The Design and Performance Evaluation of Alternative XML Storage Strategies

Feng Tian David J. DeWitt Jianjun Chen Chun Zhang
Department of Computer Science
University of Wisconsin, Madison
{ftian, dewitt, jchen, czhang}@cs.wisc.edu

Abstract

This paper studies five strategies for storing XML documents including one that leaves documents in the file system, three that use a relational database system, and one that uses an object manager. We implement and evaluate each approach using a number of XQuery queries. A number of interesting insights are gained from these experiments and a summary of the advantages and disadvantages of the approaches is presented.

1. Introduction

XML is the new standard for Internet data representation and exchange. An important question is what is the best way of storing XML documents since the performance of the underlying storage representation has a significant impact on query processing efficiency. Several projects [1][9][10][16] have proposed alternative strategies for storing XML documents. These strategies can be classified according to the underlying system used: file system, database system, or object manager. To the best of our knowledge there has been no careful performance study comparing these alternatives and it is still an open question which of the strategies is the best.

We briefly describe these alternatives. One way is to store each XML document in a text file. The main advantage of this approach is that it is easy to implement and does not require the use of a database system or storage manager. It has several significant disadvantages, however. First, XML documents need to be parsed every time they are accessed. Second, the entire parsed file must be memory-resident during query processing. These problems can be solved by building external indices on XML documents. A query engine can use these indices to retrieve document segments relevant to a query. This type of index usually stores offsets of XML elements in the text file to help retrieve partial documents. Consequently, the indices are difficult to maintain if the XML document is updated.

An alternative is to store XML documents in a database system. Several recent papers [9][10][16] have examined how to map and store XML data in a relational database system. The disadvantage of this approach is that current database system may not be well tuned for XML workload and accessing XML data through an interface such as SQL incurs overhead not related to storage.

The third alternative is to use an object manager such as Shore [4]. While this approach allows special purpose processing, an object manager requires more work to use than a full-blown database system.

This paper studies five alternative ways of storing XML documents: one that employs text files stored in the file system, three that use a relational database system, and one that uses an object manager. We omit the approach of using an object-oriented database mainly because the underlying storage structure of an OODBMS is not fundamentally different from that of an object manager.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the different strategies for storing XML. The performance of these strategies is evaluated in Section 4. We conclude in Section 5.

2. Related Work

Recently, several projects have investigated strategies for storing and XML data to facilitate efficient query processing. Abiteboul et al. examine the use of a text file [1]. In [10], Kanne and Moerkotte store each XML file as a collection of records in an object manager and evaluate alternative strategies for grouping XML elements into page-sized records. Lore [11] is a special purpose database system that exploits features of the semi-structured data model. Another approach is to store XML data in a relational DBMS or OODBMS [7][8][9][15][16]. [16] examined how to map XML data into a relational database given the DTD of the file. This study used the number of join operations performed as its performance metric and not response times for running real queries against XML datasets. The STORED[7] system utilizes data mining to extract a schema from XML data and converts them to relations. In [9], Florescu and Kossman evaluated several alternative mappings for storing XML documents in a relational database system without using DTD. Our work extends [9] and [16] by comparing them with a few other strategies and evaluating them using extensive experiments.

All major relational database vendors now offer some form of XML support [6][12][14]. These commercial

tools are all conceptually similar to the relational DTD approach that we evaluate in this paper. Two object-oriented database systems, Excelon [8] and POET [15], map each XML element into a separate object. Their approaches are similar to the Object approach described in Section 3.

3. Different Storage Strategies

We use the XML document "Dept.xml" in Figure 3.1 to illustrate how XML data is actually stored with each strategies. An XML document can be modeled as a directed graph, with nodes in the graph representing XML elements or attributes and edges representing parent-children relationships. Such a graph is shown in Figure 3.2. Boxes with rounded corners represent attribute or text nodes.

