; there can be multiple pseudo-roots. The (optional) *CandidateKey* element gives the key to be used when retrieving data from the database.

The second way to eliminate unwanted hierarchy is to "pass through" non-root element types, e.g., treat their properties as if they were properties of their parent. For example, the following declares that the *Address* element type is to be passed through. Like element types-as-properties, passed-through element types are mapped on a per-parent basis, so this map is nested inside the class map for *Customer*.

```
<PassThrough>
  <ElementType Name="Address"/>
  <PropertyMap>
    <ElementType Name="Street"/>
    <ToColumn>
      <Column Name="Street"/>
    </ToColumn>
  </PropertyMap>
  . . . other property maps . . .
  . . . related class maps . . .
  . . . pass-through maps . . .
</PassThrough>
```

The first *ElementType* element gives the name of the passed through element type. Because passed-through element types are viewed as classes, the *PassThrough* element must map the properties, related classes, and passed-through child classes of the passed-through class, just like a class map does. Shown above is a property map that maps the *Street* element type to the *Street* column of the parent table (*Customers*).

So that passed-through elements can be recreated when retrieving data from the database, only one pass-through element of a given type can occur directly beneath a given parent.

Mapping Mixed Content. Mixed content consists of both PCDATA and elements. The order in which the PCDATA and elements appear is usually important, so we need to keep order information for the PCDATA as well as each element. For example, the following maps the *Lessee* element type to the *Lessee* column and stores system-generated order information in the *LesseeOrder* column; this map is nested inside the class map for the *LeaseContract* element type.

```
<PropertyMap>
  <ElementType Name="Lessee"/>
  <ToColumn>
    <Column Name="Lessee"/>
  </ToColumn>
  <OrderColumn Name="LesseeOrder" Generate="Yes"/>
</ElementTypeMap>
```

Because PCDATA can occur multiple times in mixed content, it is usually mapped to a separate table. For example, the following maps the PCDATA to the *ContractText* table; this map is nested inside the class map for the *LeaseContract* element type as well.

```
<PropertyMap>
  <PCDATA/>
  <ToPropertyTable KeyInParentTable="Candidate">
    <Table Name="ContractText">
    <CandidateKey Generate="Yes">
      <Column Name="ContractNumber"/>
    </CandidateKey>
    <ForeignKey>
      <Column Name="Number"/>
    </ForeignKey>
    <Column Name="Text"/>
    <OrderColumn Name="Order" Generate="Yes"/>
  </ToPropertyTable>
</PropertyMap>
```

The *ToPropertyTable* element states that the table contains only property values, not a class. In addition to giving the candidate and foreign keys needed to retrieve PCDATA values from the table, we give the names of the columns (*Text* and *Order*) in which the values and the order information are stored. Notice that we ask the system to generate both the candidate key (*ContractNumber*) and the order information; this is because the lease does not contain this information, which is a common situation in document-centric documents.

6. Dynamic Generation of XML DTDs and Relational Schemas

The mapping process described in Section 5 requires that the structure of both the XML document and the relational schema be known. If either is not the case, as might occur when storing an XML document on the fly or generating an XML document from the output of an ad-hoc query, the XML-DBMS tool first generates the corresponding DTD or relational schema and then applies the mapping process described above.

The mapping process between XML DTDs and relational schemas is not lossless. That is, generating a relational schema from an XML DTD followed by generating an XML DTD from that relational schema yields a valid DTD but not necessarily the initial DTD. In addition to naming problems, we lose data types when creating DTDs from relational schema and we add key and order columns when creating relational schema from DTDs.

6.1. Generating an XML DTD from a Relational Schema

To generate a DTD from a relational schema, we start from a root table and recursively follow primary key/foreign key

relationships to discover all related tables, noting circular references and building the schema for the corresponding XML structures as we go. To generate a DTD from a result set, we treat the result set as the root table; obviously, there are no primary key/foreign key relationships to follow in this case.

For each table in the hierarchy, we create an element type with element content. It contains a sequence of PCDATA-only elements (one for each column[3]), followed by a sequence of elements with element content (one for each related table). Whether the PCDATA-only elements are optional ('?' operator) depends on the nullability of the column. Related table elements are optional ('?' operator) if they represent a table with an imported foreign key. They can occur zero or more times ('*' operator) if they represent a table to which a foreign key is exported. Note that this mapping saves key values as data but does not impose any order.

