Management of XML Documents without Schema in Relational Database Systems

Thomas Kudrass
Leipzig University of Applied Sciences,
Department of Computer Science and Mathematics, D-04251 Leipzig, Germany
kudrass@imn.htwk-leipzig.de

Many applications deal with highly flexible XML documents from different sources, which makes it difficult to define their structure by a fixed schema or a DTD. Therefore, it is necessary to explore ways how to cope with such XML documents. The paper analyzes different storage and retrieval methods for schemaless XML documents using the capabilities of relational systems. We compare our experiences gathered at the implementation of an XML-to-relation mapping with a LOB implementation in a commercial RDBMS. The paper concludes with a vision how different storage methods could converge towards a common high-level XML-API for databases.

1. Motivation

Modern information systems are dominated by heterogeneous IT landscapes and decentralized structures. For that reason new solutions are developed to integrate, search and view electronic documents and information from the connected systems. The information has to be presented in a uniform way without bothering the end-user to think about the original content format such as files or tables. This characterizes roughly the functionality of a middleware layer, which requires a common model to represent the information. For it, the XML standard is well-suited for both data-centric and document-centric documents. In our experience, many commercial applications have to handle evolving schemas and structures of the local data during the development and the operation of integrated information systems. Regarding the high degree of schema evolution we pursued the idea to leave the schema out in order to reduce the overhead of a schema-driven approach to store XML documents in a database. Thus, we had to explore ways how to store and retrieve XML data from local sources without using a DTD or an XML schema. Most tools provided at the market support the conversion of XML documents into classes or tables of the underlying database system using the schema information. On the other hand, there was a requirement to use relational database systems for the management of the XML documents to avoid any additional investments in new platforms. Another basic assumption was to store the whole XML document in the database and not only the contents, unlike template-driven mapping approaches [Bou00, XDK]. This ensures the database support of all possible complex structures of XML documents beyond simple row sets
The test implementation we describe here has been built on top of the RDBMS Oracle 8i with its text database extension interMedia Text [Mun00].

The rest of this paper is organized as follows. Section 2 describes experiences with the structure-oriented approach that maps XML documents to generic relations. Section 3 does the same with the opaque approach representing XML documents as large objects. Storage and query issues of both approaches are discussed accordingly. Section 4 outlines the perspective for a unified database interface for XML documents. The conclusion in section 5 summarizes current insights and gives a short outlook.

2. Mapping XML Documents to Relations - Structure-Oriented Approach

A couple of algorithms that describe the mapping of XML documents into relations has already been published, for example [FK99, STH+99]. These algorithms comprise different methods how to map attributes and values in XML documents into the relational model. By combining different mapping rules for the elements of XML (e.g., attribute mapping) a variety of algorithms has been developed. All of them belong to the structure-oriented approach to represent XML in databases. That means the transformation of the XML documents depends only on their structure without using any semantics or access to a document schema. The basic idea of these algorithms is to consider XML documents as directed, ordered and labelled graphs. The nodes of the graph represent XML elements. The order of subelements and attributes is represented by the order of the edges outgoing from a node. Attributes, subelements and references are all represented as edges, which leads to an information loss when mapping the XML document into a graph.

2.1 Storage

Our relational storage model uses a refined structure-oriented algorithm with a graph structure that is based on the XML-QL model. The structure of the graph is represented by an edge table enriched by some information in order to distinguish between different target nodes that may be elements, attributes or content. Since a leaf is considered as a node without outgoing edges, nodes are managed in the edge table as well. The content is stored in a leaf table; attribute values are separated in an attribute table.
The relational schema is defined as follows:

```
Edge (sourceid, targetid, leafid, attrid, docid, type, name, depth)
Leaf (id, value)
Attr (id, value)
Doc  (id, value)
```

