Paper ID: 151

Contact Author:
Feng Tian ftian@cs.wisc.edu
Department of Computer Science
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI, 53706
Phone: (608)2626622

Paper ID: 151

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

Authors:
Feng Tian, David J. DeWitt, Jianjun Chen, Chun Zhang

Category: Research

The Design and Performance Evaluation of Alternative XML Storage Strategies

Feng Tian David J. DeWitt Jianjun Chen Chun Zhang

Computer Sciences Department
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. This paper describes five different approaches for storing XML documents including two strategies that use the OS file system, one that uses a relational database system, two that use an object-oriented storage manager. We implemented each of the five approaches and evaluated its performance using a number of different XML-QL queries in conjunction with XML data from the DBLP web site. We conclude with some insights gained from these experiments and a number of recommendations on the advantages and disadvantages of each of the five approaches.
Category: Research

1. Introduction

The Extensible Markup Language (XML) [BPS98] is an emerging standard for data representation and exchange on the Internet. In the near future it is expected that XML 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 file can have a Document Type Definition (DTD) associated with it. A DTD serves an implicit semantic schema for the XML document and makes it possible to define much more powerful queries than simple, text-based 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 mark-up language, a web browser can browse XML file in the same way as an HTML file. Second, and more interesting to the database community, it can serve as a standard way of storing semi-structured data sets. XML offers significant opportunities for users to ask very powerful queries against the web. For example, doing a keyword search of "car", "price" and "safety" on the web will probably return millions of documents. Posing the query "find the top 10 safest cars that are priced below $25000" using an XML-based query language such as XML-QL [DFF+99] makes it possible to return the answer the user really wants.

While there have been numerous research projects on semi-structured query languages and data models [Abi97] [AQM+97] [MAG+97], focus recently has shifted toward querying XML data sets. One important question is what is the best way of storing XML documents as the performance of the underlying storage representation has a significant impact on query processing efficiency.

There have been numerous studies of alternative storage models and systems [CDN+94][CK85] and in recent years several projects have proposed alternative strategies for storing XML data sets [FK99][STZ+99]. XML storage strategies can be classified into three categories according to the underlying system used: file system, database system, or storage manager. To the best of our knowledge there have been no careful performance studies comparing these alternatives and it is still an open question which strategy is best.

We briefly describe these three alternatives. One common, and obvious, solution is to store XML documents as ASCII files in the operating system file system. This is the standard approach today. Such files tend to be small and are mainly accessed via HTTP requests. When an XML query is run against XML data stored in the file system, typically the data is read from the file and then parsed into a memory tree format such as DOM 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. On the other hand, it has several major drawbacks. First, XML files in ASCII format need to be parsed every time when they are accessed for either browsing or querying. Second, the entire parsed file, which is always much larger than the original XML document, must be memory-resident during query processing. Third, it is hard to build and maintain indices on documents stored this way. Another serious drawback is that update operations are difficult to implement.

More promising from both a research and commercial point of view is to use a commercial database management system (DBMSs) to store XML documents. Several researchers [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 these products have shown very limited commercial success when compared with their relational counterparts, in this paper we explore the use of a RDBMS instead. Storing XML data in a relational database system is not trivial, since the data model of XML is very different from the relational data model. Recently several papers [FK99][STZ+99] have studied methods for disassembling XML data into multiple relations.

Since storing and accessing XML data through a query interface of a DBMS incurs overhead that is not related to storage, one may wonder why not use an object manager (OM) such as Shore[CDN+94] instead. While the use of a storage manager can be expected to provide better performance, the record-level interface provided by a typical storage manager requires more work to use than a SQL or OQL based-interface. In addition, as a standard SQL, offers a degree of portability not found among different object managers.