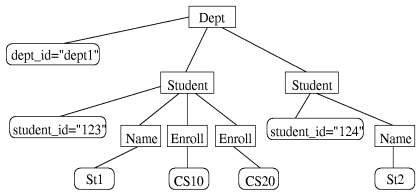

Figure 3.2 The graph representation of "Dept.xml"

3.1 The Text Approach

The first strategy stores each XML document as a text file. One way to implement a query engine with this approach is to parse the XML file into a memory-resident tree against which the query is then executed. The tree is retained in memory as long as some nodes in the tree are needed for query evaluation. We found that the parsing time dominated query execution time and the approach was unacceptably slow. To make this approach competitive we adopted the following indexing strategy. Using the offset of an XML element inside the text file as its id, we build a path index mapping (*parent_offset, tag*) to *child_offset* and an inverse path index mapping *child_offset* to *parent_offset*. These two indices are used to facilitate navigation through the XML graph. Another index mapping (*tagname, value*) or (*attribute_name, attribute_value*) to element offset is built to help evaluate selection predicates. A query engine can use these indices to retrieve segments of an XML file relevant to the query, reducing parsing time dramatically.

3.2 The Relational DTD approach

The second strategy is the shared-inlining method proposed in [16] and requires the existence of a DTD. A separate table is used to capture the set-containment relationship between an element and a set of children elements with the same tag. Each tuple in a table is assigned an *ID* and contains a *parentID* column to identify its parent. An element can appear only once in its parent is inlined as a column of the table representing its parent. If the DTD graph contains a cycle, a separate table must be used to break the cycle. The relational schema generated from the Dept DTD and how the document is stored are shown below.

ParentID	ID	Dept_id
1	2	"dept1"

Table 3.1 The Dept table

ParentID	ID	TEXT
3	5	"CS10"
3	6	"CS20"

Table 3.2 The Enroll table

ParentID	ID	Student_id	Name
2	3	"123"	"St1"
2	4	"124"	"St2"

Table 3.3 The Student table

3.3 The Edge Approach

The third strategy is the "Edge" approach described in [9]. The directed graph of an XML file is stored in a single Edge table. Each node in the directed graph is assigned an id in the dept first order. Each tuple in the Edge table corresponds to one edge in the directed graph and contains the ids of the two nodes connected by the edge, the tag of the target node, and an ordinal number that is used to encode the order of children nodes. When an element has only one text child, the text is inlined.

SourceID	tag	ordinal	TargetID	Data
1	Dept	1	2	NULL
2	dept_id	0	0	"dept1"
2	Student	1	3	NULL
2	Student	2	4	NULL
3	student_id	0	0	"123"
3	Name	1	0	"St1"
3	Enroll	2	0	"CS10"
3	Enroll	3	0	"CS20"
4	Student_id	0	0	"124"
4	Name	1	0	"St2"

Table 3.4 The Edge Table

Table 3.4 contains the Edge table for the example shown in Figure 3.1. *TargetID* 0 indicates that the edge points to a TEXT node or ATTRIBUTE node. 0 in *ordinal* field indicates an attribute edge.

As suggested in [9], an index is built on *(tag, data)* in order to reduce the execution time of selection queries. We found that it was also very important to build indices on *(sourceId, ordinal)* and *(targetID)*. The former is used to lookup children elements of a given element and the later is used when traversing from a child node to its parent.

```
<?xml?>
<!ELEMENT Dept (Student*)>
<!ATTLIST Dept dept_id ID #REQUIRED>
<!ELEMETN Student (Name, Enroll*)>
<!ATTLIST Student student_id ID #REQUIRED>
<!ELEMENT Name #PCDATA>
<!ELEMENT Enroll #PCDATA>
```

```
<?xml version="1.0"?>
<!DOCTYPE Dept SYSTEM "Dept.dtd">
<Dept dept_id="dept1">
<Student student_id="123">
<Name>St1</Name><Enroll>CS10</Enroll><Enroll>CS20</Enroll></Student>
<Student student_id="124"><Name>St2</Name></Student>
</Dept>
```

Figure 3.1 Sample XML file "Dept.xml" and its DTD

The clustering strategy on the Edge table has significant impacts on query performance. While we clustered the Edge table on the *Tag* field, an alternative strategy is to cluster the table according to *SourceID*. This strategy has the benefit that sub-elements of one XML element are stored close to each other. The drawback is that elements with the same tag name are not clustered. Consequently, queries such as "select all students whose major is Computer Science" will incur a large number of random I/Os. Our experiments showed clustering on the *Tag* attribute has better performance, except when reconstructing the original XML file. Thus, we only consider clustering on the *Tag* attribute in this paper.