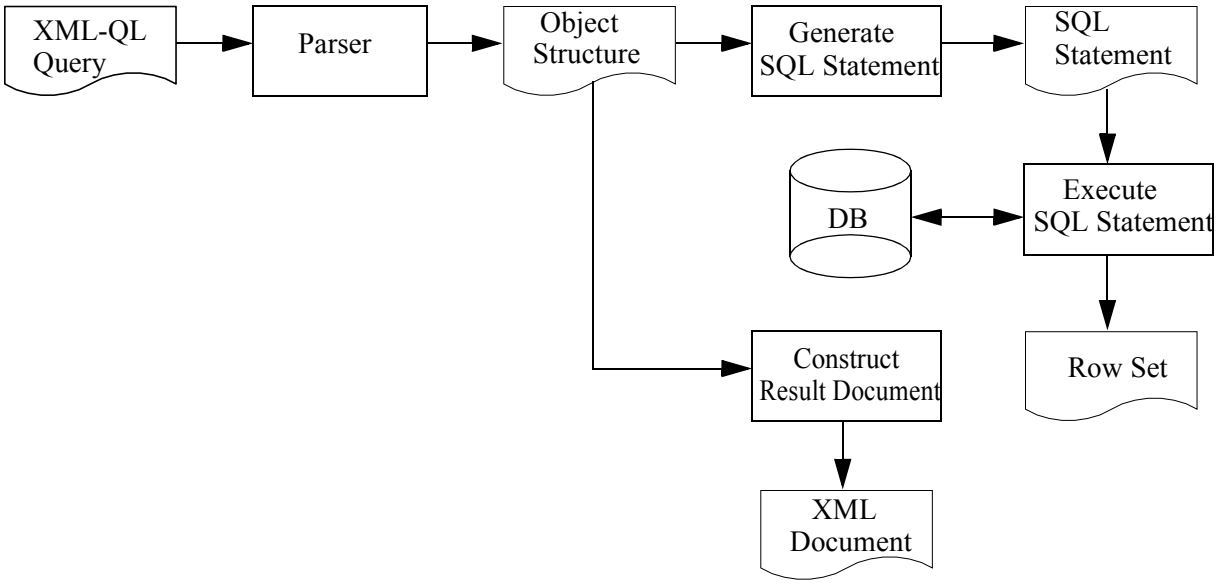
The content of a document is stored in a leaf value (in the `Leaf` table) or in an attribute value (in the `Attr` table). Both are referenced from the `Edge` table via the foreign keys `leafid` and `attrid`. In our approach, we don't bother about the decision whether to choose an attribute or element text to store content. The edges of the document tree are identified by source node and target node, denoted as `sourceid` and `targetid`. Each document has a unique document identifier so that an edge can be assigned to one document. The `type` column of the `Edge` table is used to classify the edge type that can be attribute, leaf, or reference. The names of elements are stored in the `name` column. The `depth` represents the depth of the subtree relative to the current node, which is used for the generation of the resulting XML documents when processing queries.

In order to store the document it has to be parsed before. This can be done best by an event-driven SAX parser using a stack to process the tags, without requiring a tree for the document import. The import algorithm starts with storing document name and id in the document table. For each start tag a node entry is generated and put on the stack. The same applies to attributes except that they are not stored in the stack. Element text is represented as leaf table entry; the corresponding node references the leaf entry via the leaf id.

2.2. Queries

Storing the XML documents as relations makes it simpler to retrieve or update the documents or parts of them. The usage of an XML query language supporting path expressions allows us to navigate throughout the whole document structure. The chosen implementation of the document model is based on XML-QL [XML-QL] that can be used as query language at the same time.

Therefore, an XML-QL query has to be compiled into an SQL statement that can executed by the RDBMS. This is done by a transformation of an XML-QL statement into an internal tree-based object structure consisting of a WHERE part and a CONSTRUCT part. The WHERE-Part of the object structure can easily be translated into the corresponding WHERE clause of the target SQL statement in a straightforward manner. After executing the SQL statement the retrieved tuples are processed with the CONSTRUCT part of the object structure to produce the resulting XML document.



Example

The following document has to be imported into the relational DBMS.

```
<tree>
       <person age = "36">
         <name>Peter</name>
         <address><street>Main Street 4</street>
                  <zip>04236</zip>
                  <city>Leipzig</city>
         </address>
     </person>
</tree>
```

After importing the document the database looks as following:

Edge

sourceid	targetid	leafid	attrid	docid	name	type	depth
0	1	NULL	NULL	1	tree	ref	3
1	2	NULL	NULL	1	person	ref	2
2	3	NULL	1	1	age	attr	0
2	4	1	NULL	1	name	leaf	0
2	5	NULL	NULL	1	address	ref	1
5	6	2	NULL	1	street	leaf	0
5	7	3	NULL	1	zip	leaf	0
5	8	4	NULL	1	city	leaf	0