6.2. Generating a Relational Schema from an XML DTD

To generate a relational schema from a DTD, we read the DTD, noting circular references and build the database schema as we go. We consider an element type with mixed or element content to be a class and create a class table for it. We consider all attributes and element types with PCDATA-only content to be properties. For single-valued properties, we create columns in the class table. For multi-valued properties, we create separate property tables.

We determine nullability of columns in class tables by checking if an attribute or singly-occurring PCDATA-only element type is optional; in the latter case, this is determined on a per-parent basis. Besides the obvious use of the '?' and '*' operators, PCDATA-only element types that appear directly or indirectly in a choice are also optional. Columns in property tables are never nullable, as the absence of a value simply results in the absence of a row.

For each element type-as-class, we create a primary key column and use this to create a primary key/foreign key relationship with each of its child element types that is mapped as a class. The relationship between parent and child is always one-to-many and the mapping specifies that the key values are always generated.

The user can choose whether to store order information. If so, we create one order column for each property and parent/child relationship; the mapping specifies that order values are always generated.

Default data types are assigned for all columns depending on a field's content but may be overwritten by the user.

[3]This applies to all columns except those with a data type of BINARY, VARBINARY, or LONGVARBINARY, which are not currently mapped.

7. XML-DBMS: A Data Transfer Utility

XML-DBMS is a utility for transferring data between XML documents and relational databases, implemented as a set of Java packages: a data transfer package, a map factory package, and a coordinator package. Figure 3 shows the architecture of XML-DBMS.

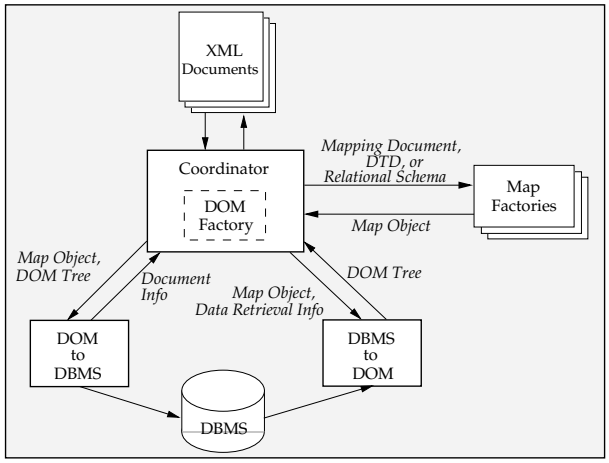


Figure 3. XML-DBMS architecture.

When the coordinator receives a request to transfer data, it first creates a map object that describes the mapping between the XML DTD and the relational schema. It then uses the DOMToDBMS and DBMSToDOM components to transfer data to or from the database. When transferring data to the database, the coordinator must construct a DOM tree from the XML document and pass this, along with the map object, to the DOMToDBMS component; in return, it receives a document information object, which contains information about how to retrieve the data at a later point in time. When transferring data from the database, the coordinator passes the map object, along with document retrieval information (a document information object, a list of tables and key values, or a result set), to the DBMSToDOM component; in return, it receives a DOM tree.

The coordinator can store mapping documents and DTDs (in the form of DDML documents) for each class of documents. This is possible because mapping documents and DDML documents are XML documents, and can therefore be stored using the DOMToDBMS and DBMSToDOM components. The coordinator can also store information to retrieve the document at a later time, keyed by a user-defined name such as a file name, as well as the original form of the document in a BLOB column. The latter is useful for documents defined according to a document-centric language, in which the original physical structure (entity references, CDATA sections, and so on) is often important.

7.1. Map Objects and Map Factories

A map object controls how data is transferred. Map objects contain both XML-centric and database-centric views of the

mapping between a particular DTD and a particular relational schema as well as general metadata such as column data types. The map object can also create the INSERT, SELECT, and CREATE TABLE statements needed to process data according to a particular mapping.

Map objects are created by a map factory. The map factory package has factories to create maps from mapping documents and DTDs, with factories planned for relational schemas and result sets. The first of these is a SAX application that reads a mapping document. The second is a SAX application that recognizes DDML events. It can be used either with DDML documents or DTDs (in the latter case using a special parser that translates DTDs into DDML) and can easily be adapted for use with other XML schema languages. The third and fourth rely on the metadata facilities of JDBC, some of which (information about keys) are not supported by all JDBC drivers. All of the factories produce a one directional view of the mapping – either from XML to the database or vice versa – and then use the general facilities of the map factory package to invert the view.