In this paper, we describe five alternative ways of storing XML documents: two that employ ASCII files stored in the file system, one that uses a relational database system, and two that use a storage manager providing objects and B-tree indices. These alternatives are evaluated using two different workloads. The first mimics the workload of an XML storage server and consists of the type of requests an XML server is likely to receive from either a web browser or database query engine. Web browser requests usually generate workloads that are heavily navigational. A navigational workload 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 in a semi-structured environment. With an index, the query can directly access the relevant elements. Without an index, each document must be examined by navigating from the root node to the desired elements. The storage design must support both workloads efficiently.

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

The remainder of this paper is organized as follows. Section 2 contains a survey of related work. In Section 3 we describe five 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]. Storage models and systems have also been studied for decades [CDN+94][CK85].

Recently several projects have investigated strategies for storing semi-structured and XML data sets to facilitate efficient query processing. [MAG+97] developed a special purpose database system that utilizes special features of the XML model. A second approach is to store XML data in a relational DBMS or

OODBMS [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] extracts a relational schema from the DTD, the storage strategies we describe in this paper involve storing the labeled XML graph (at least logically) and do not depend on the existence of a DTD, as some XML files do not have an associated DTD. One implementation of LORE [MAG+97] explored the use of O2 as an underlying storage system but did not evaluate the performance compared to the standard LORE storage manager.

[FK99] evaluates several alternative mappings for storing XML documents in a relational database system. The goals of [FK99] and our own work are essentially identical; both explore and evaluate alternative storage strategies for XML documents. Our work extends [FK99] in several ways. First, we explore an object approach and evaluate its performance when implemented both in the file system and in a general-purpose object manager. Second, we consider a novel approach that uses only a B-tree. While we evaluate this approach only in the context of the Shore object manager, it would be straightforward to add it a relational database system. We believe that our results will be useful to a wide range of those people attempting who want to host XML documents in the years to come.

3. Different Storage Strategies

Recently there has been a lot of interest in XML and strategies for storing and querying XML documents. One simple approach is to store each XML document as a separate operating system file and use a DOM parser whenever the document is accessed by a query. This approach is trivial to implement and DOM parsers are widely available. Two other approaches include using a relational DBMS or a storage manager (SM) to store the XML documents. In the relational approach, XML data is stored in relations and the XML query language (for example, XML-QL) is translated to SQL and executed by the underlying relational database system. In the SM approach, the XML query is parsed, translated to a suitable operator tree representation, optimized, and then executed by an XML Query Engine. Figure 3.1 illustrates the two different approaches. The boxes with dashed lines indicate the components that need to be developed with each approach. The RDBMS approach on the left is clearly much easier to implement, but the SM approach can probably produce better performance since an XML-specific query engine can exploit the knowledge of XML access patterns. Also, the RDBMS layer incurs additional overhead for parsing and optimizing the generated SQL queries, binding SQL parameters, and fetching results out of the RDBMS.

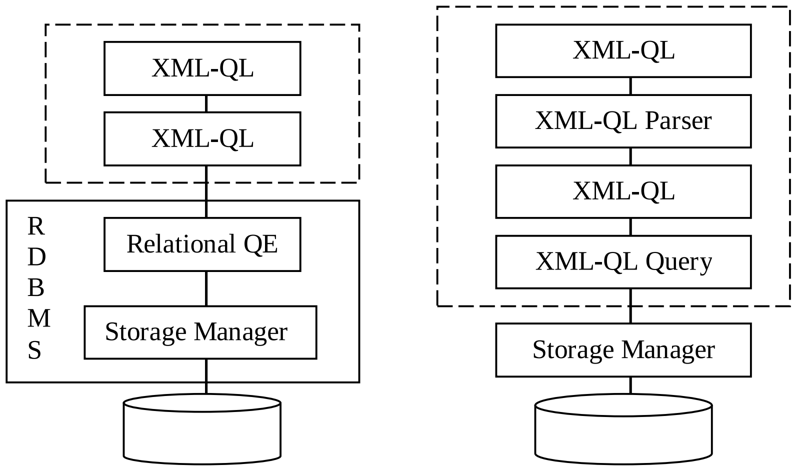

Figure 3.1 Various XML Database System Architectures

In this section, we describe three different XML storage strategies, one of which is implemented in two different ways. The first strategy uses a Relational DBMS as underlying storage system. Two other strategies use Shore[CDN+94], a storage manager developed at Wisconsin. The last strategy uses the Unix file system and the Berkeley libdb toolkit [OBS].