Leaf

id	value
1	Peter
2	Main Street 4
3	04236
4	Leipzig

Attr

id	value
1	36

Doc

doc	url
1	sample.xml

The sample XML-QL query is executed on the document:

```
CONSTRUCT <result> {
     WHERE  <person>
              <name>$n</name>
              <address>$a>/address>
            </person>
     CONSTRUCT
            <person>
              <name>$n</name>
              <address>$a</address>
          </person>
} </result>
```

The XML-QL WHERE clause is the source for the generation of an SQL statement. The result tuples are generated using outer joins (+). In our example, the resulting SQL statement looks as follows:

```
SELECT   DISTINCT B.type AS n_type, B.targetid AS n_targetid,
           B.depth AS n_depth, C.value AS n_value,
           D.type AS a_type, D.targetid AS a_targetid,
           D.depth AS a_depth, E.value AS a_value
FROM     Edge A, Edge B, Leaf C, Edge D, Leaf E
WHERE    A.name = 'person' AND A.targetid = B.sourceid AND
         B.name = 'name' AND B.leafid = C.leafid(+) AND
         A.targetid = D.sourceid AND D.name = 'address' AND
         D.leafid = E.leafid(+)
```

This query returns one tuple:

n_type	n_targetid	n_depth	n_value	a_type	a_targetid	a_depth	a_value
leaf	4	0	Peter	ref	5	1	

The algorithm processes the CONSTRUCT part of the internal object structure for each tuple whose values replace the variables of the internal XML-QL representation. The variable `$n` (`name`) is an atomic value and can therefore be substituted by text. The variable `$a` is bound to a reference to a subtree that has to be inserted. This requires to submit another SQL query to the database that retrieves the subtree. With the depth of the subtree stored for each node, it becomes possible to reconstruct the subtree with a single SQL statement.

```
SELECT   A.name, A.type, Al.value AS a_leafval,
         Aa.value AS a_attrval
FROM     Edge A, Leaf Al, Attr Aa
WHERE    A.sourcid = 5 AND
         A.leafid = Al.leafid(+) AND A.attrid = Aa.attrid(+)
```

This query returns the tuples that form the subtree represented by the variable `$a`, which allows us to reconstruct the original XML document.

name	type	a_leafval	a_attrval
street	leaf	Main Street 4	
zip	leaf	04236	
city	leaf	Leipzig	

2.3 Experience Summary

Advantages

- *Vendor Independence:* The algorithms to store and query XML documents in a structure-oriented way are completely independent from the underlying RDBMS because no object-relational features are used that are specific to a RDBMS.
- *Stability:* The algorithms worked fine for larger documents as well. A sample document with 47000 edges could be processed without problems.
- *High Flexibility of Queries:* After storing the objects in the database it is easy to change single values or to retrieve them in a very flexible manner because the whole set of SQL functionality can be used. The same applies to the document structure that is represented in the relational database. It is possible to map XML-QL (or another XML query language) to SQL and to retrieve a new XML document. Although a search operation involves some join operations, it doesn't slow down the performance significantly because the decomposition is restricted to three tables.

Drawbacks

- *Decomposition of Documents:* The decomposition algorithm produces lots of tuples to be inserted into the database. The load time may increase for large document (in our scenario: 15 minutes for a 4 MB document). Yet, this can be optimized by using bulk loading tools, as Oracle and other vendors provide.
- *Restrictions of DBMS Data Types:* A frequent problem may occur with element text larger than 4000 characters. The varchar data type of an RDBMS is restricted to the maximum page size (in general 4000 bytes). Indexed columns are restricted to 3218 bytes. The definition of each element by a text data type would produce much overhead in most cases. An additional information in the document would be very helpful for this problem.
- *Information Loss:* The used mapping algorithm, like other available algorithms, does not care about some information stored in the original XML document. Among them are comments, processing instructions, external entities, and CDATA sections. The mapping algorithms need some enhancements to consider special cases such as multiple occurrence of text in one element.

3. XML Documents as Large Objects - Opaque Approach

3.1. Storage

An alternative to the structure-oriented approach is to store XML documents as Character Large Objects (CLOBs) in a relational database (opaque approach). The opaque approach presumes text database functionality, as most relational database vendors (e.g., Oracle intermedia Text, DB2 Text Extender) provide.

A document table contains a document ID (`id`) and a CLOB column in which the whole document (`xmlfile`) is stored. There is no available decomposition algorithm that preserves the document with all components such as comments and processing instructions. The access to XML documents that are stored as large objects is not yet standardized. The DBMS vendors offer some text database extensions. For our purpose, we applied the interMedia Text cartridge from Oracle. It mainly supports full-text retrieval and text indexing.

