Storing and Querying XML Data using an RDMBS

Daniela Florescu
INRIA, Roquencourt
daniela.florescu@inria.fr

Donald Kossmann
University of Passau
kossmann@db.fmi.uni-passau.de

1 Introduction

XML is rapidly becoming a popular data format. It can be expected that soon large volumes of XML data will exist. XML data is either produced manually (like html documents today), or it is generated by a new generation of software tools for the WWW and/or electronic data interchange (EDI).

The purpose of this paper is to present the results of an initial study about storing and querying XML data. As a first step, this study was focussed on the use of relational database systems and on very simplistic schemes to store and query XML data. In other words, we would like to study how the simplest and most obvious approaches perform, before thinking about more sophisticated approaches.

In general, numerous different options to store and query XML data exist. In addition to a relational database, XML data can be stored in a file system, an object-oriented database (e.g., Excelon), or a special-purpose (or semi-structured) system such as Lore (Stanford), Lotus Notes, or Tamino (Software AG). It is still unclear which of these options will ultimately find wide-spread acceptance. A file system could be used with very little effort to store XML data, but a file system would not provide any support for querying the XML data. Object-oriented database systems would allow to *cluster* XML elements and sub-elements; this feature might be useful for certain applications, but the current generation of object-oriented database systems is not mature enough to process complex queries on large databases. It is going to take even longer before special-purpose systems are mature.

Even when using an RDBMS, there are many different ways to store XML data. One strategy is to ask the user or a system administrator in order to decide how XML elements are stored in relational tables. Such an approach is supported, e.g., by Oracle 8i. Another option is to infer from the DTDs of the XML documents how the XML elements should be mapped into tables; such an approach has been studied in [4]. Yet another option is to analyze the XML data and the expected query workload; such an approach has been devised, e.g., in [2]. In this work, we will only study very simple *ad-hoc* schemes; we think that such a study is necessary before adopting a more complex approach. The schemes that we analyze require no input by the user, they work in the absence of DTDs or if DTDs are meaningless, and they do not involve any analysis of the XML data. Due to their simplicity, the approaches we study will not show the best possible performance, but as we will see, some of them will show very good query performance in most situations. Also, there is no guarantee that any of the more sophisticated approaches known so far will perform better than our simple schemes; see [3] for some experimental results in this respect. Furthermore, the results of our study can be used as input for more sophisticated approaches.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Mapping XML Data into Relational Tables

The starting point is one or a set of XML documents. We propose to scan and parse these documents one at a time and store all the information into relational tables. For simplicity, we assume here that an XML document can be represented as an ordered and labeled directed graph. Each XML element is represented by a node in the graph; the node is labeled with the *oid* of the XML object[1]. Element-subelement relationships are represented by edges in the graph and labeled by the name of the subelement. In order to represent the order of subelements of an XML object, we also order the outgoing edges of a node in the graph. Values (e.g., strings) of an XML document are represented as leaves in the graph. In all, we consider six ways to store XML data (i.e., graphs) in a relational database: three alternative ways to store the edges of a graph and two alternative ways to store the leaves (i.e., values), resulting in overall three times two different schemes. Other schemes and variants of the schemes presented in this paper are described and discussed in [3]. In particular, we describe and evaluate a scheme in [3] which would take advantage of an object-relational database system's feature to store multi-valued attributes.

Representing XML data as a graph is a simplification and some information can be lost in this process. The reason is that our graph model does not differentiate between XML subelements and attributes, and it is does not differentiate between subelements and references (i.e., IDREFs). Using one of our schemes, therefore, the original XML document cannot exactly be reconstructed from the relational data. However, these simplifications can easily be alleviated with additional bookkeeping in the relational database.

