Contact Author:

Feng Tian
Department of Computer Science
1210 W. Dayton
Madison, WI, 53706
Phone: (608) 2626622
Email:ftian@cs.wisc.edu

Paper ID 162

Title: The Design and Performance Evaluation of Alternative XML Storage Strategies

Topic Area: Core Database Technology

Category: Research

Authors:

Feng Tian
David J. DeWitt
Jianjun Chen
Chun Zhang

Topics:

- **Databases and database services in new context - Internet and the WWW**
- **Semi-structured data**
- **Database benchmark and measurement**

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

XML is an emerging Internet standard for data representation and exchange. When used in conjunction with a DTD (Document Type Definition) XML permits the execution of a rich collection of queries using a query language such as XML-QL or Quilt. This paper describes six strategies for storing XML documents including one that leaves documents in the file system, three that use a relational database system, and two that use an object manager. Each approach was implemented and evaluated using a number of different Quilt queries. A number of interesting insights were gained from these experiments and a summary of the advantages and disadvantages of each of the six approaches is presented.

Topic Area: Core database technology
Category: Research

1. Introduction

The Extensible Markup Language (XML) [BPS98] is an emerging standard for Internet data representation and exchange. In the near future it is expected that XML (in combination with XSL) will replace HTML as the dominant file format for web-resident data. When compared with other mark-up languages such as HTML the main advantage of XML is that each XML document can have a Document Type Definition (DTD) associated with it. A DTD serves as an implicit semantic schema for the XML document and makes it possible to define much more powerful queries than what is possible with simple, keyword-based text retrievals. In many ways XML documents and DTDs closely resemble the semi-structured data model that has been actively studied in recent years by the database research community. Overall, XML can serve at least two roles. First, as a new markup language, a web browser can browse an XML file in the same way as an HTML file. Second, and more interesting to the database community, XML can serve as a standard way of storing semi-structured data sets. XML makes it possible for users to ask very powerful queries against the web. For example, doing a keyword search of "car", "price" and "safety" will probably return millions of documents. Posing the query such as "find the top 10 safest cars

that are priced below $25,000" using an XML-based query language such as XML-QL [DFF+99] or Quilt [CRF00] makes it more likely that the user will get the answer he/she really wants.

There have been a number of research projects on semi-structured query languages and data models [Abi97] [AQM+97] [MAG+97]. Another 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 [ACM93][FK99][STZ+99][KM00] 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 three alternatives. One common, and obvious, solution is to store each XML document in a text file. This is the standard approach today. When an XML query is evaluated against the document, the document is read and parsed into a tree such as a DOM tree as the first step in preparation for query evaluation. 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. Another advantage is there is no loading or reconstructing cost when the original file is needed. It does have several significant disadvantages, however. First, XML documents stored as text files need to be parsed every time they are accessed for either browsing or querying. Second, the entire parsed file must be memory-resident during query processing. These problems can be solved by building external indices on XML documents stored as text files. A query engine can then use these indices to retrieve document segments related to a query. This type of index usually stores offsets of XML elements inside the text file to help retrieve partial documents. Consequently, the indices will be difficult to maintain if the text XML file is ever updated.

An alternative is to store XML documents in a database system. Several recent papers [DFS99][FK99][STZ+99] have examined how to map and store XML data in a relational database

system. One version of Lore [MAG+97] explored the use of O2 [BBB+88], an object-oriented database system (OODBMS) as its underlying storage system.

Since storing and accessing XML data through a SQL interface incurs overhead not related to storage, a third alternative is to use an object manager such as Shore [CDN+94]. While this approach is expected to have a lower overhead than a relational database system, the object-level interface provided by such systems requires more work to use than a SQL or OQL based-interface for many operations (e.g. when iterating through a set of elements applying a predicate). In addition, as a standard, SQL offers a degree of portability not found among different object managers.

Yet another alternative would be to develop a special storage format and supporting software for storing XML data using raw disk. Although such an approach might provide better performance, we doubt whether it is commercially viable given the maturity of existing file systems and database systems. In addition, it is highly unlikely that the fundamental architecture of such a system would be significantly different from the object managers used to implement either relational or object-oriented database systems. Thus, we omit this approach from our paper.

This paper describes six alternative ways of storing XML documents: one that employs text files stored in the file system, three that use a relational database system, and two that use an object manager. These alternatives are evaluated using different queries representing both navigational and associative query workloads. A navigational workload can be generated from requests to an XML server from either a web browser or database query engine. It can also be generated from database queries since XML data is usually modeled as a labeled graph and queries on XML documents generally involve navigation on the graph. On the other hand, many database queries involve selection predicates that test the contents of an element for a particular value or range of values. An index is indispensable for executing this type of query. With an index, the query can directly access the relevant elements without having to repeatedly traverse the tree from document's root node. Both types of workloads need to be supported efficiently.

The remainder of this paper is organized as follows. Section 2 discusses related work. In Section 3 we describe six different strategies for storing XML. The performance of these strategies is evaluated in Section 4. Our conclusions are contained in Section 5.