3.4 The Attribute Approach

Florescu and Kossman [9] suggested another approach called the "Attribute" approach. The Attribute approach is a **horizontal partition** of the Edge approach by the *Tag* field. Tuples with different tags are stored in separate tables. While one might argue that the Attribute approach saves space by not storing the tag field, it sacrifices a very important property of Edge approach. With the Attribute approach, a query processor needs a DTD to decide which table contains sub-elements since the tags of the sub-elements are not recorded in the table. Furthermore, for a large collection of XML documents, the attribute approach can result in a large number of tables.

3.5 The Object Approach

An obvious way of storing XML documents in an object manager is to store each XML element as a separate object. However, since XML elements are usually quite small, we found the space overhead of this strategy prohibitive. Instead, all the elements of an XML document are stored in a single object with the XML elements becoming light-weight objects inside the object. We use the term ***lw_object*** to refer to the light-weight object and ***file_object*** to denote the object corresponding to the entire XML document.

Offset	Record
0	Length=40, **Dept**, parent=nil, prev=nil, next=nil first_child=40, last_child=140, Attr(dept_id="dept1")
40	Length=40, **Student**, parent=0, prev=nil, next=140, first_child=80, last_child=120, Attr(student_id="123")
80	Length=20, **Name**, parent=40, prev=nil, next=100, no children, no attribute, #PCDATA="St1"
100	Length=20, **Enroll**, parent=40, prev=80, next=120, no children, no attribute,#PCDATA="CS10"
120	Length=20, **Enroll**, parent=40, prev=100, next=nil, no children, no attribute, #PCDATA="CS20"
140	Length=40, **Student**, parent=0, prev=40, next=nil, first_child=180, last_child=180,Attr(student_id="124")
180	Length=20, **Name**, parent=140, prev=nil, next=nil, no children, no attribute, #PCDATA="St2"

Figure 3.3 File object holding "Dept.xml"

Figure 3.3 shows how the example XML file is stored in a *file_object*. The format of each *lw_object* is shown below:

length	flag	tag	parent	prev	next	opt_child	opt_attr	opt_text

The offset of the *lw_object* inside a *file_object* is used as its identifier *(lw_oid*), as shown at the upper left corner of each *lw_object* in Figure 3.3. The *length* field records the total length of the *lw_object.* The *flag* field contains bits that indicate whether this *lw_object* has *opt_child, opt_attr*, or *opt_text* fields. The *tag* field is the tag name of the XML element. The *parent* field records the *lw_oid* of the parent node. *Opt_child* records the *lw_oids* of the first and last child, if the *lw_object* has children. The sibling list of a node is implemented as doubly linked list via the *prev* and *next* fields. *Opt_attr* records the (name, value) pair of each attribute of the XML element. Text data is in-lined in the *opt_text* field if the text is the only child of the XML element; otherwise, the text data is treated as a separate *lw_object*. We build a B-Tree index that maps (*tag, opt_text*) and (*attr_name, attr_value*) to *lw_oid*. An element is entered in this index even if the *opt_text* field is empty so that this index can be used to retrieve all XML elements with a specific tag name. We also build a path index that maps (*parent_id, tag*) to child *lw_oid*.

4. Performance Study

This section evaluates the performance of the five strategies described in Section 3 on two different datasets. The first dataset models a university department database like that described in [5]. It contains 250 XML files, 114MB in total. Figure 4.1 presents an overall picture of the DTD for the dataset. The arrows indicate element containment relationships. Strong lines with a "*" indicate that there may be multiple sub-element occurrences.

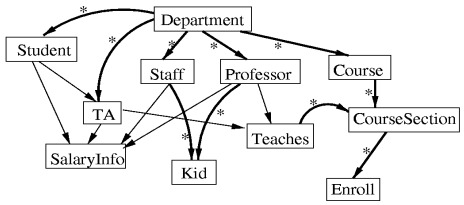

Figure 4.1 DTD graph of Department dataset

The second dataset we used is the Open Directory Project dataset [13], which contains a comprehensive directory of the web. The size of the ODP data set we used is about 140 MB. Web pages are organized into topics and each topic may contain nested sub topics. This hierarchical information is captured by cycles in DTD graph shown in Figure 4.2. A *Topic* element can have several other *Topic* elements as its children. This

cycle in DTD graph will require that a path expression query be translated into a fixed-point evaluation.