3.2. Queries

In this section, we compare the query facilities of InterMedia Text with XPath due to their similarities. Full-text SQL queries return the IDs of all documents containing the search string (WHERE CONTAINS clause). In order to reference XML elements within a text, the index has to be modified by using a component called a *sectioner*. Once the sectioner has been built with a CREATE INDEX statement, the text search can be narrowed as the following example shows. The query

```
SELECT id FROM xmltable
WHERE CONTAINS (xmlfile, 'searchstring WITHIN element') > 0
```

delivers results that are semantically similar to the XPath expression

```
//element[contains(.,'searchstring')]
```

There are some differences between XPath and CONTAINS queries: SQL CONTAINS queries apply to a set of documents, whereas XPath queries are restricted to a single document. The result of an XPath query can be a document fragment by selecting subelements. On the other hand, SQL queries return only full column values specified in the SELECT list. An XPath query can match documents based on the pure existence of elements. Since SQL CONTAINS has been designed for text searching, it does not support the search for the existence of an element. XPath `contains()` does substring matching. The SQL CONTAINS, on the other hand, does *word* matching. Another difference affects phrase searching. When searching for phrases, interMedia looks for two words in a specific order; intervening whitespace, newline, tab are ignored. XPath `contains()` does a strict substring matching. An expression `contains(..'faulty brakes')` would not accept any control character or punctuation between both words. The handling of structure-oriented queries as they can be expressed in XPath [XPath] requires the extension of SQL CONTAINS queries by path expressions.

The interMedia query language provides the WITHIN operator according to a fairly simple syntax:

`text_subquery` `WITHIN` *`elementname`*

The element name can be any XML tag. Unlike XPath, all case variations of the tag are matched. WITHIN has a higher precedence than the logical operators AND and OR. This requires the use of parentheses to force the WITHIN operator to apply to the whole subquery. The query `rain OR snow WITHIN weather` searches for documents, where the word "snow" appears in betweeen the <weather> tags, and documents where the word "rain" appears anywhere in the document. Using parentheses, the correct syntax would be

```
(rain OR snow) WITHIN weather.
```

This corresponds with the XPath expression:

```
.//weather[contains(.,'rain') or contains(.,'snow')]
```

An interMedia query can combine multiple WITHIN operators to express more complex searches. When querying heterogeneous documents the combination of WITHIN and OR can deal with synonymous tag names as the following example shows:

```
row1   <Last>Mueller</Last>
row2   <LastName>Mueller</LastName>
```

In order to search the whole document for Mueller, multiple WITHIN clauses have to be combined with OR:

```
(Mueller WITHIN Last) OR (Mueller WITHIN LastName)
```

This expression can be simplified by creating an XML sectioner index before. An XML sectioner indexes only user-defined tags at the time the index is created. The creation of section groups makes it possible to map multiple tags to the same section name, which allows a single WITHIN search to find content in different documents independently from the lexical expression of the tags.
Besides combining, the nesting of WITHIN operators is possible, which corresponds with more complex path expressions in XPath. The query

```
(xml WITHIN title) WITHIN book
```

should have the semantics of the XPath expression

```
.//Book/Title[contains(.,'xml')],
```

Yet, this cannot be guaranteed for the WITHIN operator that does not refer to the immediate parent element.

Attribute value searching with interMedia text has some important limitations because interMedia searches attribute values as text. To search attribute text, an attribute has to be specified in the element operand of the WITHIN clause. We consider the following example:

```
<book author = "Shakespeare">
     Romeo and Juliet
</book>
```

The query for `Shakespeare` appearing within the `author` attribute of the `<book>` element can be expressed as:

```
Shakespeare WITHIN book@author
```

close to the analogous XPath expression

```
.//book/@author[contains(.,'Shakespeare').
```

One important limitation is, however, that attributes cannot be nested. The CONTAINS query to search for a `book` that has the value `Shakespeare` in the `author` attribute and the string `Romeo` in the element text should look like this:

```
((Shakespeare WITHIN book@author) AND Romeo) WITHIN book
```

This kind of nesting with an attribute is currently not allowed in interMedia Text. The rewritten query, even though syntactically correct, does not have the same semantics:

```
(Shakespeare WITHIN book@author) AND (Romeo WITHIN book)
```

Those query cannot guarantee that elements are found that match both the attribut predicate and the content predicate. The found elements shall be filtered further on to restrict the result according to the original query. InterMedia Text as a text-searching engine does not allow us to express the search as exact equality search.