2. Related Work

Semi-structured query languages and data models have been widely studied, for example, in [Abi97][AQM+97][Bun97][MAG+97]. Recently, several projects have investigated strategies for storing semi-structured and XML data sets to facilitate efficient query processing. [ACM93] examines the use of a text file. [KM00] stores each XML file as a collection of records in a storage manager and evaluates alternative strategies for grouping XML elements into page-sized records. This approach is very similar to our Object approach except that we use one object per XML file (our objects automatically grow into large objects when they become bigger than one database page). [MAG+97] described a special purpose database system that exploits special features of the semi-structured data model. Another approach is to store XML data in a relational DBMS or OODBMS [DFS99][STZ+99][FK99]. [STZ+99] 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 real XML data sets. While [STZ+99] automatically extracts a relational schema given the DTD of the XML document, [FK99] describes several storage strategies that use a relational database system which do not require the existence of a DTD. One implementation of LORE [MAG+97] explored the use of O2 as an underlying storage system but did not evaluate its performance to that of the standard LORE storage manager.

All major relational database vendors now offer some form of XML support. In addition to being able to produce an XML document as the result of a query, the systems can also be used as a repository for XML documents. For example, IBM's DB2 XML Extender [DB2EXT] either can store a whole XML document in one column of a relational table or can decompose XML documents into a set of tables at

load time with the mapping from DTD to relational table defined by DAD (Data Access Definition). Microsoft SQL Server 2000's OPENXML statement can insert or update records of a relational table by specifying meta-properties [MSSQL]. Oracle's XML SQL utilities [ORXSU] can extract data from an XML document, then do insert, update, delete to a relational table. These commercial tools are all conceptually similar to the relational DTD approach that we evaluate in this paper. [POET] and [Excelon], two object-oriented database systems, map each XML element into a separate object. Their approaches are similar to the Object approach described in Section 3 except that they use a separate object for each XML element.

[FK99] evaluates several alternative mappings for storing XML documents in a relational database system. The goals of [FK99] and our work are essentially identical; both explore and evaluate alternative storage strategies for XML documents. Our work extends [FK99] in several ways. First, all strategies considered in [FK99] store XML documents in relational database as XML graphs. We also evaluate a relational database strategy described in [STZ+99] that takes advantage of DTDs. Second, we explore an object-manager approach in which XML files are stored as objects. Third, we consider a new approach that uses only a B-Tree. While this approach is evaluated only in the context of the Shore object manager, it would be possible to integrate this approach with a commercial database system since B-Trees are provided by essentially all database systems. We believe that our results will be useful to a wide range of projects attempting to host XML documents in the years to come.

3. Different Storage Strategies

In this section, six alternative XML storage strategies are described. The XML document "Dept.xml" in Figure 3.1 is used to illustrate how XML data is actually stored with each alternative. An XML document can be modeled as directed graph, with nodes in the graph representing XML elements or attributes and edges representing parent-children relationships between different elements or an element and its

attribute. Such a graph is shown in Figure 3.2. Boxes with rounded corners represent attribute or text nodes.

```
<?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>
```

```
<?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>
```

Figure 3.1 Sample XML file “Dept.xml” and its DTD

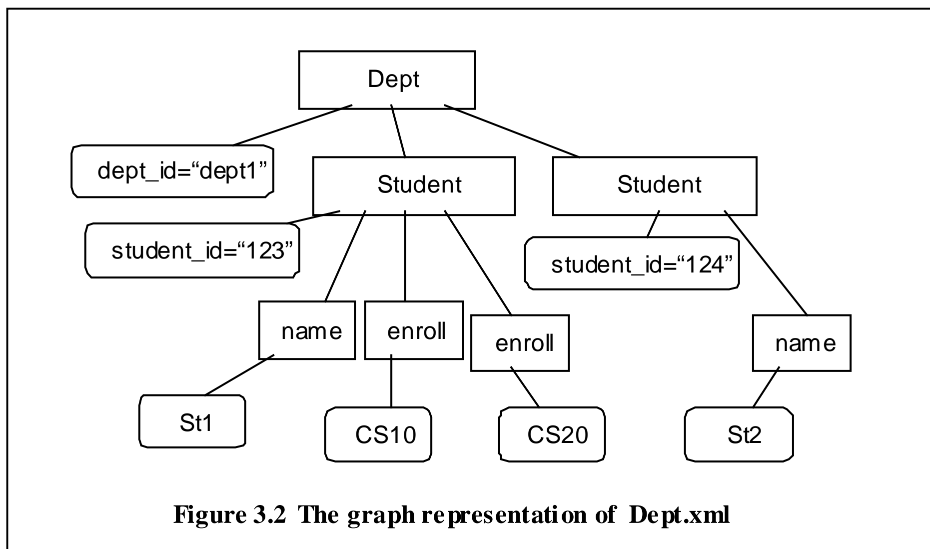

Figure 3.2 The graph representation of Dept.xml

3.1 The Text Approach

The first strategy stores each original XML document an OS file. One way to implement a query engine with this approach is to parse the XML file(s) into a memory-resident DOM tree(s) against which the query is then executed (retaining the DOM in memory as long as some nodes in the tree are needed for query evaluation). This was our first approach but 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 is used 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. An index mapping (tagname, value) or (attribute_name, attribute_value) to element offset is also built to help evaluate selection predicates. A query engine can use these indices to retrieve segments of an XML file relevant to the query. Since only the retrieved segments need to be parsed, the time required for parsing is reduced dramatically. The main disadvantage of this approach is that the indices are hard to maintain if the XML documents are updated, as all the offsets used in the indices will be invalidated.