In order to show how XML data is mapped into relational tables in each scheme, we will use the following XML example which contains information about four persons:

```
⟨person⟩ ⟨id='1' age='55'⟩
    ⟨name⟩ Peter ⟨/name⟩
    ⟨address⟩ 4711 Fruitdale Ave. ⟨/address⟩
    ⟨child⟩
        ⟨person⟩ ⟨id='3' age='22'⟩
            ⟨name⟩ John ⟨/name⟩
            ⟨address⟩ 5361 Columbia Ave. ⟨/address⟩
            ⟨hobby⟩ swimming ⟨/hobby⟩
            ⟨hobby⟩ cycling ⟨/hobby⟩
        ⟨/person⟩
    ⟨/child⟩
    ⟨child⟩
        ⟨person⟩ ⟨id='4' age='7'⟩
            ⟨name⟩ David ⟨/name⟩
            ⟨address⟩ 4711 Fruitdale Ave. ⟨/address⟩
        ⟨/person⟩
    ⟨/child⟩
⟨/person⟩

⟨person⟩ ⟨id='2' age='38' child='4'⟩
    ⟨name⟩ Mary ⟨/name⟩
    ⟨address⟩ 4711 Fruitdale Ave. ⟨/address⟩
        ⟨hobby⟩ painting⟨/hobby⟩
⟨/person⟩
```

2.1 Mapping Edges

2.1.1 Edge Approach

The simplest scheme is to store all edges of the graph that represents an XML document in a single table; let us call this table the *Edge* table. The *Edge* table records the oids of the source and target objects of each edge of the graph, the label of the edge, a flag that indicates whether the edge represents an inter-object reference (i.e., an internal node) or points to a value (i.e., a leaf), and an ordinal number because the edges are ordered, as mentioned above. The *Edge* table, therefore, has the following structure:

[1] We assume that every XML element has a unique identifier. In the absence of such an identifier in the imported data, the system will automatically generate one.

Edge

source	*ordinal*	*name*	*flag*	*target*
1	1	age	int	*v1*
1	2	name	string	*v2*
1	3	address	string	*v3*
1	4	child	ref	**3**
1	5	child	ref	**4**
2	1	age	int	*v4*
...	...	...	...	

V_{int}

vid	*value*
v1	55
v4	38
v8	22
v13	7

V_{string}

vid	*value*
v2	Peter
v3	4711 Fruitdale Ave.
v5	Mary
v6	4711 Fruitdale Ave.
v7	painting
...	...
v15	4711 Fruitdale Ave.

Figure 1: Example: Edge Table with Separate Value Tables

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

The key of the *Edge* table is {*source, ordinal*}. Figure 1 shows how the *Edge* table would be populated for our XML example. The figure also shows one particular way to store values in separate *Value* tables; this approach is explained in more detail in Section 2.2. The bold faced numbers in the *target* column of the *Edge* table (i.e., **3** and **4**) are the oids of the target objects. The italicized entries in the *target* column refer to representations of values (explained later).

In terms of indices, we propose to establish an index on the *source* column and a combined index on the {*name, target*} columns. The index on the *source* column is useful for forward traversals such as needed to reconstruct a specific object given its *oid*. The index on {*name, target*} is useful for backward traversals; e.g., "find all objects that have a child named John." We experimented with different sets of indices as part of our performance experiments. We found these two indices to be the overall most useful ones.

2.1.2 Binary Approach

In the second mapping scheme, we propose to group all edges with the same label into one table. This approach resembles the binary storage scheme proposed to store semi-structured data in [5]. Conceptually, this approach corresponds to a horizontal partitioning of the *Edge* table used in the first approach, using *name* as the partitioning attribute. Thus, we create as many *Binary* tables as different subelement and attribute names occur in the XML document. Each *Binary* table has the following structure:

B_{name}(source, ordinal, flag, target)

The key of such a *Binary* table is {*source, ordinal*}, and all the fields have the same meaning as in the *Edge* approach. In terms of indices, we propose to construct an index on the *source* column of every *Binary* table and a separate index on the *target* column. This is analogous to the indexing scheme we propose to use for the *Edge* approach.

2.1.3 Universal Table

The third approach we study generates a single *Universal* table to store all the edges. Conceptionally, this *Universal* table corresponds to the result of a full outer join of all *Binary* tables. The structure of the *Universal* table is as follows, if $n_1, \ldots, n_k$ are the label names.