We will use the XML document in Figure 3.2 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 a parent-children relationship between different elements or an element and an attribute. Such a directed graph is shown in Figure 3.2.

```
<dblp><article key=“sample_1”>
<title>Sample One</titel>
<author>Author1</author>
<author>Author2</author>
<cite label=“sampart2”>
      sample_2
</cite></article>
</dblp>
```

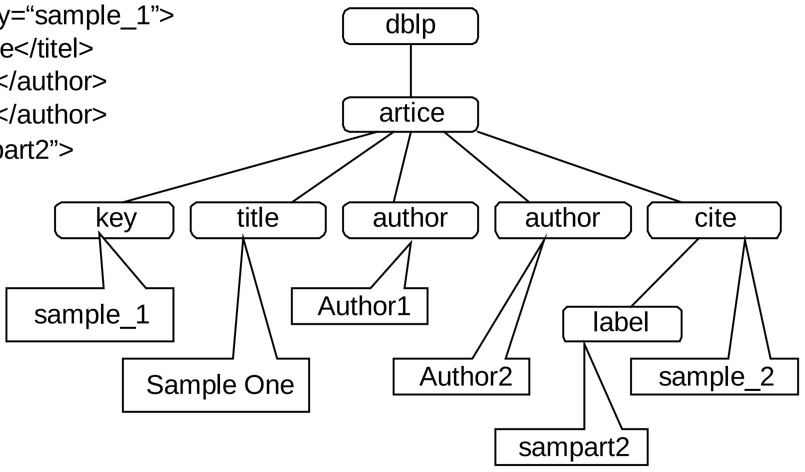

Figure 3.2 A Simple XML File and its Tree Model

3.1 The Edge Table

The first 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". Every 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 source and target of the edge, the tag of target element, and an ordinal number that is used to record 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]). A special tag _TEXT_ is reserved for elements that have both text fields and other children nodes. The edge table has the following schema:

Edge_table(sourceID, ordinal, flag, tag, targetID, data)

with (sourceID, ordinal) as key. The flag field contains bookkeeping information such as whether the child is an element or attribute. Table 3.1 contains the Edge_Table for the example shown in Figure 3.2 (the flag column has been omitted to simplify the presentation).

sourceID	Ordinal	Tag	targetID	Data
0	1	dblp	1	""
1	1	article	2	""
2	1	key	0	"sample_1"
2	2	title	0	"Sample One"
2	3	author	0	"Author1"
2	4	author	0	"Author2"
2	5	cite	3	""
3	1	label	0	"sampart2"
3	2	_TEXT_	0	"sample_2"

Table 3.1 Edge Table for example in Figure 3.2

To reduce the number of tables in the database and the amount of disk space consumed, we store multiple XML files in one Edge table. A separate table is used to record the mapping of the name of the XML file to the id of each document's root element. As suggested in [FK99], an index is built on the *tag* attribute in order to reduce the execution time of common queries such as *select documents with title = 'sample article'*. In our study, we found it is also very important to build an index on the *targetID* attribute. This index is used when traversing from a child node to its parent. This type of navigation is very common when either browsing or querying XML documents.

[FK99] concluded that another approach called “Attribute” actually provides better performance than the “Edge” approach. We did not use this approach because:

1. The “Attribute” approach is really a ***horizontal*** partition of the “Edge” approach using the ***tag*** field. It is more suitably considered as a clustering policy. Specifying a clustered index on (tag) for the Edge table achieves essentially the same effect.
2. The “Attribute” approach breaks one XML file into multiple relations, which wastes disk space and incurs significant overhead in maintaining the database catalog. Furthermore, since database systems have fairly low limits on the total number of tables permitted, this approach is probably not viable for large collections of XML documents.
3. There are XML files that do not have a DTD. When evaluating a query that has a regular expression path, it would be impossible to express the XML query in SQL because there is no way to determine which tables in the relational database should be queried.