3.1 The Relational DTD approach

The second strategy was proposed in [STZ+99] and requires the existence of a DTD to insure that conforming documents can be automatically mapped without manual intervention. First, the DTD is itself interpreted as a graph. For example, the "Dept.dtd" can be represented by the

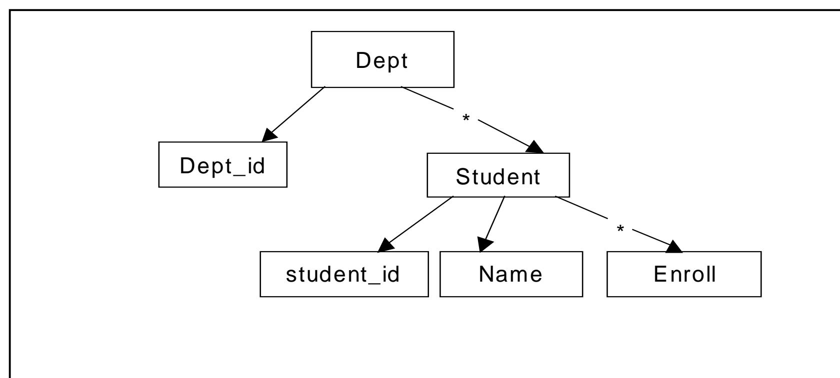

Figure 3.3 The graph representation of Dept.dtd

graph shown in Figure 3.3. The "*" in the edge indicates an element can appear multiple times in its parent element. A separate table must be used to capture this set-containment relationship. Each tuple in a table is assigned an ID and contains a parentID column to identify its parent. If an element can appear only once in its parent, then it is inlined as a column of the table representing its parent. If the DTD graph contains a cycle, a separate relation table must be used to break the cycle. The relational schema generated from the Dept DTD and how the document is stored using this approach is shown below.

parentID	ID	Dept_id
1	2	“dept1”

Table 3.1 The Dept table

ParentID	ID	Student_id	Name
2	3	“123”	“St1”
2	4	“124”	“St2”

Table 3.2 The Student table

ParentID	ID	TEXT
3	5	“CS10”
3	6	“CS20”

Table 3.3 The Enroll table

3.3 The Edge Approach

The third strategy is the "Edge" approach described in [FK99]. The directed graph of an XML file is stored in a single relational table called the “Edge table”. Each node (XML element) in the directed graph is assigned an id. 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 element, and an ordinal number that is used to encode the order of any children nodes. When an element has only one text child, the text is stored with the edge (in-lining in [FK99]).

sourceID	tag	ordinal	TargeteID	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. In this table, the XML document node is assigned ID 1, the Dept element is assigned ID 2, and the two Student elements are assigned IDs 3 and 4. *TargetID* 0 represents the edge points to a TEXT node or ATTRIBUTE node. 0 in *ordinal* field indicates an attribute edge.

As suggested in [FK99], an index is built on **(tag, data)** in order to reduce the execution time of common queries such as *select students with student_id = ‘123'*. In our study, we found that it was also very important to build indices on **(sourceId, ordinal)** and **(targetID)**. This 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.

The clustering strategy on the Edge table has a significant impact on querying performance. While we clustered the "Edge" table on the **Tag** field, an alternative strategy is to cluster the table according to the order of the elements in the original XML files. This strategy has the benefit that elements from one XML file will be stored close to one another in the Edge table. The drawback of this strategy is that elements with the same tag name are not clustered together in the Edge table. 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

[FK99] suggested another approach called the "Attribute" approach. The "Attribute" approach is really a **horizontal partition** of the "Edge" approach by the **Tag** field. Tuples with different tags are stored in separate tables. Since this approach avoids storing tag value in the table, it results in a more compact representation. 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 because it requires a DTD for query processing. With the "Attribute" approach, a query processor needs a DTD to decide which table contains sub-elements since the tags of the sub-elements of an element are not recorded in the table. Thus, the "Attribute" approach cannot be used to store XML files without a DTD. Furthermore, for a large collection of XML documents, the attribute approach can result in a large number of relation tables.

3.5 The Object Approach

There are several ways of mapping XML documents into objects stored in an object manager providing B-tree indices. The obvious approach is to store each XML element as a separate object. However, since elements are usually quite small, we found the space overhead of this strategy prohibitive. Instead, all the elements of an XML document were 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.

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 Attributes, #PCDATA=”St1”
100✍	Length=20, **Enroll**, parent=40, prev=80, next=120, No children, No Attributes, #PCDATA=”CS10”
120✍	Length=20, **Enroll**, parent=40, prev=100, next=nil, No children, No Attributes, #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 Attributes, #PCDATA=”St2”

Figure 3.4 File object holding “Dept.xml”

Figure 3.4 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 3.4. The **length** field records the total length of the *lw_object*. The **flag** field contains bits that indicate whether or not 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. The children list of a node is implemented as doubly linked list via the **prev** and **next** fields. There are three optional fields. **Opt_child** records the *lw_oids* of the first and last child, if the lw_object has children. **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* and is marked as `TEXT_NODE` in the flag field. We built 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. This index is helpful in retrieving all children of a node with a given tag.

3.5 The B-tree Approach

If updates are frequent, the object approach has a number of drawbacks. First, since updated lw_objects that grow in size must be invalidated and appended to the end of the file_object, the file_object tends to become fragmented, and space utilization deteriorates. Linking information of other lw_objects also needs to be updated. Implementing a map from logical element id to physical offset would eliminate this problem. In the B-tree approach, we extend the object approach further by implementing a map from logical element id to the lw_object itself instead of to the physical offset of the lw_object inside the file_object.