7.2. Transferring XML Data to a Relational Database

The DOMToDBMS component recursively walks the DOM tree in modified depth-first, width-second order. The modified order is needed to meet referential integrity constraints in the database. For example, consider the DOM tree for the first XML document in Section 3, part of which is shown in Figure 4.

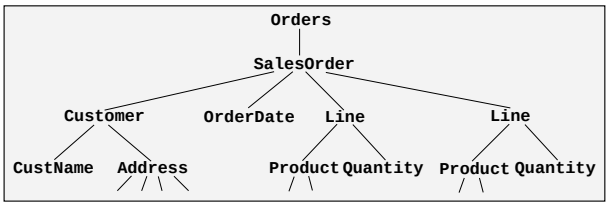


Figure 4. DOM tree of sales order example.

Because the primary key in the *SalesOrder*/*Line* relationship is in the sales order table, the row in the sales order table must be added before the rows in the lines table can be added.

Thus, DOMToDBMS walks the tree as follows. When it encounters a node mapped to a table, such as the *SalesOrder* node, it creates a buffer for a row in that table. It then processes the children of that node. If a child is mapped to a column, such as the *OrderDate* node, the node's value is stored in the buffer. If a child is mapped to a table that stores a foreign key in the current table, such as the *Customer* node, the node is immediately processed and the foreign key stored in the current row buffer. If a child is mapped to a table that uses the primary key of the current table, such as the *Line* nodes, the node is saved for later processing. Note that children of nodes mapped as pass-through, such as the *Address* node, are processed immediately, as if they were siblings of the other children.

After processing all the children of a table node, DOMToDBMS inserts the current row – whether to commit the insert immediately or after the entire document is inserted is left as an option for the user – and then processes the saved children, passing them the primary key from the current row.

Although this method may require multiple row buffers to be open at any one time, it minimizes the number of times the DOM tree is traversed. It also minimizes the number of statements open on the database, since a statement is needed only when a row is inserted. We are currently investigating the possibility of rewriting this component as a SAX application, which would greatly increase its speed, as well as eliminating most memory problems encountered when processing large XML documents.

If the mapping specifies that DOMToDBMS generates the key (object ID) for a class, DOMToDBMS calls a method in a user-overridable class.

7.3. Transferring Relational Data to an XML Document

The DBMSToDOM component traverses the relations and builds a DOM tree. For each table, it opens a result set over the table and reads a row. If the table is a class table, it creates an element node for the row. It then creates nodes (subelements, attributes, and PCDATA) for each property value stored in the row. Following the key/foreign key relationships it processes each referenced table.

While constructing the tree, DBMSToDOM orders nodes according to their related order information, if any. It also checks whether any passed-through or ignored elements need to be recreated, and creates child nodes for any markup in the values of element types-as-properties.

By building a DOM tree, DBMSToDOM does not have to process child elements in any particular order. Instead, it can simply insert each child in its correct position as it is processed. Thus, it can process child tables one at a time and the number of result sets open at any point in time is equal to the depth of the hierarchy.

7.4. Data Transfer Issues

Because of mismatches in the capabilities of XML and relational DBMSs a number of data transfer issues arise which, due to space limitations, are only outlined here. For details see [2].

Except for unparsed entities, all data in an XML document is text, i.e., XML does not support *data types*. We use the conversion facilities of Java to convert relational data to and from text.

XML supports the concept of *null data* through optional element types and attributes. We give users the choice of whether empty strings in elements and attributes should be treated as empty strings or null data.

There are two common ways to store *binary data* in XML: as an unparsed entity or through Base64 encoding [10]. We use the former. When transferring data to the database, we open a byte stream over the unparsed entity and copy the data to a BLOB column. We cannot currently transfer BLOB data to an XML document, as DOM does not support creation of new entities.

One unsolved problem is handling *non-ASCII characters* when transferring XML data to the database. XML documents accept all of Unicode except the control characters. Most databases do not support Unicode and require special configuration to handle non-ASCII encodings of character data.

8. Related Work

Object-relational mappings are well described in the literature, e.g., [27, 17]. Our mapping supports a subset of this work, not preserving such things as super-/subtype relationships, which are not supported by XML, or many-to-many relationships, which are not widely used in XML. Unlike traditional object-relational mappings, we introduce the notion of order shared by multiple properties of the same class.

Although developed separately, our work is a superset and practical implementation of the Basic, Shared, and Hybrid Inlining techniques described in [24] for mapping a DTD to relational schema. Our pass-through technique is equivalent to their inlining technique and our designation of root tables is equivalent to their isRoot fields. Furthermore, we allow root element types to be ignored and children to be passed through regardless of whether they have descendants mapped to separate relations. Our mapping language allows us to create maps exactly representing the techniques in [24].