Numeric and date values are not type-converted because everything is considered as text in interMedia Text. This restricts the search for attribute values such as decimal values.
The query `3 WITHIN review@rating` would find the fragment:

```
<Review Rating = "3"></Review>
```

However, it cannot find this fragment due to the difference between "3" and "3.0" in the world of text:

```
 <Review Rating = "3.0"></Review>
```

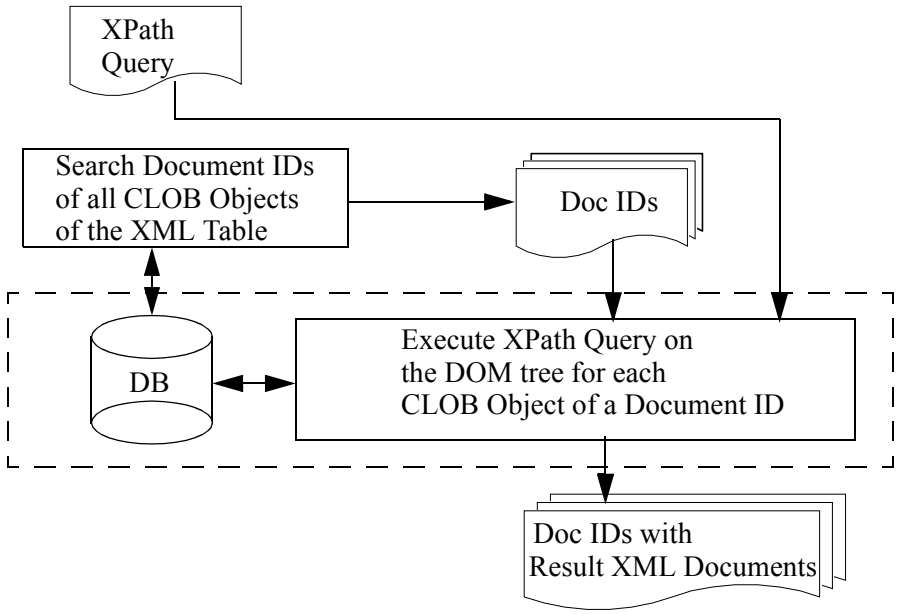
InterMedia Text provides no means to do range searches on attribute values or searches for the existence of attributes.

Some of the differences between XPath and Oracle can be resolved by changing the default configuration of interMedia Text (e.g., case sensitivity). The examples above show that interMedia Text imposes

several limitations, which makes it impossible to map every kind of XPath query to an equivalent SQL CONTAINS query. Therefore, it is desirable that a future version of Oracle interMedia will offer native XPath element extraction in its core engine.

Using the DOM-API

There is an alternative how to deal with XML CLOB objects using the Oracle XML Development Kit [XDK]. XDK comprises a set of PL/SQL packages installed on the server. Among them is an XML-DOM parser and an XSL processor. So it is possible to parse CLOB objects and to translate them into a DOM representation. The XSL processor provides methods to extract nodes using XPath queries on the DOM tree. The resulting node list can be transformed into an XML document that is sent to the client. This approach supports the usage of the full set of XPath functionality.



The drawback of this procedure is the overhead at the creation of the DOM tree before an XPath query can be submitted, which decreases the query performance. The processing of very large documents (more than 50 MB) is error-prone. There is a trade-off between the performance of interMedia text queries (using a text index) and the accuracy of XPath queries that construct XML documents from the matching document parts.

3.3 Experience Summary

Advantages

- *Information Preservation:* Compared to the structure-oriented approach, there is no loss of information about the original XML document during storage und retrieval.
- *Dealing with Large Documents:* The CLOB approach fits well for large document-centric documents, viz., documents with little structure und prose-rich elements that exceed the capacity of CHAR columns.
- *Different XML Document APIs:* There are different ways how to submit queries on XML documents stored as CLOBs. By using an XML-DOM parser a different document representation, capable of executing XPath queries at the API, is generated.