We also considered other mapping strategies. For example, instead of storing edge information, one could store each node of the directed graph in a "Node" table and also record its parent and children id. We conclude these two strategies are equivalent with the appropriate indices (path index in "Node" table, and the target index on "Edge" table). Our experiments also showed the "Node" approach and "Edge" approach have similar performance. Thus, only Edge table approach is considered further in this paper.

3.2 The SM Object Approach

In the SM Object approach, Shore[CDN+94] is used as the underlying storage system. The obvious solution is to store each element of the XML file as a separate object in Shore. However, since each element in an XML document is usually quite small, the space overhead of this strategy is very high. Instead we use one Shore object to hold the entire XML document with the XML elements becoming *light-weight objects* inside the Shore object. We use the term ***lw_object*** to refer to the light-weight object and ***file_object*** to denote the Shore storage manager object corresponding to the XML document.

Figure3.3 shows how the example XML file is stored in a *file_object*. The format of each *lw_object* is:

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 (ele_id), 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.

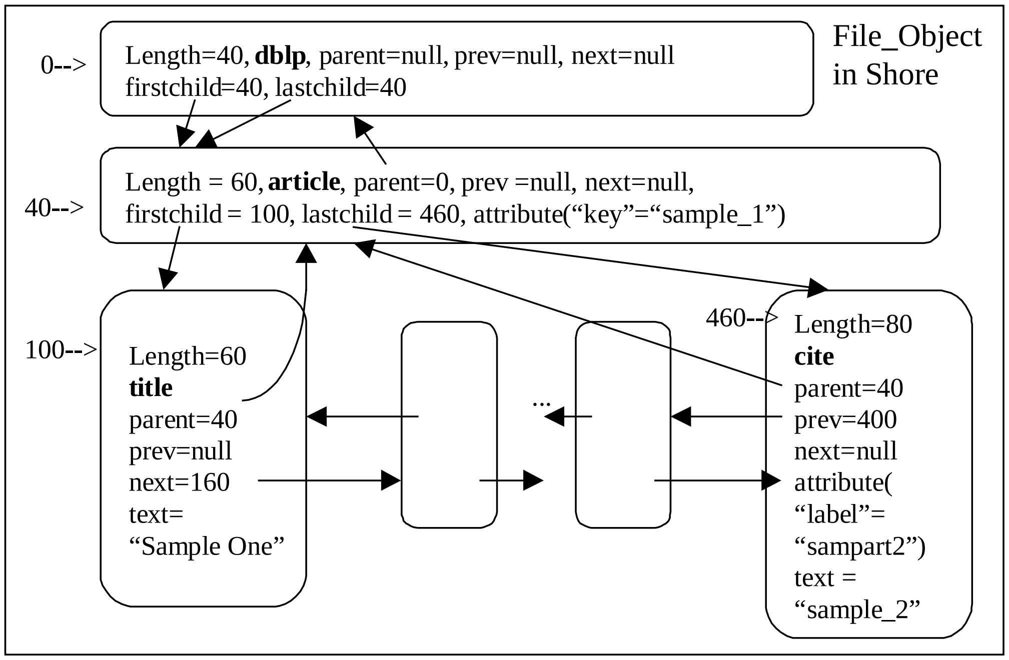
File_Object in Shore
0-->
Length=40, **dblp**, parent=null, prev=null, next=null firstchild=40, lastchild=40
40-->
Length = 60, **article**, parent=0, prev =null, next=null, firstchild = 100, lastchild = 460, attribute("key"="sample_1")
100-->
Length=60 **title** parent=40 prev=null next=160 text= "Sample One"
...
460-->
Length=80 **cite** parent=40 prev=400 next=null attribute("label"= "sampart2") text = "sample_2"

Figure 3.3 Object-based Approach.
Each circular box corresponds to a *lw_object*

The *tag* field is the tag name of the XML element. The *parent* field records the ele_id 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 ele_ids 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 build a Shore B-tree index that maps (*tag, opt_text)* to ele_id. 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 ele_id. This index is very helpful in retrieving all children of a node with a given tag.