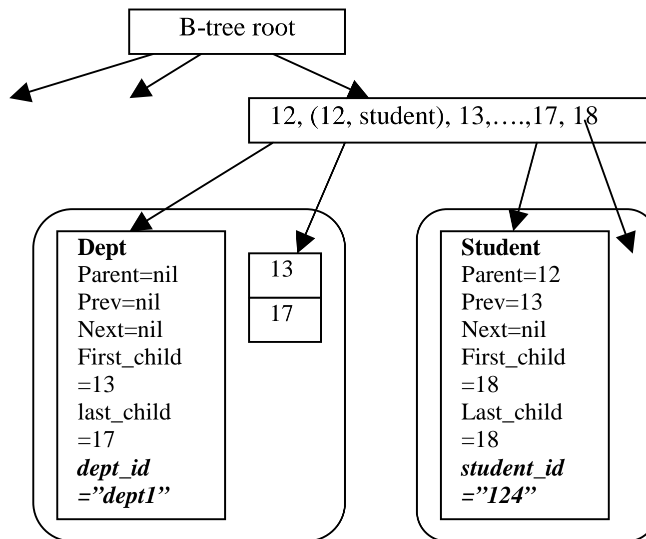

Figure 3.5 B-tree representation of "Dept.xml"

As with the Edge approach, multiple XML documents can be stored in one B-tree in order to save disk space. Each XML document in the B-tree is assigned a range of element ids and elements of the DOM Tree corresponding to an XML document are assigned element ids in this range in a depth-first order when the document is initially loaded. This element id is used as the key of the B-tree. The data values in the leaves of the B-tree have exactly the same format as the lw_object in the object approach. Figure 3.5 shows the example XML file inside a B-tree.

When updating an *lw_object*, only that *lw_object* needs to be updated; the pointers from other lw_objects do not have to be changed. Since the B-tree code automatically manages the use of

space in the leaves there is no need to invalidate or move objects when updates cause them to grow in size. This strategy does, however, incur some additional overhead when locating elements. Instead of simply performing an offset lookup, the element id is used to perform a B-tree look up. Since this kind of look up is used frequently while navigating from one element to another, the depth-first assignment of element ids makes the overhead relatively small since we can expect the related element to be on either the same leaf page or a neighboring page. As with the object approach, we also built an index on (tag, opt_text) and (attr_name, attr_value) fields and a path index mapping (parent_id, tag) to child_id. Since XML documents are modeled as a directed graph, most queries on XML data will generate a navigational workload from the parent nodes to their children. In order to optimize this type of navigation, the path index is clustered with the lw_object so that that a single disk read will bring both the *lw_object* and the path index into memory. Figure 3.5 illustrates how path index from the element 12 (the dept element) via tag "student" can be inlined in the B-tree.

4. Performance Study

This section evaluates the performance of the six strategies described in Section 3 on two different datasets. The first dataset models a university department database like that described in [CDN]. It contains 250 XML files, 114MB in total. Data about each department is stored in a separate XML file. Each department has a number of Professors, Staffs, and Students. A student may be a teaching assistant for a course in or outside of his major department. These XML files also contain typical course enrollment information. 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. The second dataset we used is the Open Directory Project data from [ODP], 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

