A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database

Daniela Florescu
Inria, France
dana@rodin.inria.fr

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

August 3, 1999

Abstract

XML is emerging as one of the dominant data formats for data processing on the Internet. To query XML data, query languages like XQL, Lorel, XML-QL, or XML-GL have been proposed. In this paper, we study how XML data can be stored and queried using a standard relational database system. For this purpose, we present alternative mapping schemes to store XML data in a relational database and discuss how XML-QL queries can be translated into SQL queries for every mapping scheme. We present the results of comprehensive performance experiments that analyze the tradeoffs of the alternative mapping schemes in terms of database size, query performance and update performance. While our discussion is focussed on XML and XML-QL, the results of this paper are relevant for most semi-structured data models and most query languages for semi-structured data.

1 Introduction

It has become clear that not all applications are met by the relational, object-relational, or object-oriented data models. Examples are applications that need to integrate data from several data sources or applications for which the schema is not known at the time the data is generated[Bos99]. To support these kinds of applications, semi-structured data models have been proposed [Abi97, Bun97, Suc98]. One common feature of these data models is the lack of schema so that the data is self-describing. XML is emerging as a standard to define such semi-structured data. Also, to retrieve data from such semi-structured databases, special semi-structured query languages have been proposed [QL'98]; examples are XQL [RLS98], Lorel [AQM+97], XML-QL [DFF+99], or XML-GL [CCD+99]. Common features of these languages are the use of regular path expressions and the ability to extract information about the schema from the data.

There are three possible approaches to store semi-structured data (i.e., XML documents) and to execute queries on that data. One, build a special-purpose database system. Example research prototypes are Rufus [SLS+93], Lore [MAG+97] and Strudel [FFK+98]; Lotus Notes is an example commercial product[Lot98]. Such a system is particularly tailored to store and retrieve XML data, using specially designed structures and indices[MWA+98, Moh99] and particular query optimization techniques[FLS97, MW97]. To some extent SGML database systems [YIU96, SD96] or systems like GRAS [KSW95] which are designed to store graphs fall into this category of special-purpose systems as well. Two, use an object-oriented database system. In this approach, the rich data modeling capabilities of OODBMSs are exploited. This approach has, for example, been studied in [CACS94] and implemented in commercial systems like O_2 or Objectsore. This approach is also pursued as part of the Monet project [vZAW99]. Three, use a (standard) relational database system. In this approach, XML data is *mapped* into tables of a relational schema and queries posed

in a semi-structured query language are translated into SQL queries. Apparently, Oracle and Microsoft are currently building tools to facilitate this approach.

It is still unclear which of these three approaches is going to find wide-spread acceptance. In theory, special-purpose systems should work best, but it is going to take a long time before such systems are mature and scale well for large amounts of data. Likewise, the current generation of object-oriented database systems is not yet mature enough to evaluate queries on very large databases. Relational database systems are mature and scale very well, and they have the additional advantage that in a relational database XML data and traditional (structured) data can co-exist making it possible to build applications that involve both kinds of data with little extra effort. Relational databases, however, have been built to support traditional (structured) data and the requirements of processing XML data are vastly different from the requirements to process such traditional data. To optimize the use of relational database systems for XML, recent work has concentrated on models and algorithms to *extract schema* from XML documents or semi-structured data in general; e.g., [BDFS97, NAM98, DFS]. The goal of that work is to analyze the semi-structured data and (possibly) the query workload of the target application in order to find the best approximated schema. This way the semi-structured data can be stored in the relational database with little offcuts, and schema extraction makes it also easier to formulate queries [GW97].

The purpose of this experience paper is to study the overall performance of relational databases to process XML data. Rather than *extracting schema*, our goal is to study the general advantages and pitfalls of using relational databases to store and manage XML data and to study the tradeoffs of fundamentally different schemes to store XML documents in relational databases. We describe five alternative mapping schemes that can be used to store XML documents, show how queries and updates are processed for each of these mapping schemes, and present the results of comprehensive performance experiments that analyze the space requirements, the bulkloading times, the running times to reconstruct an XML document, and the running times of a series of queries and update functions for each mapping scheme. The mapping schemes we study are very simple and can be implemented in an ad-hoc way, but they can also be improved by using one of the models to extract schema. Also, the results of our performance experiments can be used as input for such models. In our experiments, we use a synthetic experimental database and synthetic benchmark queries and update functions.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of XML and query languages for XML. Section 3 describes alternative mapping schemes that can be used to store XML documents in a relational database. Section 4 addresses query processing issues. Section 5 presents the results of our performance evaluation. Section 6 contains conclusions and suggestions for future work.

2 Data Model and Query Language

2.1 XML

The Extensible Markup Language (XML) has been created by the World Wide Web Consortium (W3C) as a simplified subset of SGML specially designed for Web applications. The goal of XML is to enable the delivery of self-describing data structures of arbitrary depth and complexity to applications that require such

structures. XML retains the key SGML advantages of extensibility, structure, and validation in a language that is designed to be vastly easier to learn, use, and implement than full SGML. XML differs from HTML in three major respects: (a) information providers can define new tag and attribute names at will, (b) document structures can be nested to any level of complexity, and (c) any XML document can contain an optional description of its grammar for use by applications that need to perform structural validation.

Structurally, each XML document consists of a set of elements, the boundaries of which are delimited by start-tags and end-tags. Each element has a type, identified by name, sometimes called its "generic identifier", and may have a set of attribute specifications. Each attribute specification has a name and a value. In addition, each element can have an arbitrary list of (nested) subelements. The example below gives a flavor of the XML language.

Example 2.1: Let's assume that a data source wants to export information (e.g. names, addresses and hobbies) of the members of a family. A possible way of structuring this information and representing it in XML is as follows. Each person has a unique identifier *id* and an *age* attribute. Information about the hobbies of a given person are provided as valued subelements. The *child* elements of Person 1 contain the children of the family as subelements, while the *child* element of Person 2 only contains a pointer to its child.

```
⟨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⟩
```

2.2 Data Model for semi-structured data

XML is nothing else than a particular standard syntax for semi-structured data exchange[Bos99, Suc98]. The need for managing semi-structured data arised independently of the Web[Bun97] and various aspects of managing semi-structured data have been extensively studied in the last years[Abi97, Bun97]. Broadly speaking, semi-structured data refers to data with some of the following characteristics: (a) the schema is not given in advance and may be implicit in the data, (b) the schema is relatively large (w.r.t. the size of the data) and may be changing frequently, (c) the schema is *descriptive* rather than *prescriptive*, i.e., it describes the current state of the data, but violations of the schema are still tolerated, (d) the data is not strongly typed, i.e., for different objects, the values of the same attribute may be of differing types.