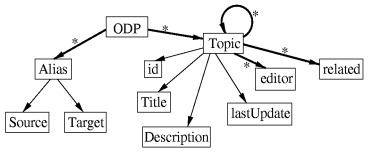

Figure 4.2 DTD graph of the ODP dataset

Table 4.1 lists the indices used with each approach. Table 4.2 summarizes the space consumed by each strategy.

	Indices
TEXT	path index, inverted path index, (tag,data) or (attrname, attrvalue) to element_offset
DTD	Indices on each column containing XML data value. Indices on parentId and myId
Edge	(tag, data), (sourceId, ordinal), (targetId)
ATTR	(sourceId), (targetId), (data)
Object	(tag, data), (attr_name, attr_value), path index

Table 4.1 Indices of each approach

		TEXT	**DTD**	**Edge**	**ATTR**	**Object**
Department Dataset	**Data**	114	69.7	223	165	104
	Indices	206	29.3	167	130	164
ODP Dataset	**Data**	145	126	222	187	160
	Indices	212	132	190	181	192

Table 4.2 Space usage of each approach (in MB)

Our experiments were conducted using an 800 MHz Pentium III with 256 MB memory running Linux 2.2. We used DB2 V7.1 as the relational DBMS. The Object strategy was implemented using Shore [4]. Both DB2 and Shore were configured to use a 30MB memory buffer pool. There was no buffer pool for the TEXT approach and the query processor used as much physical memory as available (256M). The indices for the TEXT approach were implemented using Berkley DB [2]. For the DTD and Edge approaches, XQuery queries were manually translated to SQL queries to be executed by DB2.

We conducted extensive experiments to compare the strategies. The results presented in this section were obtained with cold buffer pools. More results can be found in [17].

4.1 Reconstruct Original XML Documents

This experiment measures the time to reconstruct documents in the original datasets. There is no reconstruct time for the TEXT approach since the original XML files were stored in the file system.

	DTD	**Edge**	**Attribute**	**Object**
Department Dataset	1404	2011	3100	78
ODP Dataset	1184	1833	2856	81

Table 4.3 Reconstruction time (sec)

DTD and Edge approach clustered elements according to tag names. Hence, the order of tuples in the tables no longer reflects the original order of elements in XML documents and reconstruction incurs many random I/Os. In the Edge approach, one SQL is used to retrieve element id of all sub-elements. For the Attribute approach, DTD information is required to decide which tables that may contain sub-elements. The number of SQL queries needed to find all sub-elements equals the number of *possible* tags.

4.2 Selection Queries

Our second set of experiments measures the performance of different types of selection queries.

Selection Query 1: Index look up

Index look up on Department data

SQ_1A: Find Staff name whose id is 'P_77'
FOR $s in document()/department/Staff WHERE $s/@id='P_77' RETURN <result> $s/name </result>

	DTD	**Edge**	**Attribute**	**Object**	**TEXT**
SQ_1A	0.4	0.5	0.5	0.21	0.3

Table 4.4 SQ_1A (time in seconds)

There is only one *Staff* that satisfies the predicate in SQ_1A. The relational database based approaches have worse performance than object manager and text based strategies due to the overhead of relational query engine.

Index scan on ODP data

SQ_1B: select Topic description with title "Photography".
FOR $t in document()//topic WHERE $t/Title='Photography' RETURN $t/Descripton
SQ_1B*: select Topic description which has a sub-topic with Title "Photography"
FOR $t in document()//topic WHERE $t//Title = 'Photography' RETURN $t/Description

	DTD	**Edge**	**Attribute**	**Object**	**TEXT**
SQ_1B	0.8	1.2	2.3	9.4	6.7
SQ_1B*	2.4	10.7	7.4	9.6	7.3

Table 4.5 SQ_1B and SQ_1B* (time in seconds)

For SQ_1B, objects in Object and TEXT approaches were clustered according to the document order. After the index look up using *Title='Photography'*, chasing child/parent links incurred lots of random I/O. The relational approaches performed much better because tuples were clustered according to tag names. For SQ_1B*, the cycle in DTD graph required a fixed-point evaluation with relational approaches, thus their running times were much worse than SQ_1B.

Selection Query 2: Scan Selection

Scan Selection on Department data

SQ_2A: Select professor id, name with salaries higher than $60,000
FOR $p in document()/department/professor WHERE salary($p) > 60000 RETURN $p.id, $p.name

The Salary of an employee of the department is computed by the ***salary()*** function using the *SalaryInfo* sub-element of *Professor*.