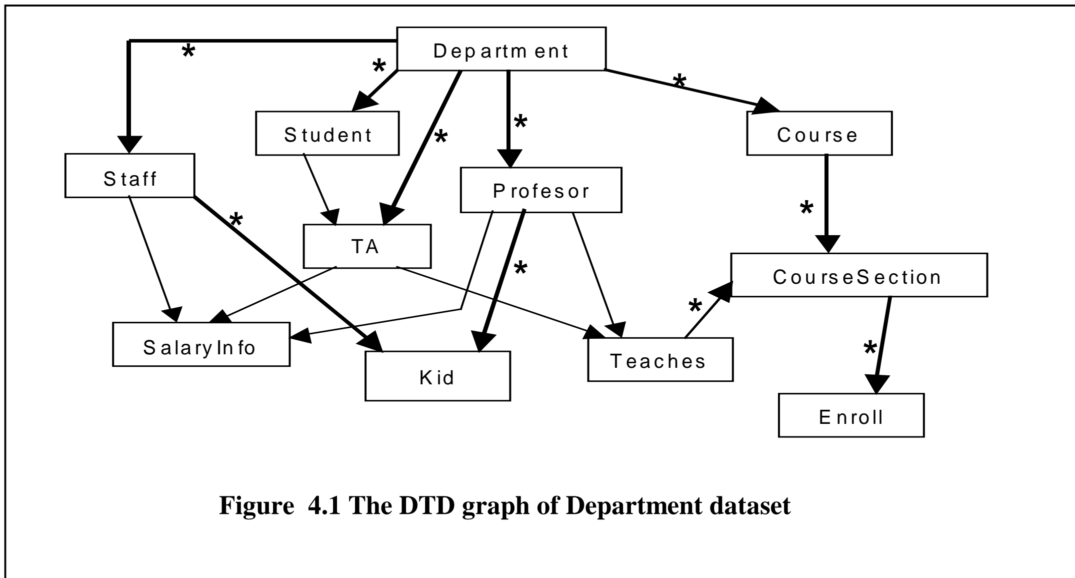

Figure 4.1 The DTD graph of Department dataset

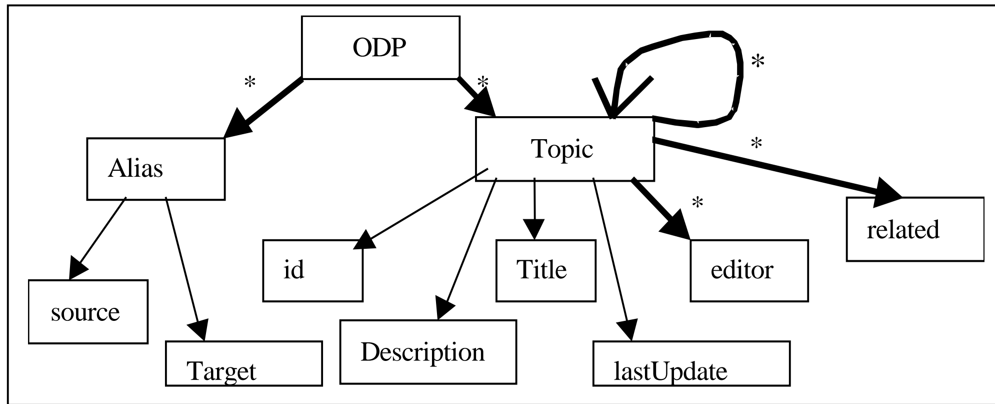

Figure 4.2 The DTD graph of ODP dataset

Notice 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. In the experiments, we will examine the impact of DTD cycle on query processing.

Appropriate indices for each strategy were built to facilitate query processing. Table 4.1 lists the indices used with each approach.

	Table name/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 (parentId, tag, childId)
B-tree	(tag, data), (attr_name, attr_value), inlined path index (parentId, tag, childId)

Table 4.1 Indices of each approach

Table 4.2 summarizes the space consumed by each strategy. While a separate B-tree is used for the path index with the Object approach, the index is inlined in the B-tree approach.

		TEXT	DTD	Edge	ATTR	Object	B-Tree
Department Dataset	**Data**	114	69.7	223	165	104	188
	Indices	206	29.3	167	130	164	104
ODP Dataset	**Data**	145	126	222	187	160	208
	Indices	212	132	190	181	192	160

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 Redhat Linux 6.2. DB2 V7.1 was used as the relational DBMS. The Object and B-tree strategies were implemented using Shore [CDN+94]. Both DB2 and Shore were configured to use a 30MB memory buffer pool. There is no buffer pool for the Text approach, and the application uses as much physical memory as available (256M). The indices for the text approach are implemented using Berkley DB toolkit [BDB]. For the DTD and Edge approaches, Quilt queries were manually translated to SQL queries to be executed by DB2.

When appropriate, all tests were conducted with both cold and warm buffer pools. The DB2, Shore, and Linux buffer pools were flushed in order to measure performance from a cold start.

4.1 Experiment 1: Load Database and Reconstruct Original XML Documents

The first experiment measures the database load time and the time required to reconstruct the original

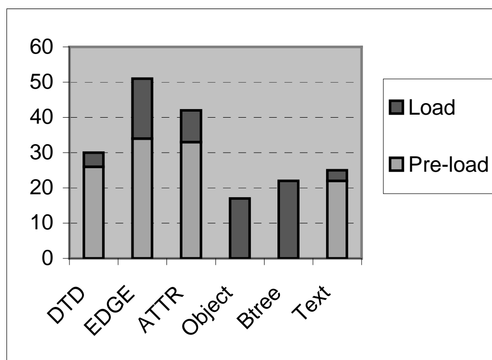

Fig 4.3 Department Data load time (in minutes)

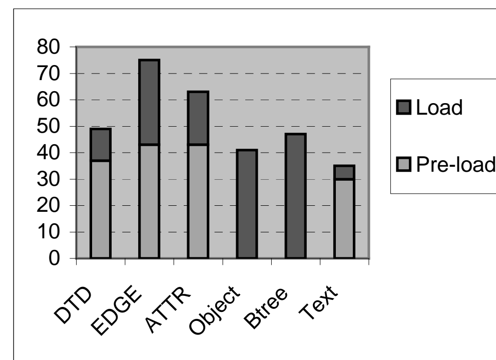

Fig 4.4 ODP Data load time (in minutes)

documents from the database and write them to the file system. For the DTD, Edge and Text strategies,

we first generate database load files (the pre-load phase) and then the load files are used to populate tables in DB2 or Berkeley DB (load phase). These load files are generated using a Python script. The Reconstruct experiment measures the time to traverse from the root to all of its descendants. This access pattern is also encountered when constructing the answer to Quilt queries. There is no reconstruct time for TEXT approach since the original XML files is stored in file system. We only present performance when memory is cold because this experiment scans the whole database and warming up memory does not significantly reduce execution time.

	DTD	**Edge**	**Attribute**	**Object**	**B-Tree**
Department Dataset	1404 sec.	2011 sec.	3100 sec.	78 sec.	154 sec.
ODP Dataset	1184 sec.	1833 sec.	2856 sec.	81 sec.	149 sec.

Table 4.3 Reconstruction time

We observe that the Object approach has the best performance reconstructing the original XML documents because each XML document is stored as a single object. The B-tree approach is slower mainly as the result of the cost of performing a B-tree lookup for each element. Both are faster than relational database approaches. As discussed previously, the Edge table is clustered according to Tag name. Hence, the order of tuples in the Edge table no longer reflects the original order of elements in XML files and reconstruction incurs a large numbers of random I/Os. In the Edge approach, ***one*** SQL query is issued to obtain element id of all sub-elements. In order to reconstruct an XML element with Attribute approach, the 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 of sub-elements. This results in a much larger number of SQL queries being executed.