Many models have been proposed in the literature for semi-structured data, the big majority being based on labeled directed graphs [Abi97, Bun97]. In this paper, we use a simple graph data model, similar to the OEM model proposed in [PGMW95]. In this model, a semi-structured database is modeled as a directed labeled graph in which the nodes model objects and the outgoing edges of an object model the attributes of the object. Edges are labeled with attribute names. Each object has a unique identifier and each internal object has a set of attributes. In addition, the leaves in this graph are labeled with data values (e.g. integers, strings, dates).[1] The edges whose target objects are leaves are called data attributes; the other attributes are called references. Unlike the object oriented data model, objects in the database are not constrained to have similar sets of attributes and the values of the data attributes are not constrained to be of the same type. The graph modeling the fragment of XML data given in Example 2.1 is depicted in Figure 2.2.

Mapping XML, which has been originally developed for mostly-text documents, into this theoretical data model raises several problems. First, is the order of attributes and subelements of an object relevant? In XML, the order can be sometimes artificially introduced for the purpose of serialization, even if the order is not semantically relevant. However, in order not to loose the order for the cases where order *is* relevant, we consider an ordered graph model in this paper; i.e., there is a total order on the set of outgoing arcs of a node in the graph. The second question concerns the distinction between attributes and subelements. In our work, we decided not to maintain this distinction, both being modeled as arcs in the graph. Finally, another decision concerns the modeling of XML references (i.e., IDREFs). In our model, we implement references as regular arcs in the graph. Obviously, the last two decisions imply loss of information when translating from XML to the data model. However, taking the distinction between attributes and elements and the particularities of references into consideration would not impact the results of our work, it would only involve some extra bookkeeping.

2.3 Query language

The state of the art contains abundant work on query languages for semi-structured data (e.g. [AQM+97, BDHS96, FFLS97]) or for XML[QL'98]. The major common characteristic of all these languages is the fact that they are all based on a labeled graph model. Moreover, all such languages emphasize the ability to query the schema of the data, and the ability to accommodate irregularities in the data, such as missing or repeated fields, heterogeneous records using regular path expressions.

In this paper we will take as an example the XML-QL query language[DFF+99]. XML-QL enables data extraction from XML documents and allows to express mappings between different ontologies. To query unknown or unpredictable data structures, XML-QL and other query languages for semi-structured data have two features in common: regular path expressions and the ability to query the schema (i.e., attributes and references). In addition, XML-QL has a powerful restructuring mechanism; that is, the result of an XML-QL query could be a complex XML document. In this paper, we will mostly ignore the restructuring part and concentrate on the data extraction part of the language because data extraction is always the costly part of query evaluation.

We consider queries to be of the form:

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

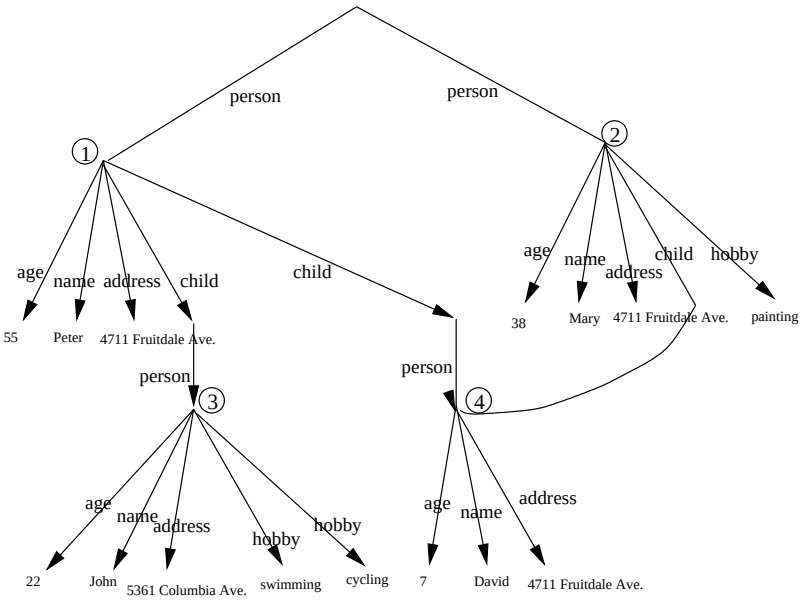

Figure 1: Data Graph Corresponding to the XML Fragment of Example 2.1.

select ⟨variable-list⟩ where ⟨XML-pattern⟩$^+$

The **where** clause is a conjunction of **XML-patterns**. Each **XML-pattern** is defined as a tree where the leaves are labeled with either constants, variables or unary predicates. Edges in the pattern are labeled with either string constants, variables, unary predicates or regular path expressions constructed using alternation, concatenation or a Kleene star over the alphabet of strings and the special "_" symbol (underscore) used to denote so-called wild cards which match every attribute. Furthermore, subpatterns can be marked as *optional*.

The query "find all the persons who are older than 18 and live in 4711 Fruitdale Ave., and retrieve their names with (possibly) their hobbies" would be expressed in XML-QL as follows:

```
select $n, $h
where   <person>
           <age> $a </age>
           <name> $n </name>
           <address> 4711 Fruitdale Ave.  </address>
           [<hobby> $h </hobby> ]
        </person>, $a>18
```

The evaluation of the query results in a table which contains bindings for all the variables used in the query. The semantic of the language is the natural one: An object in the database will match a pattern if the data tree rooted at the object matches the required pattern as a prefix and if all the predicates and constant selections in the pattern are satisfied. With optional edges (enclosed in brackets in the XML-QL syntax), objects qualify if they do not match that edge or if they match and the corresponding sub-tree matches the optional sub-pattern. Optional sub-patterns can, themselves, contain optional edges. In our example, the query returns two tuples ["Peter", *null*] and ["Mary", "painting"].

3 Storing XML Documents in a Relational Database

In this section, we describe alternative mapping schemes that can be used to store an XML document in a relational database. The starting point is the labeled graph which represents the XML document, as described in Section 2.2. A mapping scheme determines which tables to create in the relational database, which indices to construct, and in which tables to store the objects (internal nodes of the graph), attributes (edges), and values (leaves). Of course, there are many meaningful mapping schemes conceivable for a given XML document; in fact, indefinitely many. We will concentrate on basic, canonic mapping schemes in this section, and we characterize these mapping schemes along two dimensions: (a) how to represent *attributes*, and (b) how to represent *values*. Along the first dimension, we discuss four different variants and along the second dimension, we discuss two alternative schemes, resulting in overall eight mapping schemes. (We will assess five of these mapping schemes as part of our performance evaluation in Section 5.)