Universal(source, $\mathrm{ordinal}_{n_1}$, flag_{n_1}, target_{n_1}, $\mathrm{ordinal}_{n_2}$, flag_{n_2}, target_{n_2}, ..., $\mathrm{ordinal}_{n_k}$, flag_{n_k}, target_{n_k})

Figure 2 shows the instance of the *Universal* table for our XML example. As we can see, the *Universal* table has many fields which are set to *null*, and it also has a great deal of redundancy; the value *Peter*, for instance, is represented twice because Object 1 has two *child* edges. (How values are exactly represented is described in the next section.) In other words, the *Universal* table is denormalized—with all the known advantages and disadvantages of such a denormalization. Corresponding to the indexing scheme of the *Binary* approach, we propose to establish separate indices on all the *source* and all the *target* columns of the *Universal* table.

source	...	ord_{name}	$targ_{name}$	...	ord_{child}	$targ_{child}$	ord_{hobby}	$targ_{hobby}$
1	...	2	*Peter*	...	4	**3**	null	null
1	...	2	*Peter*	...	5	**4**	null	null
2	...	2	*Mary*	...	4	**4**	5	*painting*
3	...	2	*John*	...	null	null	4	*swimming*
3	...	2	*John*	...	null	null	5	*cycling*
4	...	2	*David*	...	null	null	null	null

Figure 2: Example Universal Table

B_{hobby}

source	*ord*	val_{int}	val_{string}	*target*
2	5	null	*painting*	null
3	4	null	*swimming*	null
3	5	null	*cycling*	null

B_{child}

source	*ord*	val_{int}	val_{string}	*target*
1	4	null	null	**3**
1	5	null	null	**4**
2	4	null	null	**4**

Figure 3: Example: Binary Tables with Inlining

2.2 Mapping Values

We now turn to alternative ways to map the values of an XML document (e.g., strings like "Peter" or "4711 Fruitdale Ave."). We study two variants in this work: (a) storing values in separate *Value* tables; (b) storing values together with edges. Both variants can be used together with the *Edge, Binary*, and *Universal* approaches, resulting in a total of six possible mapping schemes.

2.2.1 Separate Value Tables

The first way to store values is to establish separate *Value* tables for each conceivable data type. There could, for example, be separate *Value* tables storing all integers, dates, and all strings.[2] The structure of each *Value* table is as follows, where the type of the *value* column depends on the *type* of the *Value* table:

V_{type}(vid, value)

Figure 1 shows how this approach can be combined with the *Edge* approach. The *vids* of the *Value* tables are generated as part of an implementation of the mapping scheme. The *flag* column in the *Edge* table indicates in which *Value* table a value is stored; a *flag* can, therefore, take values such as *integer, date, string*, or *ref* indicating an inter-object reference. In the very same way, separate *Value* tables can be established for the *Binary* and *Universal* approaches. In terms of indices, we propose to index the *vid* and the *value* columns of the *Value* tables.

2.2.2 Inlining

The obvious alternative is to store values and attributes in the same tables. In the *Edge* approach, this corresponds to an outer join of the *Edge* table and the *Value* tables. (Analogously, this corresponds to outer joins between the *Binary* and *Universal* tables for the other approaches.) Hence, we need a column for each data type. We refer to such an approach as *inlining*. Figure 3 shows how inlining would work for the *Binary* approach. Obviously, no *flag* is needed anymore, and a large number of *null* values occur. In terms of indexing, we propose to establish indices for every *value* column separately, in addition to the *source* and *target* indices.

3 Performance Experiments and Results

We carried out a series of performance experiments in order to study the tradeoffs of the alternative mapping schemes and the viability to store XML data in an RDBMS. In this paper, we present the size of the resulting relational database for each mapping scheme, the time to reconstruct an XML document from the relational data,

[2] XML currently does not differentiate between different data types, but there are several proposals to extend XML in this respect.

n	100,000	number of objects
f_n	4	maximum number of attributes with inter-object references per object
f_v	9	maximum number of attributes with values per object
s	15	size of a short string value [bytes]
t	500	size of a long text value [bytes]
p_s	80	percent of the values that are strings
p_t	20	percent of the values that are text
d	20	number of different attribute names
l	10	size of an attribute name [bytes]

Table 1: Characteristics of the XML Document

and the time to execute different classes of XML queries. Other experimental results such as bulkloading times and times to execute different kinds of update functions are presented in [3].