4.2 Experiment 2: Selection Queries

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

4.2.1 Selection Query 1: Index look up

4.2.1.a Index look up on Department data

SQ_1A: Find Staff name whose id is ‘P_77’ #of result : 1	**Quilt**: FOR $s in document()/department/Staff WHERE $s/@id=’P_77’ RETURN <result> $s/name </result>
SQ_1A*: Find personnel whose Id is ‘P_77’ #of result : 1	**Quilt**: FOR $p in document()/department/* WHERE $p/@id = ‘P_77’ RETURN <result> $p/name </result>

While the DTD and Attribute approaches simply perform an indexed selection query on the Staff table, the other approaches must apply the predicate Tag=”Staff”. SQ_1A* performs an index lookup with a regular path expression. The SQL queries for the DTD and Attribute approach consist of unions of three selection queries on the Staff, Professor, and Student tables. The other approaches no longer need to check tag name. Since the Edge approach actually needs to perform a join operation to retrieve the tag name, this means there is one less join operation than for query SQ_1A, even though one still needs a join to retrieve the Name value.

		DTD	Edge	Attribute	Object	B-Tree	TEXT
SQ_1A	**Cold**	0.4	0.5	0.5	0.21	0.18	0.3
	Warm	0.04	0.07	0.04	0.03	0.03	0.0006
SQ_1A*	**Cold**	0.44	0.63	0.77	0.21	0.2	0.19
	Warm	.004	0.04	0.06	0.03	0.03	0.0004

Table 4.4 SQ_1A and SQ_1A* (time in seconds)

For these two queries, the relational database based approaches have worse performance than object manager and text based strategies due to the overhead of relational query engine. The Edge approach translates the selection query to SQL join queries using an index nested loop join. One interesting result of the Edge approach is the regular path expression actually results in a simpler SQL query.

4.2.1.b Index scan on ODP data

SQ_1B: select Topic description with Title “Photography”. # of result: 64	**Quilt:** 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” #of result: 347	**Quilt:** FOR $t in document()//topic WHERE $t//Title = ‘Photography’ RETURN $t/Description

		DTD	Edge	Attribute	Object	B-Tree	TEXT
SQ_1B	**Cold**	0.8	1.2	2.3	9.4	8.4	6.7
	Warm	0.02	0.03	0.07	0.04	0.06	0.02
SQ_1B*	**Cold**	2.4	10.7	7.4	9.6	8.7	7.3
	Warm	.16	0.37	0.43	0.04	0.06	0.02

Table 4.5 SQ_1B and SQ_1B* (time in seconds)

DTD, Edge and Attribute approaches cluster elements with the same tag name together. Elements with same tag are scattered for object manager based approaches. After the index look up using Title='Photography', chase parent/child links in Object, B-Tree and TEXT incurs lots of random I/O. In SQ_1B*, Object, B-tree and text shows similar results compared to SQ_1B. The cycle in DTD graph forces an expensive fixed point evaluation with Edge and Attribute approaches, thus their running time for SQ_1B* is much slower that SQ_1B.

4.2.2.a Scan Selection on Department data

SQ_2A: Select professor id, name with salaries higher than $60,000 # of results: 19289	**Quilt**: FOR $p in document()/department/professor WHERE salary($p) > 60000 RETURN $p.id, $p.name
SQ_2A*: Select person id, name with salaries higher than $60,000 # of results: 30061	**Quilt:** FOR $p in document()/department/* WHERE salary($p) > 60000 RETURN $p.id, $p.name

The next query finds either professors or persons (professors, staff members, or TAs!) with salaries greater than $60,000. The Salary of an employee of the department is computed by the ***salary()*** function using the SalaryInfo sub-element of Professor, Staff, or TA.

		DTD	Edge	Attribute	Object	B-Tree	TEXT
SQ_2A	**Cold**	1.97	18.4	13.2	25	17	29
	Warm	0.9	13.6	8.8	6.5	9	2.6
SQ_2A*	**Cold**	6.6	32.8	28.4	50.7	31.6	34
	Warm	3.4	29.2	25.3	21.8	23.1	11

Table 4.6 SQ_2A and SQ_2A* (time in seconds)

We observe that clustering the same type of elements together (e.g. all Professors) 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 the Professor, Staff and TA elements while the Edge and Attribute approaches need to perform joins to retrieve those values. Inlining the path index inside the B-tree helps the B-tree approach win the cache cold case over the Object approach, even though looking up an element using a B-tree key is more expensive than using simply using the element's offset. The Text approach has essentially the same access pattern as the Object approach, but has better warm cache performance because the Text approach uses all available physical memory (256M) and does not have overhead of locking and logging.

4.2.2.b Scan selection on ODP data

SQ_2B: find topics that are updated in last quarter of a year. # of resut: 19155	**Quilt:** FOR $t in document()//topic WHERE month($p/lastupdate) >= 10 RETURN $t/Description
SQ_2B*: find topics that contain a sub-topic which is updated in last quarter of a year. # of result: 24929	**Quilt:** FOR $t in document()//topic WHERE month($p/lastupdate) >= 10 RETURN $t/Description

		DTD	Edge	Attribute	Object	B-Tree	TEXT
SQ_2B	**Cold**	5.1	11.8	4.5	45	47	31
	Warm	4.4	11.7	4.3	47	44	3.7
SQ_2B*	**Cold**	83	80	72	47	49	41
	Warm	88	80	68	47	46	12

Table 4.7 SQ_2B and SQ_2B* (time in seconds)

Comparing SQ_2B with SQ_2A, we see that Edge and Attribute perform much closer to the DTD approach. The salary function used in SQ_2A contains several path expressions, while the month function in SQ_2B is a simple function of string variable. Because each path expression is translated into a join operation with the Edge and Attribute approaches, they are more sensitive to the complexity of the path expression than the DTD approach. SQ_2B* requires recursive SQL query for the relational database based approaches.