3.1 Mapping Attributes

3.1.1 The Edge Approach

The simplest scheme is to store all attributes 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 the attribute, the name of the attribute, a flag that indicates whether the attribute is an inter-object reference or points to a value, and an ordinal number used to recover all attributes of an object in the right order and to carry out updates if objects have several attributes with the same name. The *Edge* table, therefore, has the following structure:

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

The key of the *Edge* table is {*source, ordinal*}. Figure 2 shows how the *Edge* table would be populated for the example XML document from Section 2.1. The bold faced numbers in the *target* column (i.e., **3** and **4**) are the oids of the target objects. The italicized entries in the *target* column refer to representations of values. Values cannot be stored in this way, but we put these values into the table for illustration purposes, and we will discuss alternative ways to represent values for the *Edge* and other approaches in Section 3.2. In that section, we will also discuss the role of the *flag* field, which is not shown in Figure 2.

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 (not reported in this paper), and found these two indices to be the overall most useful ones.

A simple variant of the *Edge* approach is to store the attribute names in a separate table. We also experimented with that variant: that variant reduces the size of the database, but it significantly increases the cost of query processing because the {*name, target*} index can no longer be established, thereby slowing down backward traversals dramatically in some cases. We, therefore, will not consider this variant in the remainder of this paper.

source	*ordinal*	*name*	*target*
1	1	age	*55*
1	2	name	*Peter*
1	3	address	*Fruit.*
1	4	child	**3**
1	5	child	**4**
2	1	age	*38*
...	...	...	...

Figure 2: Example: Edge Table

A_{hobby}

source	*ordinal*	*target*
2	5	*painting*
3	4	*swimming*
3	5	*cycling*

A_{child}

source	*ordinal*	*target*
1	4	**3**
1	5	**4**
2	4	**4**

Figure 3: Example Attribute Tables

3.1.2 Attribute Approach

In the second mapping scheme, we propose to group all attributes with the same name into one table. This approach resembles the binary storage scheme proposed to store semi-structured in [vZAW99]. Conceptually, this approach corresponds to a horizontal partitioning of the *Edge* table used in the first approach, using *name* as the partitioning attribute. Thus, there we create as many *Attribute* tables as different attribute names in the XML document, and each *Attribute* table has the following structure:

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

The key of such an *Attribute* table is {*source, ordinal*}, and all the fields have the same meaning as in the *Edge* approach. Figure 3 shows the *hobby* and *child Attribute* tables for our example XML document from Section 2.1. In terms of indices, we propose to construct an index on the *source* column of every *Attribute* table and a separate index on the *target* column. This is analogous to the indexing scheme we propose to use for the *Edge* approach.

3.1.3 Universal Table

The third approach we study generates a single *Universal* table to store all the attributes of an XML document. This corresponds to a *Universal* table [Ull89] with separate columns for all the attribute names that occur in the XML document. Conceptionally, this *Universal* table corresponds to the result of an outer join of all *Attribute* tables. The structure of the *Universal* table is as follows, if $n_1, \ldots, n_k$ are the attribute names in the XML document.

$\text{Universal}(\text{source}, \text{ordinal}_{n_1}, \text{flag}_{n_1}, \text{target}_{n_1}, \text{ordinal}_{n_2}, \text{flag}_{n_2}, \text{target}_{n_2}, \ldots, \text{ordinal}_{n_k}, \text{flag}_{n_k}, \text{target}_{n_k})$

Figure 4 shows the instance of the *Universal* table for our example XML document. As we can see in Figure 4, 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 a multi-valued attribute (i.e., *child*).

source	...	ord_{name}	$targ_{\text{name}}$	...	ord_{child}	$targ_{\text{child}}$	ord_{hobby}	$targ_{\text{hobby}}$
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 4: Example Universal Table

source	...	ord_{name}	$flag_{\text{name}}$	$targ_{\text{name}}$	...	ord_{hobby}	$flag_{\text{hobby}}$	$targ_{\text{hobby}}$
1	...	2	-	*Peter*	...	null	null	null
2	...	2	-	*Mary*	...	5	-	*painting*
3	...	2	-	*John*	...	4	m	null
4	...	2	-	*David*	...	null	null	null

$Overflow_{\text{hobby}}$

source	*ord*	*target*
3	4	*swimming*
3	5	*cycling*

Figure 5: Example UnivNorm and Overflow Table

Putting it differently, the *Universal* table is denormalized—with all the known advantages and disadvantages of such a denormalization. Corresponding to the indexing scheme of the *Attribute* approach, we propose to establish separate indices on the *source* and all the *target* columns of the *Universal* table.

3.1.4 Normalized Universal Approach

The fourth approach, the *UnivNorm* approach, is a variant of the *Universal* table approach. The difference between the *UnivNorm* and *Universal* approach is that multi-valued attributes are stored in separate *Overflow* tables in the *UnivNorm* approach. (This corresponds to the way that object-relational databases like Oracle 8 would store multi-valued attributes.) For each attribute name that occurs in the XML document a separate *Overflow* table is established, following the principle of the *Attribute* approach.[2] This way, there is only one row per XML object in the *UnivNorm* table and that table is normalized. The structure of the *UnivNorm* table and the *Overflow* tables is as follows, if $n_1, \ldots, n_k$ are all the attribute names in the XML document:

$$\text{UnivNorm}(\text{source}, \text{ordinal}_{n_1}, \text{flag}_{n_1}, \text{target}_{n_1}, \text{ordinal}_{n_2}, \text{flag}_{n_2}, \text{target}_{n_2}, \ldots, \text{ordinal}_{n_k}, \text{flag}_{n_k}, \text{target}_{n_k})$$

$$\text{Overflow}_{n_1}(\text{source}, \text{ordinal}, \text{flag}, \text{target}), \ldots, \text{Overflow}_{n_k}(\text{source}, \text{ordinal}, \text{flag}, \text{target})$$

The key of the *UnivNorm* table is *source*. The key of an *Overflow* table is {*source, ordinal*}. If an attribute is single-valued, the *flag* field indicates whether the attribute refers to another object or a value, as in all the other approaches. If the attribute is multi-valued, the *flag* field is set to "m" to indicate this fact. If the object has not got an attribute with that name, the *flag* field is naturally set to null. Figure 5 shows the *UnivNorm* table and the *hobby Overflow* table for our example XML document. Again, we propose to establish separate indices on the *source* and all the *target* columns of the *UnivNorm* table as well as on the *source* and *target* columns of all the *Overflow* tables.