	DTD	Edge	Attribute	Object	TEXT
SQ_2A	1.97	18.4	13.2	25	29

Table 4.6 SQ_2A (time in seconds)

Clustering the same type of elements together (e.g. all *Professor*s) is important for this query. The DTD approach has the best performance because it also inlines *SalaryInfo* and personal information like *id* and *Name* with *Professor* elements, while the Edge and Attribute approaches need to perform joins to retrieve those values. The TEXT approach has essentially the same access pattern as the Object approach, except we need to parse the professor elements to retrieve *SalaryInfo, id* and *name*.

Scan selection on ODP data

SQ_2B: find topics that are updated in last quarter of a year.
FOR $t in document()//topic WHERE month($t/lastupdate) >= 10 RETURN $t/Description
SQ_2B*: find topics that contain a sub-topic which is updated in last quarter of a year.
FOR $t in document()//topic WHERE month($t//lastupdate) >= 10 RETURN $t/Description

	DTD	Edge	Attribute	Object	TEXT
SQ_2B	5.1	11.8	4.5	45	31
SQ_2B*	83	80	72	47	41

Table 4.7 SQ_2B and SQ_2B* (time in seconds)

Comparing results of SQ_2B with those of SQ_2B*, the performance of relational approaches dropped from the best to the worst. This is because SQ_2B* requires recursive SQL query processing.

4.3 Set Containment Queries

Set containment queries on Department data

CQ_1: Select ids and names of professors who have a kid named "girl16"
FOR $p in document()/department/professor WHERE $p/kid="girl16" RETURN $p/id, $p/name

	DTD	Edge	Attribute	Object	TEXT
CQ_1	1.2	27.1	9	5.6	21

Table 4.8 CQ_1 (time in seconds)

Containment queries for ODP data

CQ_2: Find sub-topic of Topic 10366
FOR $t in document()//Topic WHERE $t/@catid='10366' RETURN $//Topic/Description

	DTD	Edge	Attribute	Object	TEXT
CQ_2	1.8	2.5	2.8	1	1.4

Table 4.9 CQ_2 (time in seconds)

The DTD approach exhibits good performance for both queries because similar elements are clustered together. The information that is needed to construct the result of the query is readily available as columns of the relational tables. The Edge and Attribute approaches suffer from the cost of constructing query results as tuple corresponding to a single real world object (eg. *id* and *name*) are scattered around the tables. Since the Object approach cluster elements in the original order of the document, the I/O (sequential) needed to retrieve *Description* by CQ_2 is confined in one *Topic* element. CQ_1 requires navigating from children (*Kid* is girl16) to parent nodes (*Professor*). Traversing upward is more likely to incur random I/O. While the parent node id is stored as a field of children nodes in the Object approach, the TEXT approach must use the inverse path index to look up the parent id, therefore the performance suffers.

4.4 Join Queries

Join query on Department data

JQ_1: Find students with same birthdate and zipcode.
FOR $s1 in document()/department/student RETURN <result> FOR $s2 in document()/department/student WHERE $1/birthdate = $s2/birthdate and $s1/zipcode = $s2/zipcode and $s1/@id != $s2/@id RETURN $s1/@id, $s1/name, $s2/@id, $s2/name </result>

	DTD	Edge	Attribute	Object	TEXT
JQ_1	3.4	35	31	30	35

Table 4.10 JQ_1 (time in seconds)

JQ_1 can be directly translated into a self-join query on the Student table with the DTD approach. For the Object and Text approaches, we implemented a hash join and assumed that the hash table fits in memory. The reason that DTD approach significantly out-performed the Object approach is that all student information is clustered in one table, whereas for the Object approach, the student information is scattered in different departments.

Join on ODP data

JQ_2: Retrieve descpritions for same subtopic of Illinois and Wisconsin

FOR $it in document()//Topic[@id='Illinois']//Topic
RETRUN FOR $wt in document()//Topic[@id='Wisconsin']//Topic
WHERE $it/Title = $wt/Title
RETURN $it/Description, $wt/Description

	DTD	Edge	Attribute	Object	TEXT
JQ_2	1.5	17	15	1	1

Table 4.11 JQ_2 (time in seconds)

JQ_2 consists of fixed-point evaluation of both sides of the join operator. The cost of evaluating the recursive query with Edge and Attribute approaches is high. We examined the execution plan and found the execution plan is sub-optimal because it is hard to estimate the size of the output of fixed-point evaluation.