4.3 Set containment queries

4.3.a Set containment queries on Department data

CQ_1: Select ids and names of professors who has a kid named "girl16" # of results: 261	**Quilt:** FOR $p in document()/department/professor WHERE $p/kid="girl16" RETURN $p/id, $p/name
CQ_2: Select all student id, name in department 2. # of result: 269	**Quilt:** FOR $d in document()/department WHERE $d//@id='dept2' RETURN FOR $s in $d//student RETURN $s/@id, $s/name

		DTD	Edge	Attribute	Object	B-Tree	TEXT
CQ_1	**Cold**	1.2	27.1	9	5.6	4.6	21
	Warm	0.09	0.36	0.18	0.12	0.13	0.2
CQ_2	**Cold**	0.2	2.4	1.8	0.43	0.3	0.6
	Warm	0.06	1	0.06	0.07	0.08	0.02

Table 4.8 CQ_1 and CQ_2 (time in seconds)

CQ_1, searches for professor elements ***containing*** child named "girl16". The second, CQ_2, first locates the department with id "dept2" and then it retrieve students ***contained in*** that department. As shown in Table 4.7, the DTD approach exhibits excellent performance for both containment queries because similar elements are clustered together (such as all Students). The set containment query is translated into a join query in the relational query engine. The information that is needed to construct the result of the query is readily available as columns of the relational table. The Object and B-tree approaches cluster elements in the original order of the text file, storing contained elements "close" to their containing elements. However, since there may be many contained elements they may end up residing on different pages. CQ_1 requires navigating from child (Kid is girl16) to parent nodes (Professor). While the parent id is stored as a field of children in the Object and B-tree approaches, the Text approach must use the inverted path index to look up the parent id. The Edge and Attribute approach suffer significantly from the cost of constructing query results as tuple corresponding to a single real world object (eg. id and name) are scattered around the relation table.

4.3.b Containment queries for ODP data

CQ_3: Find descripton of Topics who have a sub-topic edited by “tim”. # of result: 42	**Quilt:** FOR $t in document()//Topic WHERE $//editor = ‘tim’ RETURN $t/Description
CQ_4: Find sub-topic of Topic 10366 #result = 98	**Quilt:** FOR $t in document()//Topic WHERE $t/@catid=’10366’ RETURN $//Topic/Dedsciption

		DTD	Edge	Attribute	Object	B-Tree	TEXT
CQ_3	**Cold**	0.7	2.2	1.1	1.7	1.5	1.2
	Warm	0.08	0.08	0.1	0.03	0.02	0.03
CQ_4	**Cold**	1.8	2.5	2.8	1	0.9	1.4
	Warm	0.5	0.6	0.6	0.2	0.2	0.14

Table 4.9 CQ_3 and CQ_4 (time in seconds)

Both CQ_3 and CQ_4 contains recursive query processing within the set containment queries. For Object, B-Tree and Text approaches, CQ_3 traverse tree upward (from child to parent) and CQ_4 traverse the tree downward (from parent to child). Because Object, B-tree approaches cluster elements according to the order of original XML file, all I/O (sequential) needs to be performed by CQ_4 is confined in one topic element. On the other hand, traversing upward is more likely to incur random I/O.

4.4 Join Queries

4.4.a Join queries on Department data

JQ_1: Find students with same birthdate and zipcode. # of results : 3	**Quilt**: FOR $s1 in document()/department/student RETURN <result> FOR $s2 in document()/department/student WHERE $1/birthdata = $s2/birthdate and $s1/zipcod = $/s2/zipcode and $s1/@id != $s2/@id RETURN $s1/@id, $s1/name, $s2/@id, $s2/name </result>
JQ_2: Find name of professors and TA of department 99 that they teach in same room # of results: 25	**Quilt:** FOR $t indocument()/department[@id=’dept99’]/TA RETURN FOR $p in document()/departartment[@id=’dept99]/professor WHERE $p/teaches/CourseSection/Building =$t/teaches/CourseSection/Building and $p/teaches/CourseSection/RoomNo =$t/teaches/CourseSection/RoomNo RETURN $t.name, $p.name

		DTD	Edge	Attribute	Object	B-Tree	TEXT
JQ_1	**Cold**	3.4	35	31	30	22	35
	Warm	2.5	32	21	12	15	4.8
JQ_2	**Cold**	0.28	15	0.7	0.52	0.58	0.8
	Warm	0.03	12	0.45	0.3	0.42	0.17

Table 4.10 JQ_1 and JQ_2 (time in seconds)

JQ_1 is relatively simple and can be directly translated into a self-join query on the Student table with the DTD approach. For the Object and B-tree approaches, we implemented a hash join and assumed that the hash table fits in memory (the hash table is built on the BirthDate, Zipcode to the element id values of the Student elements). The Same hash join algorithm is used for the Text approach. The second join query, JQ_2, has a much more complicated path expression. The Edge approach translates this XML-QL query to an SQL query that contains 14 joins over the Edge table. DB2 was unable to find a good plan for this query so we manually broke it up into several smaller queries connected with temporary tables to hold the intermediate results.