3.2 Mapping Values

We now turn to alternative ways to map the values of an XML document (e.g., strings like "Mary" or "4711 Fruitdale Ave."). We study two variants in this work: (a) storing values in separate *Value* tables; (b) storing values together with attributes. Both variants can be used together with the *Edge, Attribute, Universal*, and *UnivNorm* approaches, resulting in a total of eight possible mapping schemes. (In Section 5, however, we will only assess five of these eight mapping schemes for brevity.)

[2] An alternative would be to establish a single *Overflow* table for all multi-valued attributes and organize that *Overflow* table in the same way as the *Edge* table. We experimented with such an approach and found it to be inferior.

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 6: Example: Edge Table with Separate Value Tables

3.2.1 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.[3] The structure of each *Value* table would simply be as follows, where the type of the *value* column depends on the *type* of the *Value* table:

V_{type}(vid, value)

Figure 6 shows how this approach would be combined with the *Edge* approach, completing our example of Figure 2. The *vids* of the *Value* tables are generated as part of an implementation of the mapping scheme. The *flag* column in the other tables now 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 *Attribute, Universal,* and *UnivNorm* approaches. In terms of indices, we propose to index the *vid* and the *value* columns of the *Value* tables.

Looking closer at Figure 6, we observe that the value "4711 Fruitdale Ave." is stored three times in the *string* value table. The reason is that this value occurs three times in the original XML document and in the labeled graph that represents that document. Of course, it would be possible to find a more compact mapping of the original XML document by storing such *strings* only once, but such an approach would severely complicate the implementation of updates (e.g., it would require reference counting and garbage collection). We, therefore, will not study such a compact representation in this paper. After all, the author of the XML document could have established a separate *address* object which is referenced by the Peter, Mary, and David objects in order to get a compact representation and model that Peter, Mary, and David live at the same place.

3.2.2 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 *Attribute, Universal, UnivNorm*, and *Overflow* tables in the other approaches.) Hence, we need a column for each data type. We refer to such an approach as *inlining*. Figure 7 shows how inlining would work for the *Attribute* approach. Obviously, no *flag* is needed anymore, and a large number of *null*

[3] As stated in Section 2.2, XML currently does not differentiate between different data types, but there are several standard proposals to extend XML in this respect.

A_{hobby}

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

A_{child}

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

Figure 7: Example: Attribute Tables with Inlining

values occur. In terms of indexing, we propose to establish indices for every *value* column separately, in addition to the *source* and *target* indices.

An alternative to the representation with one value column for each data type would be to establish a single value column that stores all values as strings, the most general type. In such a scheme, all values would be converted to strings (e.g., 7 would be represented as "7"). Such an approach, however, would make it impossible to use an index in order to find all objects with `5 < age < 22`.

3.3 Other Mapping Schemes

As stated at the beginning of this section, there are, of course, many other mapping schemes conceivable. There are even many variants of the eight simple mapping schemes presented in the previous two sections conceivable; e.g., one or several *Overflow* tables in the *UnivNorm* approach or hybrid approaches such as establishing *Attribute* tables for frequent attributes and storing all the other attributes in an *Edge* table, or inlining small values (e.g., integers) and storing large values (e.g., text) in separate value tables. In the following, we will name just a few other mapping schemes that have been addressed in the literature or which we have heard of in discussions.

- Selective Outer Joins of *Attribute* tables: as a compromise between the *Attribute* and *Universal* approach, it is possible to, say, store *child* and *hobby* attributes in separate *Attribute* tables and to store *name, age, address* in a single, combined table; that is, to use $A_{\text{name}} \bowtie\!\!\!\!\!\!= A_{\text{age}} \bowtie\!\!\!\!\!\!= A_{\text{address}}$ instead of the individual *Attribute* tables. This approach is, of course, attractive if most objects that have a *name* also have an *age* and an *address*, if each of these attributes only occur once per object, and if queries that involve these objects typically ask for two or all three of these attributes. As stated in the introduction, models and algorithms to extract such information have been proposed in the literature, and we will discuss some of the tradeoffs of such an approach in Section 5.8.

- Selective Joins of *Attribute* tables: rather than storing $A_{\text{name}} \bowtie\!\!\!\!\!\!= A_{\text{age}} \bowtie\!\!\!\!\!\!= A_{\text{address}}$, this variant would store $A_{\text{name}} \bowtie A_{\text{age}} \bowtie A_{\text{address}}$. In addition, *Overflow* tables need to be established in order to store the information of objects that have, say, a *name* and *age*, but no *address*. Materializing joins is attractive if queries ask for *all* attributes whereas outer joins are also attractive if queries just ask for *subsets* of the attributes. Some of the related theory of materializing joins and outer joins has been developed in the context of *access support relations* [KM92].

- Redundancy: just as in any other database, materialized views can be established in order to speed up the execution of queries. Finding the right views to materialize has been an active research area in the context of data warehouses [HRU96, TS97]. It is likely to be at least as difficult in the context of mapping XML documents.

4 Query Processing and Updates

In this section, we will discuss some of the details of implementing queries and updates with the alternative mapping schemes. Going into the full details is beyond the scope of this paper so that we will concentrate on a brief overview of the kinds of operations that are involved in order to implement queries posed in an XML query language or to propagate updates carried out on the original XML document.

4.1 Queries

The overall approach is to translate an XML query into SQL, let the RDBMS execute the query, and do some postprocessing in order to get the right query result (e.g., generate XML if this is requested by the query). All three of these steps can be carried out in a straightforward way, but of course the most interesting step is the translation into SQL. To get a flavor for what the generated SQL looks like, and thus, what kinds of operations the RDBMS must carry out, we will characterize the generated SQL for different categories of queries and the alternative mapping schemes. We will give performance results in Section 5.

Reconstructing Objects of the XML Document One very important type of query is to reconstruct an object of the XML document, given the object's *oid* (i.e., get all attributes of the object without computing the transitive closure). Such a simple query can be executed as follows for each mapping scheme:

- *Edge*/separate *Value* tables: a simple (indexed) lookup by *source* on the *Edge* table followed by an (indexed) join with the *Value* tables
- *Attribute*/separate *Value* tables: an (indexed) lookup by *source* on *all Attribute* tables; take the `union` of the results produced by these lookups and join with the *Value* tables
- *Universal*/separate *Value* tables: an indexed lookup by *source* on the *Universal* table; take the `union` of all *target* columns and join that with the *Value* tables
- *UnivNorm*/separate *Value* tables: combination of what is done for the *Universal* and *Attribute* approaches for the *UnivNorm* and *Overflow* tables, respectively
- *inlining*: simple (indexed) lookups on the *Edge, Attribute, Universal, UnivNorm,* and *Overflow* tables; joins with *Value* tables are not necessary.