To simplify the discussion, we will only present experimental results for four of the six alternative mapping schemes described in Section 2. We will study the *Edge, Binary*, and *Universal* approaches with separate *Value* tables in order to study the tradeoffs of the different ways to map edges. In addition, we will study the *Binary* approach with inlining in order to compare inlining and the separate *Value* tables variants.

As an experimental platform, we use a commercial relational database system[3] installed on a Sun Sparc Station 20 with two 75 MHz processors and 128 MB of main memory and a disk that stores the database and intermediate results of query processing. The machine runs under Solaris 2.6. In all our experiments, we limited the size of the main memory buffer pool of the database system to 6.4 MB, which was less than a tenth of the size of the XML document. Other than that, we use the default configuration of the database system, if not stated otherwise. (For some experiments, we used non-default options for query optimization; we will indicate those experiments when we describe the results.) All software which runs outside of the RDBMS (e.g., programs to prepare the XML document for bulkloading) is implemented in Java and runs on the same machine. Calls to the relational database from the Java programs are implemented using JDBC.

3.1 Benchmark Specification

3.1.1 Benchmark Database

The characteristics of the synthetic XML document we generate for the performance experiments are described in Table 1. The XML document consists of n objects. Each object has $0..f_n$ attributes containing inter-object references (i.e., IDREFs) and $0..f_v$ attributes with values. The document is flat; that is, there is no nesting of objects. (Given our graph model described in Section 2, flat documents with IDREFs are stored in the same way as documents with nested objects.) All attributes are labeled with one of d different attribute names; we will refer to these names as $a_1, \ldots, a_d$, but in fact each name is l bytes long. There are two types of values: short strings with s bytes and long texts with t bytes. $p_s\%$ of the values are strings and $p_t\%$ of the values are text. We use a uniform distribution in order to select the number of attributes for each object individually and to determine the objects referenced by an object and the name of every attribute. The graph that represents the XML document contains cycles, but this fact is not relevant for our experiments.

Since the XML document contains values of two different data types (string and text), two *Value* tables are generated in the relational database for the mapping schemes without inlining and two *value* columns are included in the *Binary* scheme with inlining. We index the strings completely, as proposed in Section 2.2, but we do not index the text (for obvious reasons), deviating from the proposed indexing scheme of Section 2.2. Strings and text, as well as attribute names (in the *Edge* table) are represented as `varchars` in the relational database. *flags* are represented as `chars`, and all other information (e.g., *oids, vids, ordinals,* etc.) is represented as `number(10,0)`.

The parameter settings we use for our experiments are also shown in Table 1. We create a database with 100,000 objects. Each object has, on an average, two attributes with inter-object references and 4.5 attributes with values. So, we have a total of approximately 450,000 values; 90,000 texts of 500 bytes and 360,000 short strings of 15 bytes.

[3]Our licenses agreement does not allow us to publish the name of the database vendor.

Query	*Description*	*Feature*
Q1	reconstruct XML object with oid = 1	select by oid
Q2	find objects that have attribute a_1 with value in certain range	select by value
Q3	find objects that have attributes a_1 and a_2 with certain values	two predicates
Q4	find objects that have a_1 and a_2 with certain value or just a_1 with certain value	optional predicate
Q5	find objects that have a_1 or a_2 or a_3 with certain value	predicate on attribute name
Q6	find object that match a complex pattern with seven references and eight nodes	pattern matching
Q7	find all objects that are connected by a chain of a_1 references to an object with a specific a_1 value	regular path expression
Q8	find all objects that are connected by a chain of a_1 or a_2 references to an object with a specific a_1 or a_2 value	regular path expression with a predicate on the attribute name