4.4.b Join on ODP data

JQ_3: Retrieve descpritions for same subtopic of Illinios and Wisconsin # of result: 250	**Quilt:** 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	B-Tree	TEXT
JQ_3	**Cold**	1.5	17	15	1	1.2	1
	Warm	1	10	9	0.3	0.3	0.4

Table 4.11 JQ_3 (time in seconds)

This is a join consists of fixed point evalationation of both side of the join operator. The cost of evaluate the recursive query with Edge and Attribute approaches is high. We examined the execution plan and find the execution plan is sub-optimal because it is hard to estimate the size of the output of fix point evaluation.

4.5 Discussion

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 clustering strategies 1 and 2 aggressively. Also, DTD information helps to produce a much more compact data representation and much more compact indices. The drawback of this approach is it cannot handle DTD-less XML files. 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 web's data currently resides, and will continue to reside, in relational database systems, using a relational DMBS to store XML documents makes it possible to seamlessly 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 of the Edge table. This produces very complex SQL queries involving tens of joins for complex path expressions making 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. Breaking up edge table also helps relational database to optimize query, as demonstrated by JQ_2. 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 need to fetch all sub-elements.

Both the Object and B-tree approaches use 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 producing 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. Our experiments have demonstrated that the object and B-tree strategies have similar performance across the entire range of queries. This is hardly surprising as the Object and B-tree approaches have similar access paths. The object approach generally has better warm cache performance because using offsets inside the object as element ids is faster than performing a B-tree lookup. However, inlining the path index in the B-tree approach incurs fewer I/Os when the cache is cold. Furthermore, since the B-tree implementation exploits Shore's B-tree code to handle space allocation, de-allocation, and clustering, it was much easier to implement than the object approach.

We have also run these queries against XML documents stored as text XML files in the file system but without any external indices. Parsing a whole text XML file for query processing is prohibitively expensive. By building the appropriate indices, the query engine can retrieve and parse only the necessary parts of the relevant text files. This approach has similar performance to that of an object manager. Its main disadvantage is the high cost of maintaining the indices when the text XML files are updated. This approach is really only viable in a read-only environment such as web caching. The Text approach usually has better cache warm performance than object manager based approaches mainly because there is no buffer pool size limitation.

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 and 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 more compact data representation and excellent performance across different datasets and different queries. We conclude DTD approach is the best strategy among the six approaches we studied and there is no clear need to build an “XML-specific” database system.

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 Quilt 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 approaches. However, the cost of maintaining indices will make this approach only useful when update frequency is low.

References

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

[ACM93] S Abiteboul, S. Cluet, T Milo. *Querying and updating the file*. VLDB 1993, pp73-84

[AQM+97] S. Abiteboul, D. Quass, J. MeHugh, J.Widom, J.Wiener. *The Lorel Query Language for Semi-structured Data*, International Journal on Digital Libraries, 1(1), pp. 68-88, April 1997.

[BBB+88] F. Bancihon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, F. Velez. *The design and implementation of O2, an object-oriented database system*. In Proceedings of the second international workshop on object-oriented database, 1988, ed. K Dittrich.

[BPS98] T. Bray, J. Paoli, C.M. Sperberg-McQueen, *Extensible Markup Language (XML) 1.0,* http://www.w3.org/TR/REC-xml

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

[Bun97] P. Buneman, *Semi-structured data*, PODS 1997, 117-121.

[CDN+94] M. Carey, D. DeWitt, J. Naughton, M. Solomon, et. al, *Shoring Up Persistent Applications*, Proc. of the 1994 ACM SIGMOD Conference

[CDN+97] M. Carey, D. DeWitt, J. Naughton, M. Asgarian, P. Brown, J. Gehrke, D. Shah, *The BUCKY Object-Relational Benchmark,* SIGMOD 1997

[DB2EXT] IBM DB2 XML Extender. http://www4.ibm.com/software/data/db2/extenders/xmlext/

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, *XML-QL: a query language for XML,* In Proc. Of the Int. WWW Conf., 1999.

[DFS99] A. Deutsch, M. F. Fernandez, D. Suciu, *Storing Semi-structured Data with STORED*, SIGMOD Conference 1999: 431-442

[Excelon] Excelon, the ebusiness information server. http://www.odi.com/excelon

[FK99] D. Florescu, D. Kossman, *A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database,* Rapport de Recherche No. 3680 INRIA, Rocquencourt, France, May 1999

[KM00] C. Kanne, G. Moerkotte, *Efficient storage of XML data*, ICDE 2000, pp198

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, *Lore: A Database Management System for Semi-structured Data,* SIGMOD Record 26(3): 54-66 (1997)

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

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

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

[POET] POET content manager suit. http://www.poet.com/

[CRF00] Quilt: An XML Query Language for Heterogeneous Data Sources. http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html

[SLS+93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas, *The Rufus system: Information organization for semi-structured data,* Proc. Of the Int. Conf. On VLDB, pages 97-107, Dublin, Ireland, 1993.

[STZ+99] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, J. F. Naughton, *Relational Databases for Querying XML Documents: Limitations and Opportunities*. VLDB 1999: 302--214

[Wid99] J. Widom, *Data Management for XML Research Directions,* IEEE Data Engineering Bulletin, Special Issue on XML, 22(3):44-52, September 1999.

[XML4C] XML4C parser by IBM AlphaWork. http://xml.apache.org/xerces-c/index.html