In all mapping schemes, sorting by *ordinal* is required at the end, if this is required by the semantics of the query. If the query involves reconstructing more than one object, in addition, sorting by *source* is required so that the resulting objects can easily be re-grouped.

Selections on Values and Pattern Matching Queries that involve a selection on a value can also be translated into SQL queries in a straightforward way; as an example, consider a query that asks for the *oids* of objects that have swimming as a hobby. In the inline variants, such a query can be executed by a simple (indexed) lookup of the appropriate *value* column in the *Edge,* A_{hobby}*, Universal, UnivNorm,* or $Overflow_{\text{hobby}}$ tables. If values are stored in separate tables, such a query involves an (indexed) lookup of

the appropriate *Value* table and a join with the *Edge* table, the A_{hobby} table, the *Universal* table, or the *UnivNorm* and $Overflow_{\text{hobby}}$ tables.

If the query involves two predicates, then additional joins are necessary in the *Edge, Attribute*, and *UnivNorm* approaches. If, for instance, a user is interested in the *oids* of all objects that have swimming as a hobby and age 35, then executing this query would involve a self-join of the *Edge* table in the *Edge* approach, an $A_{\text{hobby}} \bowtie A_{\text{age}}$ join in the *Attribute* approach, and a `union` of $UnivNorm \cup (UnivNorm \bowtie Overflow_{\text{hobby}}) \cup (UnivNorm \bowtie Overflow_{\text{age}}) \cup (Overflow_{\text{hobby}} \bowtie Overflow_{\text{age}})$ in the *UnivNorm* approach. The *Universal* approach materializes the results of theses kinds of joins so that such joins are not necessary in the *Universal* approach.

Query languages like XML-QL make it also possible to pose queries that ask for a specific *pattern*. Just like the XML document itself, the `WHERE` clause of an XML-QL query can be interpreted as a graph (see Section 2.3). In general, evaluating a query in the *Edge, Attribute* and *UnivNorm* approaches involves the execution of an e-way join (in addition to possibly joins with *Value* tables), where e is the number of edges in the pattern. Evaluating a query in the *Universal* approach involves the execution of an n-way join (in addition to joins with *Value* tables), where n is the number of nodes in the pattern, and $n < e$.

Optional Predicates As a result of the irregularity of the data, XML query languages also allow to pose queries with optional predicates as in: find all objects that are 35 years old and have swimming as a hobby, if they have a hobby. Such a query can be translated for all approaches into a SQL `union` query that asks (a) for all objects that are 35 years old and have swimming as a hobby, and (b) for all objects that are 35 years old and have no hobby. (Note that no duplicate elimination is required as part of this `union`.)

Predicates on Attribute Names Another common feature of most XML query languages is the capability to support queries with predicates on attribute names. An example would be a query that asks for objects that have an *address* or *street* with value "4711 Fruitdale Ave." In the *Edge* approach, such queries can easily be translated into SQL by adding a predicate on the *Edge.name* column; e.g., `name = 'address' or name = 'street'`. In all other approaches, the translation of such a query involves an initial step in which the schema of the relational database is queried in order to find all the relevant *Attribute* and *Overflow* tables and/or the relevant fields of the *Universal* and *UnivNorm* tables. Doing so is possible because information about table and attribute names is stored in special, user-readable tables in most commercial database products.

Regular Path Expressions XML query languages also support regular path expressions that make it possible to navigate through irregular or unpredictable structures and ask for the transitive closure of objects; e.g., find all ancestors of John. Such queries can be translated into *recursive* SQL queries in a straightforward way. (If the RDBMS does not support recursive SQL queries, such queries need to be unrolled step by step.)

4.2 Updates

Updates to an XML document can also be propagated to the relational database in a straightforward way. Typically, a series of SQL insert, update, and/or delete statements are required to propagate an update. In

the following, we will briefly discuss insertions and deletions of objects and attributes.

Inserting New Objects Objects can very easily be inserted in all mapping schemes. In the *Edge* and *Attribute* approaches, one row is generated for each attribute of the new object. In the *Universal* approach, the number of new rows depends on the presence of multi-valued attributes: if the object has no multi-valued attributes, exactly one row is inserted for the new object into the *Universal* table; if the object has, say, two multi-valued attributes with cardinality three and five, then fifteen new rows are generated. In the *UnivNorm* approach exactly one row is inserted into the *UnivNorm* table and m rows are inserted into the corresponding *Overflow* table for a multi-valued attribute with cardinality m. If values are stored in separate *Value* tables, then of course, a row must be inserted into those *Value* tables for each value of the new object. If the new object contains new attribute names, then the *Universal* and *UnivNorm* tables must be extended, and new *Attribute* and *Overflow* tables must be created.

Inserting New Attributes Adding an attribute to an existing object is again simple and straightforward in the *Edge* and *Attribute* approaches and slightly more complicated in the other two schemes to map attributes. In the *Universal* and *UnivNorm* approaches, different cases must be differentiated; depending on the presence of multi-valued attributes in the object and the existence of another attribute with the same name as the new attribute, new rows must be generated or the tables must be updated. In all cases, the insertion of a new attribute may involve updating the *ordinal*'s of the attributes that belong to the same object and are located behind the new attribute in the XML document. (This procedure can be simplified in certain cases if we do not consider *ordered* semantics; see Section 2.2.) If values are stored in separate *Value* tables and the new attribute contains a value, then of course, the corresponding *Value* table must be updated. Also, if the name of the new attribute is novel, then the same measurements as for new objects with new attribute names must be made in the *Attribute, Universal*, and *UnivNorm* schemes.

Deleting an Object In all approaches, objects can easily be deleted by deleting the corresponding rows from all tables. If values are stored in separate *Value* tables, then an object must be read before it is deleted in order to find the object's values which must be deleted from the *Value* tables.

Deleting an Attribute Deleting an attribute from an object is, again, very simple to implement for the *Edge* and *Attribute* approaches and it involves looking at different cases for the *Universal* and *UnivNorm* approaches (analogous to inserts). In all approaches, deleting an attribute may involve reorganization of the *ordinal* values. Furthermore, in all approaches that use separate *Value* tables, the attribute must be read first in order to find whether the attribute contains a value which must be deleted, too.

5 Evaluating the Mapping Schemes

5.1 Plan of Attack

In order to study the tradeoffs of the alternative mapping schemes we carried out a series of performance experiments. We study, in particular, the size of the resulting relational database for each mapping scheme,

the time to bulkload the relational database given an XML document, the time to reconstruct the XML document from the relational data, the time to execute different classes of XML queries, and the time to execute different kinds of update functions.