Drawbacks

- *Restricted Expressiveness of Text Queries:* It is hard to map an XML query language (such as XML-QL) to interMedia Text that is poor with respect to structure-oriented queries and result presentation (see the examples above).
- *Performance vs Accuracy:* There is a trade-off between performance and accuracy. InterMedia Text queries are faster than XPath queries because they rely on the index only. On the other hand, they cannot express the XPath Semantics needed for full-fledged XML queries.
- *Restrictions of Indexes:* With interMedia Text it is only possible to index XML tags with a maximum length of 64 characters. Besides the tag name, the length also includes prefix and namespace.
- *Problems with Markup:* Once character entities are decoded, they can be retrieved in the stored document that loses its well-formedness, on the other hand.
- *Vendor Dependence:* The implementation of the CLOB approach depends on the functionality of the text engine. Therefore, the XML-DB interface is determined by the proprietary platform. This can pose problems when deploying the CLOB interface in more complex applications.
- *Stability:* Very large objects (> 50 MB) cannot be transformed into a DOM with the Oracle XDK due to occurring memory errors. Depending on the DBMS configuration, it may happen even with smaller documents.

4. A Unified XML DB Interface

Depending on the processing requirements to documents, the structure-oriented approach or the opaque approach may be preferred to store an XML document in a relational database. An XML integration layer as a middleware component can combine different approaches and hide the implementation by an abstract XML DB interface. This interface provides common methods regardless of the underlying storage model. The methods are as follows:

- ImportDocument: The XML document is stored as CLOB object, or it is decomposed into tuples that are inserted into the relational database.
- ExportDocument: The XML document is exported from the database, i.e., the CLOB object is retrieved, or the correponding SQL statements are executed to reconstruct the whole document.
- DeleteDocument: The XML document is deleted from the database, which includes the deletion of all parts that belong to the document.
- GetDocumentList: It retrieves the names of all XML documents stored in the database.
- ExecuteQuery: The query string is passed as parameter. Multiple query formats are accepted such as XML-QL (supported by the structure-oriented storage approach), SQL CONTAINS and XPath. The latter works on a set of XML documents stored as CLOBs that are transformed into DOM trees before the XPath query can be submitted.

These methods can be combined, e.g., the retrieval of a set of documents matching a predicate can be considered as a composition of the GetDocumentList and the ExecuteQuery methods.

5. Conclusion

Our work led to the conclusion that multiple storage models are justified because none of them can meet all requirements at once with respect to query capabilities, performance, or scalability. We have seen advantages and shortcomings in both approaches that support XML documents without schema. Although the opaque approach seems more appropriate for document-centric documents, the functionality of a text-searching engine does not cover the needs of an XML query language. The feasibility of the structure-oriented approach to map documents into tables is restricted to data-centric documents with little prose.

Commercial database systems provide some standard procedures how to store XML objects, presupposing schema information. The use of relational DBMSs as storage engines for XML documents has to consider the coexistence of different storage models.

Our work has proved that documents without schema can be mapped to relational databases because the target data model uses generic data types independently from the actual document structure. New object-relational features of relational DBMSs facilitate the definition of user-defined types that are needed for the handling of XML data. Insteaf of user-defined types that implement the schema of a specific document, a generic XML data type is required for all kinds of documents. The main challenge will be to unify the broad range of storage choices and query capabilities in the implementation of such an XML data type. A new topic, physical XML database design, is looming.

Acknowledgments

Many thanks to:
- Tobias Krumbein who implemented both approaches and contributed good ideas
- Alex Buchmann and the reviewers of an earlier version who helped to improve the paper
- the software company *forcont* Leipzig for suggestions and support.

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

References

[Bo00] R. Bourret: XML and Databases
http://www.rpbourret.com/xml/XMLAndDatabases.html

[FK99] D. Florescu, D. Kossmann: Storing and Querying XML Data using an RDBMS.
Data Engineering, Sept. 1999, Vol.22, No.3.

[Kr01] T. Krumbein: Speicher- und Anfragemethoden für XML-Dokumente ohne Schema,
Diplomarbeit (German), Leipzig Univ. of Applied Sciences, 2001.

[Mu00] S. Muench: Building Oracle XML Applications. O'Reilly & Associates, 2000.

[STH+99] J. Shanmugasundaram et. al: Relational Databases for Querying XML Documents:
Limitations and Opportunities., Proc. 25th VLDB conference, 1999.

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

[XML-QL] World Wide Web Consortium: http://www.w3c.org/TR/NOTE-xml-ql

[XPath] World Wide Web Consortium: http://www.w3c.org/TR/xpath