The literature also contains many papers describing how to generate object schemas from relational schemas and vice versa, e.g., [27, 8, 29, 12, 1]. These cover a variety of techniques. Some of these, such as recognizing relations that represent set attributes [17], recognizing multiple objects in a single relation [15], or recognizing non-normal relations [27], are related to our work.

Several middleware products are available for transferring data between relational databases and XML documents. These fall into two broad categories: template-driven and model-driven.

In template-driven products, such as ODBC2XML [13], XSQL Servlet [20], XML Servlet [7], and XOSL [28], commands are embedded in a template that is processed by the middleware. For example, a SELECT statement might be replaced by its results, formatted as XML. Template-driven middleware is extremely flexible and often includes programming structures such as loops and if statements. Its two major disadvantages are the amount of code a user must write (in the form of templates) to generate complex documents and the fact that none of the available products can transfer data from XML documents to the database.

Model-driven products, such as XML-DBMS, ASP2XML [25], DB2XML [26], and XML SQL Utility [21], define a data model for the XML document and then explicitly or implicitly map this to the database. ASP2XML and DB2XML use a table model, meaning that nesting can only be three (ASP2XML) or four (DB2XML) layers deep, where the elements at each layer match the database (DB2XML only), tables, rows, and columns. Any document that does not match this model cannot be processed.

Like XML-DBMS, Oracle's XML SQL Utility models the document as a tree of objects. However, XML SQL is far less versatile for a number of reasons. First, the mapping is hard-coded in the utility. This leads to problems such as the inability to process data stored in attributes and the requirement that element type names must match column names. Second, it does not store information about the order in which child elements and PCDATA occur, making it impossible to reconstruct many XML documents. Third, it relies on SQL 3 object views to process documents that model anything more complex than a single table, so it is not portable to databases that do not support these.

In addition to middleware, a number of databases have integrated support for XML. The Internet File System in Oracle 8i [19] appears to be similar to our work, using "document descriptors" to map relational schema to XML documents and vice versa. One interesting aspect of document descriptors is that they can map document fragments to a single column rather than mapping each element individually. It does not appear that the Internet File System can generate document descriptors at run time, as our system does. The Internet File System is currently in beta release.

Excelon [18] and Tamino [23] store XML data in a local database: Excelon uses ObjectStore and Tamino uses the "X-Machine", a hierarchical database. Both have the capability to integrate data from relational databases and both have a mapping mechanism to do this. Although the use of a local database means that they can perform heterogeneous joins, it also means they are heavyweight solutions to the problem of simply transferring data between XML documents and relational databases.

Extensions for processing XML in Sybase Enterprise Application Server, Informix Dynamic Server, Microsoft SQL Server, and IBM DB2 have been announced, but insufficient technical details are available to compare them with our work.

Finally, a number of object-oriented and object-relational databases can be used to store XML documents. These include the Content Management Suite from POET Software, Frontier 5 from UserLand, and Texcel Information Manager from Texcel. Because they are concerned

with storing documents and document structure, as opposed to the data in those documents, they are not directly related to our work.

9. Conclusion

The main contribution of this paper is to describe a lightweight yet very flexible, reusable utility for transferring data between XML documents and relational databases. By viewing an XML document as a tree of objects, we are able to exploit and expand upon previous work in the fields of object-relational mappings and schema generation to build a utility that not only transfers data between documents and databases of known schemas, but also generates both XML DTDs and relational schemas at runtime for on-the-fly loading and extraction of data.

As part of this utility, we have developed a flexible, XML-based language for specifying the mapping between XML documents and relational schemas. This language expands on the basic principles of object-relational mappings to handle situations unique to XML.

Our utility is written in Java and is based on widely accepted standards such as JDBC, SAX, DOM, and DDML. It is therefore independent of platform, DBMS, and XML parser and DOM implementation and is suitable for deployment in a wide number of environments. We have implemented version 1.0 of the utility, including the data transfer components, the map objects, and the map factories for the mapping language and DTDs. This utility is currently being used or evaluated for use in a number of German corporations and government agencies, as well as WE-trade, a B2B e-commerce project of the European Union, and is also a part of the ExOffice Java/XML Enterprise Platform.