All experiments are carried out using a synthetic XML document as a starting point. While *real* XML data is already available to some extent[4], it is unclear what the characteristics of a *typical* XML document would be. Rather than using real XML data or generating an XML document that would resemble real XML data, we use an XML document that allows us to specifically study the tradeoffs of the mapping schemes and we only model certain characteristics that can be found in real XML data (e.g., the presence of large text fields).

To simplify the discussion, we will only present experimental results for five of the eight alternative mapping schemes described in Section 3. We will study the *Edge, Attribute, Universal*, and *UnivNorm* approaches with separate *Value* tables in order to study the tradeoffs of the different ways to map attributes. In addition, we will study the *Attribute* 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[5] 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 on Solaris 2.6. In all our experiments, we limited the size of the main memory buffer of the database 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 or implementations of the update functions) is implemented in Java and runs on the same machine. Calls to the relational database from the Java programs are implemented using JDBC.

5.2 Benchmark Specification

5.2.1 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 and $0..f_v$ attributes with values. The document is flat; that is, there is no nesting of objects. (Given our XML data model described in Section 2.2, flat documents with references have the same semantics 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 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.

[4] See, e.g., `www.oasis-open.org/cover/xml.html`.

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

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

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 *Attribute* scheme with inlining. We index the strings completely, as proposed in Section 3.2, but we do not index the text (for obvious reasons), deviating from the proposed indexing scheme of Section 3.2. 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.

5.2.2 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 the appendix of this paper. 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 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. How the queries are translated into SQL queries for each mapping scheme is outlined in Section 4.1: of course, the XML-QL to SQL translation does not depend on the selectivity of the predicates, and we made sure that the translation is correct for each mapping scheme by checking the results produced by every query.

To get reproducible experimental results, we carry out all benchmark queries in the following way: every query is carried 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

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

impact the results of the heavy queries.

5.2.3 Update Functions

As part of our benchmark, we carry out four update functions which are described in Table 4. These four update functions test the insertion and deletion of whole objects and individual attributes. The new objects have the same characteristics as the other objects of our database and the new attributes contain 15-byte strings with 80% probability and 500-byte text with 20% probability. The implementation of the update functions for each mapping scheme is described in Section 4.2. The update functions do not insert XML attributes with new attribute names so that no new *Attribute* or *Overflow* tables need to be created and the *Universal* and *UnivNorm* tables need not be altered as part of executing the update functions. The database product we used for our experiments, carries out these kinds of operations very quickly; in general, however, the cost of such operations strongly depends on the implementation of the RDBMS, and we were not interested in such particularities of an RDBMS. We carry out these four update functions in sequential order and only once.

5.3 Database Size

Table 5 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, and we see that even without indices every mapping scheme produces a larger relational database. Even the *Attribute* and *UnivNorm* approaches result in more than 80 MB of base data, although they store every attribute name only once (as part of the schema) whereas every attribute name (of 10 bytes) occurs approximately 32,500 times (650,000 attributes divided by 20 different names) in the XML document. The *Edge* approach, like the XML document, stores every name multiple times and, therefore, produces more base data than the *Attribute* and *UnivNorm* schemes. Recall

U1	generate 100 new objects; commit after every new object
U2	insert 1000 new attributes into randomly selected objects; commit after every new attribute
U3	delete 1000 random attributes form random objects; commit after every attribute
U4	delete 100 random objects; commit after every object

Table 4: Update Functions

from Section 3.1 that a variant of the *Edge* approach that stores the attribute names in a separate table would result in a smaller relational database (overall reduction of about 15 MB in our benchmark), but it would also result in significantly increased query response times. The *Universal* approach, of course, produces the most base data because the *Universal* table is denormalized as described in Section 3.1. Comparing the *Attribute* 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	Attribute	Edge	Universal	UnivNorm	Attr.+Inline
base data	79.2	105.2	122.3	138.9	109.7	86.9
indices	–	71.1	85.6	76.7	49.3	52.7
total	79.2	176.3	207.9	215.6	159.0	139.6

Table 5: Database Sizes [MB]

5.4 Bulkloading Times

Table 6 shows the time it takes to prepare the XML document for bulkloading, do the actual bulkloading, and analyze the resulting tables and create indices for each mapping scheme. There are no surprises. Obviously, bulkloading takes the longest for the *Universal* approach because this approach is complicated by the denormalization of the data and because this approach produces the most relational data.

	Attribute	Edge	Universal	UnivNorm	Attr.+Inline
prepare	5m 50s	7m 1s	13m 3s	7m 40s	4m 30s
bulkload	26m 22s	26m 51s	47m 49s	23m 54s	25m 20s
analyze/indices	9m 26s	13m 43s	11m 35s	12m 39s	9m 20s
total	41m 37s	47m 35s	1h 2m 27s	44m 13s	39m 10s

Table 6: Bulkloading Times

5.5 Reconstructing the XML Document

Table 7 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. As stated in Section 4.1, 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 times for the *Universal* and *UnivNorm* approaches with separate *Value* tables can also be explained. The *Universal* and *UnivNorm* tables 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.

Attribute	Edge	Universal	UnivNorm	Attr.+Inline
56m 52s	40m 56s	1h 41m 17s	1h 50m 58s	32m 8s

Table 7: Reconstructing the XML Document

	Attribute	Edge	Universal	UnivNorm	Attr.+Inline
Q1	0.036	0.023	0.074	0.115	0.024
Q2(l)	0.104/4.6	0.089/5.3	0.093/4.8	0.139/9.6	0.011/5.3
Q2(h)	15.7	83.0	62.1	31.0	0.644/5.5
Q3(l)	6.0	5.1	5.8	100.6	2.0
Q3(h)	15.8	133.7	70.5	150.5	3.5
Q4(l)	12.3	9.9	11.7	108.0	4.1
Q4(h)	32.0	255.7	132.9	201.5	6.7
Q5(l)	0.277/15.4	5.1	14.2	28.2	0.028/13.9
Q5(h)	48.6	148.1	185.8	169.4	14.8
Q6(l)	0.130/6.5	6.1	0.141/6.3	248.0	0.017/2.0
Q6(h)	17.0	123.7	63.7	256.9	3.3
Q7(l)	0.111/6.2	0.101/5.4	0.096/6.2	49.9	0.012/5.3
Q7(h)	16.8	221.5	62.7	57.0	1.060/6.6
Q8(l)	18.3	5.0	91.4	50.4	32.7
Q8(h)	47.2	392.0	206.9	152.1	36.3

Table 8: Running Times of the Queries [secs]; Tuned/Untuned

5.6 Running Times of the Queries

Table 8 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 8 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.