Table 2: Benchmark Query Templates

Q1	Q2L	Q2H	Q3L	Q3H	Q4L	Q4H	Q5L	Q5H	Q6L	Q6H	Q7L	Q7H	Q8L	Q8H
9	11	1805	3	131	9	1386	50	5556	1	3	11	2309	37	4616

Table 3: Size of Result Sets of Benchmark Queries

3.1.2 Benchmark Queries

Table 2 describes the XML-QL query templates that we use for our experiments. The XML-QL formulation for these queries is given in [3]. These query templates test a variety of features provided by XML-QL, including simple selections by oid and value, optional predicates, predicates on attribute names, pattern matching, and regular path expressions. In all, we test fifteen queries as part of our benchmark. We test each of the Q2 to Q8 templates in two variants: one *light* variant in which the predicates are very selective so that index lookups are effective and intermediate results fit in memory, and one *heavy* variant in which the use of indices is typically not attractive and intermediate results do not fit into the database buffers. Specifically, we set the predicates on a_1 to select 0.1% of the values in the light query variants and to select 10% of the values in the heavy variants. The predicates on a_2 are always set to select 30% of the values. All predicates involve short strings only (no text). For our benchmark database, the size of the result sets for each of these fifteen benchmark queries is listed in Table 3. To execute these XML-QL queries, we translate them into SQL queries. How this translation is done for each mapping scheme is outlined in [3] and beyond the scope of this paper.

To get reproducible experimental results, we carry out all benchmark queries in the following way: every query is carried out once to warm up the database buffers and then at least three times (depending on the query) in order to get the mean running time of the query. Warming up the buffers impacts the performance of the light queries that operate on data that fits in main memory; warming up the buffers, however, does not impact the results of the heavy queries.

3.2 Database Size

Table 4 shows the size of the XML document and of the resulting relational database for each mapping scheme. The size of the XML document is about 80 MB. We see that even without indices every mapping scheme produces a larger relational database. The *Universal* approach, of course, produces the most base data because the *Universal* table is denormalized as described in Section 2.1. Comparing the *Binary* approach with and without inlining, we see that inlining results in a smaller relational database: no *vids* are stored in the inline variant and *nulls* which are produced by the inline variant are stored in a very compact way by our RDBMS. Looking at the size of the indices, we can see that indices can consume up to 40% of the space.

	XML	Binary	Edge	Universal	Bin.+Inline
base data	79.2	105.2	122.3	138.9	86.9
indices	–	71.1	85.6	76.7	52.7
total	79.2	176.3	207.9	215.6	139.6

Table 4: Database Sizes [MB]

	Binary	Edge	Universal	Bin.+Inline
Q1	0.036	0.023	0.074	0.024
Q2(l)	0.104/4.6	0.089/5.3	0.093/4.8	0.011/5.3
Q2(h)	15.7	83.0	62.1	0.644/5.5
Q3(l)	6.0	5.1	5.8	2.0
Q3(h)	15.8	133.7	70.5	3.5
Q4(l)	12.3	9.9	11.7	4.1
Q4(h)	32.0	255.7	132.9	6.7
Q5(l)	0.277/15.4	5.1	14.2	0.028/13.9
Q5(h)	48.6	148.1	185.8	14.8
Q6(l)	0.130/6.5	6.1	0.141/6.3	0.017/2.0
Q6(h)	17.0	123.7	63.7	3.3
Q7(l)	0.111/6.2	0.101/5.4	0.096/6.2	0.012/5.3
Q7(h)	16.8	221.5	62.7	1.060/6.6
Q8(l)	18.3	5.0	91.4	32.7
Q8(h)	47.2	392.0	206.9	36.3

Table 5: Running Times of Queries [secs]; Tuned/Untuned

3.3 Running Times of the Queries

Table 5 shows the running times of our fifteen benchmark queries for each mapping scheme. In most cases, the optimizer of the RDBMS found good plans with the default configuration. In some cases, however, we were able to get significant improvements by using a non-default configuration; for such cases, Table 5 shows the running times obtained using the untuned (default) optimizer configuration and the tuned optimizer configuration. Most of the improvements were achieved for light queries and by forcing the optimizer to use indices instead of table scans and index nested-loop joins instead of hash or sort-merge joins.