References

[1] S. Amer-Yahia, S. Cluet, C. Delobel: *Bulk Loading Techniques for Object Databases and an Application to Relational Data*, VLDB, New York City, USA, 1998

[2] R. Bourret, C. Bornhövd, A. Buchmann: *A Generic Load/Extract Utility for Data Transfer between XML Documents and Relational Databases*, TR-DVS99-1, DVS, Dep. CS, Darmstadt U. of Technology, Germany, Dec. 1999

[3] R. Bourret, J. Cowan, I. Macherius, S. St. Laurent: *Document Definition Markup Language (DDML) Specification*, Version 1.0, www.w3.org/TR/NOTE-ddml, 1999

[4] P. Biron, A. Malhotra: *XML Schema Part Two: Datatypes*, www.w3.org/TR/xmlschema-2/, 1999

[5] T. Bray, D. Hollander, A. Layman: *Namespaces in XML*, www.w3.org/TR/REC-xml-names, 1999

[6] T. Bray, J. Paoli, C. Sperberg-McQueen: *Extensible Markup Language (XML) 1.0*, www.w3.org/TR/REC-xml, Feb. 10, 1998

[7] Cerium Component Software: *XML Servlet*, ceriumworks-.com/tech.html, 1998

[8] M. Castellanos, F. Saltor: *Semantic Enrichment of Database Schemas: An Object Oriented Approach*, RIDE-IMS, Kyoto, Japan, 1991

[9] L. Wood, et al.: *Document Object Model (DOM) Level 1 Specification, Version 1.0*, www.w3.org/TR/REC-DOM-Level-1/, 1998

[10] N. Freed, N. Borenstein: *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, www.cis.ohio-state.edu/htbin/rfc/rfc2045.html, 1996

[11] M. Fuchs, M. Maloney, A. Milowski: *Schema for Object-oriented XML*, www.w3.org/TR/NOTE-SOX, 1998

[12] C. Fahrner, G. Vossen: *Transforming Relational Database Schemas into Object-Oriented Schemas According to ODMG-93*, DOOD, Singapore, 1995

[13] Intelligent Systems Research: *ODBC2XML: ODBC XML Generator*, members.xoom.com/gvaughan/odbc2xml.htm, 1999

[14] ISO 8879. *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986

[15] P. Johannesson: *A Method for Transforming Relational Schemas into Conceptual Schemas*, ICDE, Houston, Texas, 1994

[16] D. Megginson: *Simple API for XML (SAX)*, www.megginson/-.com/SAX/, 1998

[17] W. Meng, A. Kamada, Y-H. Chang: *Transformation of Relational Schemas to Object-Oriented Schemas*, COMPSAC, Dallas, Texas, 1995

[18] Object Design, Inc.: *An XML Data Server For Building Enterprise Web Applications*, www.odi.com/-excelon/XMLResource/build_ent_web_apps.pdf, 1999

[19] Oracle Corporation: *XML Support in Oracle8i and Beyond*, www.oracle.com/xml/documents/xml_twp/, 1998

[20] Oracle Corporation: *Oracle XSQL Servlet*, technet/-.oracle.com/tech/xml/xsql_servlet/main.htm, 1999

[21] Oracle Corporation: *Oracle XML SQL Utility for Java*, technet.oracle.com/tech/xml/oracle_xsu/main.htm, 1999

[22] D. Ragget; A. Le Hors; I. Jacobs: *HTML 4.0 Specification*, www.w3.org/hypertext/WWW/MarkUp/MarkUp.html, 1997

[23] Software AG: Tamino: *The Information Server for Electronic Business*, www.softwareag.com/tamino/, 1999

[24] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton: *Relational Databases for Querying XML Documents: Limitations and Opportunities*, VLDB, Edinburg, Scotland, 1999

[25] Stonebroom Software: *ASP2XML: XML Interface Active Server Component*, www.stonebroom.com/asp2xml.htm, 1999

[26] V. Turau: *DB2XML: A Tool for Transforming Relational Databases into XML Documents*, www.informatik.fh-wiesbaden.de/~turau/DB2XML/index.html, 1999

[27] T. Teorey, D. Yang, J. Fry: *A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model*, ACM Comp. Surv., 18(2), 1986

[28] R. Westphal: *XOSL – The XML-Based Stylesheet Language for Converting RDBMS Legacy Data to XML*, www.riposte.com/xosl/index.html, 1998

[29] L-L. Yan, T-W. Ling: *Translating Relational Schema With Constraints Into OODB Schema*, Interoperable Database Systems, Elsevier Science Publishers B.V., 1993