4.5 Summary

Our experiments demonstrated that there are three forms of desirable clustering when storing XML files.

1. Clustering elements corresponding to the same real world object. For example, storing a student's id and name together.
2. Clustering the same kind of elements together. For example, storing all student elements together.
3. Clustering elements using the same order as in the original text XML files

The Relational-DTD approach uses strategies 1 and 2 aggressively. DTD information helps to produce much more compact data representation. The drawback of this approach is that it cannot handle XML documents without DTD. Fortunately, in many XML application such as E-business information exchange, well agreed upon DTDs have begun to appear. Using a relational database system has several other advantages including portability and scalability. In addition, since a significant fraction of the data on the web currently resides in relational database systems, using a relational DBMS to store XML documents makes it possible to query both types of data with one system and one query language.

Both Edge approach and Attribute approach exploit clustering strategy 2. Unfortunately, the benefits of clustering strategy 1 are lost. This results in much worse performance when the query must apply predicates related to several sub-elements and when constructing result documents. The parent-children relationship between XML elements are captured by SQL joins. This produces very complex SQL queries involving tens of joins for complex path expressions that make it difficult for the relational database query optimizer to produce a correct plan. The number of joins also makes these approaches sensitive to complexity of path expression. The Attribute approach has more compact data representation than Edge approach. On the other hand, Attribute approach needs DTD information in order to reconstruct an element. The reconstruction cost is higher due to more SQL queries needed to fetch all sub-elements.

The Object approach uses clustering strategy 3. Since elements corresponding to one real world object are frequently clustered together in the original XML document, strategy 3 shares some of the benefits of strategy 1. While strategy 3 provides very good performance when reconstructing query results, the fact that similar objects (elements with same tag name) are not clustered adds significant overhead to query processing when compared with the DTD approach.

5. Conclusion

This paper explores several different strategies for storing XML documents: in the file system, in a relational database system and in an object manager. We evaluated the performance of each strategy using a set of queries. Our results clearly indicate that DTD information is vital to achieve good performance and compact data representation. When DTD is available, the DTD approach has compact data representation and excellent performance across different datasets and different queries.

On the other hand, there are applications that need to handle XML files without DTDs or XML files used as a Markup Language. When DTD has cycles, a path express in XQuery will be translated into recursive SQL queries. Our results showed object storage manager based approaches can out perform relational approach on fixed-point evaluation.

With proper indices, the TEXT approach can achieve similar performance to the Object manager based approach. However, the cost of maintaining indices will make this approach only useful when update frequency is low.

References

[1] S. Abiteboul, S. Cluet, et al. *Querying and updating the file*. VLDB 1993

[2] *Berkley DB toolkit*. http://www.sleepycat.com

[3] P. Buneman, *Semi-structured dat*a, PODS 1997

[4] M. Carey, D. DeWitt, et al. *Shoring Up Persistent Application*s, SIGMOD 1994

[5] M. Carey, D. DeWitt, et al. *The BUCKY Object-Relational Benchmark,* SIGMOD 1997

[6] IBM DB2 XML Extender. http://www4.ibm.com/software/data/db2/extenders/

[7] A. Deutsch, M. F. Fernandez, et al. *Storing Semi-structured Data with STORED*, SIGMOD 1999

[8] Excelon. http://www.odi.com/excelon

[9] D. Florescu, D. Kossman, *Storing and Querying XML Data using an RDMBS.* IEEE Data Engineering Bulletin 22(3), 1999

[10] C. Kanne, G. Moerkotte, *Efficient storage of XML dat*a, ICDE 2000

[11] J. McHugh, S. Abiteboul, R. Goldman, et al. *Lore: A Database Management System for Semi-structured Data,* SIGMOD Record 26(3) (1997)

[12] Microsoft SQL Server 2000 Books Online, XML and Internet support.

[13] Open Directory Project. http://www.dmoz.org/.

[14] Oracle XML SQL Utilities. http://otn.oracle.com/tech/mxl/oracle_xsu/.

[15] POET, http://www.poet.com/.

[16] J. Shanmugasundaram, K. Tufte, et al. *Relational Databases for Querying XML Documents: Limitations and Opportunitie*s. VLDB 1999.

[17] http://www.cs.wisc.edu/~ftian/paper/xmlstore.pdf.