In all, we can make the following two observations:

- Of all the alternative ways to map attributes, the *Attribute* approach is the winner.
- Inlining clearly 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 *Attribute* 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 [BWK98]; for XML data the benefits are particularly high. The *UnivNorm* approach is less sensitive towards the size of intermediate query results, but the *UnivNorm* approach often does the same work twice: once for the *UnivNorm* table and once for the *Overflow* tables. The *UnivNorm* approach, therefore, performs poorly in almost all cases. 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.

Q8(l) and to some extent Q1 and Q5(l) are exceptions to the above rules. These three queries involve a predicate on the attribute names (Q5 and Q8) or a wild card (Q1), and the *Edge* approach is attractive for such queries because such predicates can directly be applied to the *Edge* table whereas executing such

Function		Attribute	Edge	Universal	UnivNorm	Attr.+Inline
U1	creating objects	12.0	11.5	12.5	9.5	10.3
U2	inserting attributes	67.8	51.5	332.1	131.8	47.3
U3	deleting attributes	42.9	89.6	109.4	106.5	42.3
U4	deleting objects	28.5	8.0	25.2	44.1	14.9

Table 9: Running Times of the Update Functions [secs]

predicates involves a separate "querying the schema" step and the generation of a SQL `UNION` query for the other mapping schemes.

5.7 Running Times of the Update Functions

Table 9 shows the running times of the four update functions of our benchmark for each mapping scheme. Again, there are no surprises, and we would just like to point out the most important effects:

- The cost of creating new objects (U1) is fairly much the same for all mapping schemes.
- Inserting new attributes (U2) is significantly more expensive in the *Universal* and *UnivNorm* approaches than in the other approaches because more update and insert statements must be executed for these approaches (Section 4.2). In the *Universal* approach, this is a result of the denormalization of the data. In the *UnivNorm* approach, this is a result of the complicated organization which makes it necessary to move attributes from the *UnivNorm* to an *Overflow* table if a single-valued attribute is turned into a multi-valued attribute (e.g., if Mary gets a second hobby).
- For the same reason, the deletion of attributes (U3) is most expensive in the *Universal* and *UnivNorm* approaches.
- Deleting whole objects (U4) is cheapest in the *Edge* approach; here, only three statements must be carried out: (a) probe the *Edge* table to detect the values that must be deleted from the *Value* tables; (b) delete those values from the *Value* tables; and (c) delete the relevant rows from the *Edge* table. Deleting objects in the *Attribute* approach (with or without inlining) is more expensive because it involves separate SQL (select and delete) statements for every *Attribute* table, plus, maybe, for the *Value* tables. In the *Universal* approach without inlining, the same kind of overhead occurs: every *target* column must be probed separately in order to find the values that must be deleted. In the *UnivNorm* approach without inlining, this kind of overhead is carried out twice: (a) probe the *target* columns of the *UnivNorm* table, and (b) probe the relevant *Overflow* tables.

5.8 Experiments with Other Mapping Schemes

In all, we experimented with many different mapping schemes as part of our work, and decided to present only the most relevant results in this paper. As promised in Section 3.3, however, we do want to show what happens if we materialize the outer join of two *Attribute* tables, with and without inlining, in our particular benchmark with highly irregular data. Such a mapping scheme could, for instance, be recommended by an algorithm that extracts schema. For these experiments, we materialize $A_1 \bowtie A_2$ (instead of separate

	Separate Value Tables		Inlining	
	A_1, A_2	$A_1 \bowtie A_2$	A_1, A_2	$A_1 \bowtie A_2$
Q3(h), anticip.	16.2	15.3	3.1	3.1
Q4(h), anticip.	32.6	32.3	6.7	3.1
Q3(h), unanticip.	16.1	15.8	3.1	6.2
Q4(h), unanticip.	32.0	32.8	7.0	44.1

Table 10: Materializing the Outer Join of Two Attribute Tables [secs]

Attribute tables for A_1 and A_2) and run Q3(h) and Q4(h). Both of these queries only operate on the a_1 and a_2 attributes. So, these queries are *anticipated*, and materializing $A_1 \bowtie A_2$ should help for these queries because it saves the cost of joining the individual *Attribute* tables. We also run *unanticipated* variants of Q3(h) and Q4(h). These variants operate on the a_1 and a_3 attributes so that materializing $A_1 \bowtie A_2$ does not help because a join with A_3 is needed. Table 10 shows the results of running the anticipated and unanticipated variants of Q3(h) and Q4(h) using a table that materializes $A_1 \bowtie A_2$ with and without inlining. As a baseline, Table 10 also shows how the simple *Attribute* schemes (with and without inlining) perform for these queries (denoted as $\{A_1, A_2\}$). We can make the following observations:

- if values are stored in separate tables, $\{A_1, A_2\}$ and $A_1 \bowtie A_2$ show essentially the same performance in all cases. The cost of query processing is dominated by the joins with the *Value* tables so improvements in storing the references do not result in a noticeable speed-up.

- if values are inlined, materializing $A_1 \bowtie A_2$ helps in the anticipated case (more than a factor of two for Q4(h)) for saving the join which does dominate the cost in this case, but it looses significantly in the unanticipated case. The reason is that in the unanticipated case the join with A_3 becomes more expensive because $A_1 \bowtie A_2$ is much larger than just A_1.

6 Conclusions

We studied the performance of (standard) relational databases for the purpose of managing XML documents. Relational databases have many important advantages: One, RDBMS products are mature and scale very well. Two, traditional (structured) data and semi-structured data can co-exist in a relational database, making it possible to develop applications that use both kinds of data with virtually no extra effort. Three, our experiments showed that RDBMSs are able to process even complex XML queries on large databases within seconds; it is unlikely that the same query performance can be achieved today with any other commercially available technology. On the negative side, our experiments showed that it is very expensive to reconstruct the original XML data from the relational data and that updates are both complicated and expensive to implement in certain cases. Also, extra effort must be made to translate XML queries and updates into SQL when using a relational database; in the mapping schemes we considered in this work, this translation is both easy to implement and cheap to execute, but this translation can become very complex if more sophisticated mapping schemes are used (e.g., such as those proposed in [DFS]). Another potential disadvantage, which we did not address in this work, is that components such as authorization and concurrency control need to be implemented outside of the RDBMS because the corresponding built-in components of the RDBMS do not

work properly since they require fully structured data. The result is that the same kind of functionality is implemented (and possibly executed twice) and a great deal of the functionality implemented in the RDBMS is not used.