When updating a *lw_object*, if the updated *lw_object* does not increase in size, the update is performed in place. However, if the updated *lw_object* is longer, the original *lw_object* is marked invalid, and then the updated element is appended to the *file_object*. All *lw_object*s that have links to this *lw_object* must also be updated. The linking information is updated in place. New *lw_object*s are always appended to the end of the *file_object*. Then the linking information of its parent node and siblings are updated as needed.

One should notice that if the parent originally has no children, this operation causes the parent node to grow, and thus the parent must also be moved to the end of the *file_object*.

3.3 B-tree Approach

There are several drawbacks of the object approach if the file needs to be frequently updated. First, since updated *lw_object*s that grow in size must be invalidated and then appended to the end of the *file_object*, the *file_object* tends to be fragmented, and space utilization deteriorates. Such updates also need to update linking information of other *lw_object*s. Implementing a map from logical ele_id to physical offset would eliminate this problem. In our third approach, we extend the object approach further by implementing a map from logical ele_id to the *lw_object* itself instead of to the physical offset of the *lw_object* inside the *file_object*.

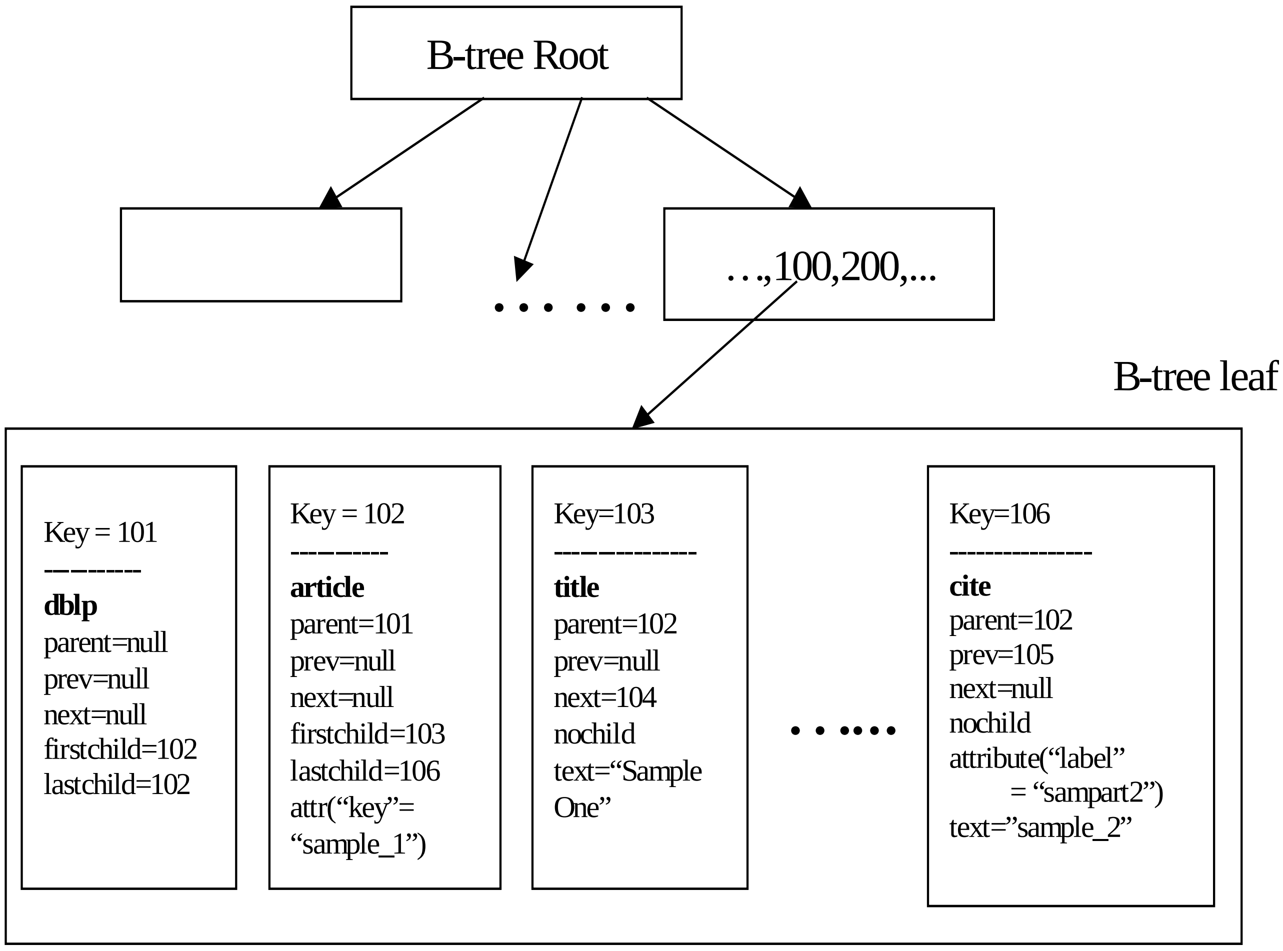

Figure 3.4 B-tree Approach.
(Only the leaf node corresponding to key value 100 is shown)

As with the relational 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 ele_ids and all elements of the DOM Tree are assigned an ele_id in this range in a depth-first order when the document is initially loaded into the database. This ele_id is used as the key of the B-tree. The data values in the leaves of the B-tree have the same format as the *lw_object* in the object approach. Figure 3.4 shows the example XML file inside a B-tree. When updating a *lw_object*, only that *lw_object* needs to be updated; the pointers from other *lw_object*s need not to be updated. 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 compared to the object approach. Instead of an offset lookup, we have to look up through the B-tree. Since this kind of look up is usually navigating from one node to another, the depth-first assignment of ele_id makes the overhead relatively small since we can expect the related element to be on either the same leaf node or a neighboring node. As with the object approach, we also build an index on (*tag, opt_text*) fields.