The main observation is that the best mapping scheme (*Binary* with inlining) shows very good performance. For all queries, the running time is acceptable. The reason is that today's relational query engines are very powerful, even if the queries involve many joins and recursion.

Comparing the alternative mapping schemes, we can see that the *Binary* approach wins over the *Edge* and *Universal* approaches and that inlining beats separate *Value* tables. Both of these results can be explained fairly easily. The *Edge* approach performs poorly for heavy queries because joins with the (large) *Edge* table become expensive in this case; in effect, most of the data in the *Edge* table is irrelevant for a specific query. For the same reason, the *Universal* approach with its very large *Universal* table performs poorly for heavy queries. In the *Binary* approach, on the other hand, only relevant data is processed. The same kind of benefits of a binary table approach have been observed in the Monet project for (structured) TPC-D data [1]; for XML data the benefits are particularly high. Explaining the differences between inlining and separate *Value* tables is even easier: inlining simply wins because it saves the cost of the joins with the *Value* tables. The results show that inlining beats separate *Value* tables even if very large values (such as text) are inlined. Inlining would also win if many different types are involved and *null* values are stored in a compact way by the RDBMS.

Q8(l) and to some extent Q1 and Q5(l) are exceptions to the above rules. Q8 involves a predicate on the attribute names. The *Edge* approach is attractive for such queries because such predicates can directly be applied to the *Edge* table. Executing such predicates involves the generation of an SQL `UNION` query which carries out duplicate work for the other mapping schemes.

3.4 Reconstructing the XML Document

Table 6 shows the overall time to reconstruct the XML document (and write it to disk) from the relational data for each mapping scheme. In all cases, it takes more than 30 minutes, and this fact is probably the most compelling argument against the use of RDMBSs to store XML data. All mapping schemes need to sort by oid in order to re-group the objects, and this sort is expensive in our environment (it is an 80 MB sort with 6.4 MB of memory). The disastrous running time for the *Universal* approach with separate *Value* tables can also be explained. The *Universal* table must be scanned $d = 20$ times (once for each attribute name) in order to restructure the data and carry out the joins with the *Value* tables. These observations indicate that it might be advantageous to store

Binary	Edge	Universal	Bin.+Inline
56m 52s	40m 56s	1h 41m 17s	32m 8s

Table 6: Reconstructing the XML Document

copies of the original XML documents in the file system in addition to loading the XML data into an RDBMS. In general, there is no way to store data to meet the requirements of all purposes. Depending on the workload, multiple copies in possibly different formats are needed – XML data is no exception to this rule.

4 Conclusion

We studied alternative mapping schemes to store XML data in a relational database. The mapping schemes we studied are extremely simple. Due to their simplicity, they will never be the best choices, but our experiments indicate that even with such simple mapping schemes, it is possible to obtain very good query performance. The only operation which had unacceptably high cost was completely reconstructing a very large XML document; more sophisticated mapping schemes, however, would show poor performance for this operation as well.

This study was only a first step towards finding the best way to store XML data. Our results can be used as a basis to develop and configure more sophisticated mapping schemes. Also, more experiments with different kinds of (real and synthetic) XML data are required. In addition, other characteristics such as authorization, locking behavior etc. need to be studied. Furthermore, performance experiments with OODBMSs and special-purpose XML data stores ought to be conducted. The XML document and the XML-QL and SQL queries we used for our experiments can be retrieved from the authors' Web pages.

References

[1] P. Boncz, A. Wilschut, and M. Kersten. Flattening an object algebra to provide performance. In *Proc. of ICDE*, Orlando, FL, 1998.

[2] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of ACM SIGMOD*, Philadelphia, PN, 1999.

[3] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report, INRIA, France, 1999.

[4] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, Edinburgh, Scotland, 1999.

[5] R. v. Zwol, P. Apers, and A. Wilschut. Modelling and querying semistructured data with MOA. *Workshop on Query processing for semistructured data and non-standard data formats*, 1999.