We also studied alternative ways to store XML data in a relational database. Our experimental results clearly show that an approach which is based on separate *Attribute* tables for every attribute name that occurs in an XML document and *inlining* of values into these *Attribute* tables is the best overall approach. Even large values (e.g., text) can be inlined without significantly hurting the performance of this approach. Ideally, of course, it is desirable to make flexible decisions about what kind of data to inline and to keep information of certain attributes in a single table, using statistics of the XML document and knowledge of the anticipated query workload. Our results, however, are encouraging because they show that robust performance can be achieved with a much simpler approach. Also, our results show that sometimes a more sophisticated approach can hurt more than it helps.

There are several avenues for future work. Our results are only one initial step in order to answer the *big* overall question whether to use relational systems, object-oriented systems, or special-purpose data stores to manage XML documents. So, we plan to run our benchmark on an OODBMSs and on special-purpose XML data repositories as well. Furthermore, our benchmark can be used to decide which is the best relational database product for the specific purpose of storing XML documents, and we are going to put all the details and the software of our benchmark on the web. In addition, our results encourage further work on the efficient implementation of binary data models [CK85, BWK98].

References

[Abi97] Serge Abiteboul. Querying semi-structured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[BDFS97] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. Adding structure to unstructured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 505–516, Montreal, Canada, 1996.

[Bos99] Adam Bosworth. XML and semi-structured data. *Workshop on Query processing for semistructured data and non-standard data formats*, 1999.

[Bun97] Peter Buneman. Semistructured data. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 117–121, Tucson, Arizona, 1997.

[BWK98] P. Boncz, A. Wilschut, and M. Kersten. Flattening an object algebra to provide performance. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 568–577, Orlando, FL, 1998.

[CACS94] Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl. From structured documents to novel query facilities. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 1994.

[CCD+99] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: a graphical language for querying and restructuring XML documents. In *Proc. of the Int. WWW Conf.*, 1999.

[CK85] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 268–279, Austin, TX, May 1985.

[DFF+99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: a query language for XML. In *Proc. of the Int. WWW Conf.*, 1999.

[DFS] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. To appear in *Proc. of ACM SIGMOD Conf. on Management of Data*, Philadelphia, PN, 1999.

[FFK+98] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.

[FFLS97] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

[FLS97] Daniela Florescu, Alon Levy, and Dan Suciu. A query optimization algorithm for semistructured data. Technical report, Inria, 1997.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[KM92] A. Kemper and G. Moerkotte. Access Support Relations: an indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

[KSW95] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Information Systems*, 20(1):21–51, 1995.

[Lot98] http://www.lotusnotes.com/, 1998.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[Moh99] C. Mohan. Lotus Domino/Notes: First semi-structured DBMS of the world. *Workshop on Query processing for semistructured adata and non-standard data formats*, 1999.

[MW97] J. McHugh and J. Widom. Query optimization for semistructured data. Technical Report, Stanford University, November 1997.

[MWA+98] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical Report, Stanford University, January 1998.

[NAM98] Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting schema from semistructured data. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 251–260, Taipei, Taiwan, 1995.

[QL'98] *Proceedings of the WWW Workshop on Query Languages for XML*, Boston, MA, December 1998. http://www.w3.org/TandS/QL/QL98/cfp.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML query language (XQL), 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[SD96] R. Sacks-Davis. The structured information manager: A database system for SGML document. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[SLS+93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The Rufus system: Information organization for semi-structured data. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 97–107, Dublin, Ireland, 1993.

[Suc98] Dan Suciu. Semistructured data and XML. *FODO'98*, 1998.

[TS97] D. Theodoratos and T. Sellis. Data warehous configuration. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 126–135, Athens, Greece, August 1997.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes I, II.* Computer Science Press, Rockville MD, 1989.

[vZAW99] 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.

[YIU96] M. Yoshikawa, O. Ichikawa, and S. Uemura. Amalgamating SGML documents and databases. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, pages 259–274, Avignon, France, March 1996.

A The benchmark queries in XML-QL

Query 1 *: Retrieve all the attributes of XML object o_1*

```
select $l, $v
where   < _ >
            < id > o1 < /id >
            < $l > $v < /$l >
        < /_ >
```

Query 2 *: Find objects that have an attribute a_1 with a value in a certain range*

```
select $oid, $v
where   < _ >
            < id > $oid < /id >
            < a1 > $v < /a1 >
        < /_ >, $v between c1 and c2
```

Query 3 *: Find objects that have attribute a_1 and a_2 with values in certain ranges*

```
select $oid. $v1, $v2
where   < _ >
            < id > $oid < /id >
            < a1 > $v1 < /a1 >
            < a2 > $v2 < /a2 >
        < /_ >, $v1 between c1 and c2, $v2 between c3 and c4
```

Query 4 *: Find objects that have attribute a_1 and a_2 with values in certain ranges, or just a_1 with a value in a certain range*

```
select $oid, $v1, $v2
where   < _ >
            < id > $oid < /id >
            < a1 > v1 < /a1 >
            [ < a2 > v2 < /a2 >, $v2 between c3 and c4]
        < /_ >, $v1 between c1 and c2
```

Query 5 *: Find objects that have attribute a_1 or a_2 or a_3 with a value in a certain range (regular expression with disjunction)*

```
select $oid, $v
where   < _ >
            < id > $oid < /id >
            < a1 | a2 | a3 > v < / >
        < /_ >, $v between c1 and c2
```

Query 6 *: Find objects that match a complex pattern with seven references and eight nodes*

```
select $oid, $v1, $v2, $v3, $v4, $v5
where   < _ >
            < id > $oid < /id >
            < a1 > $v1 < /a1 >
            < a2 >
                < a4 > $v2 < /a4 >
                < a5 > $v3 < /a5 >
            < /a2 >
            < a3 >
                < a6 > $v4 < /a6 >
                < a7 > $v5 < /a7 >
            < /a3 >
        < /_ >, $v1 between c1 and c2
```

Query 7 *: Find all objects that are connected by a chain of a_1 references to an object with an a_1 value in a certain range (Kleene star)*

```
select $oid, $v
where    < _ >
             < id > $oid < /id >
             < (a1)* > $v < / >
         < /_ >, $v between c1 and c2
```

Query 8 *: Find all objects that are connected by a chain of a_1 or a_2 references to an object with an a_1 or a_2 value in a certain range (complex regular expression)*

```
select $oid, $v
where    < _ >
             < id > $oid < /id >
             < (a1 | a2)* > $v < / >
         < /_ >, $v between c1 and c2
```