We also build a path index mapping (parent_id, tag) to child_id. Since XML documents are modeled as directed graph, most queries on XML data will generate a navigational workload from parent nodes to their children. In order to optimize this type of navigation, we store the path index clustered with the *lw_object* itself such that a single disk read will bring both the *lw_object* and the path index into memory. This is achieved by enhancing the B-tree key from (*id*) to (*id, tag, flag*). We use 1 bit in the flag to indicate whether the record in B-tree is a *lw_object* or a path index entry.

3.4 File System Object (FSO) Approach

In the future there are likely to be a large number of applications that do not require database services such as concurrency control and/or recovery which will want to store and query XML documents. Such applications will probably want to use the file system as their primary repository for XML documents. One simple solution is to store each XML document as an ASCII file and use an XML parser each time the XML file is accessed. This approach has two drawbacks. First, parsing a large XML file is expensive. Second, although the resulting DOM tree can be very large, it must be retained in its entirety in the application's address space that processes queries against the document.

The object approach described in Section 3.2 can be easily adapted to use the file system instead of an object management system like Shore (we term this approach FSO for "file system objects"). Each XML document is stored using the same format as the Shore object in the object approach. Applications access the object by using file system API. A *lw_object* is retrieved by using *seek* and *read* Unix system calls and only the part of the file corresponding to the desired *lw_object* is fetched into application address space. However, since the operating system does caching and pre-fetching using the file system buffer cache, each access to a *lw_object* does not necessarily imply a random disk I/O.

We also constructed an index mapping the (*tag, opt_text*) to ele_id and a path index mapping (parent_id, *tag*) to child_id using the B-tree implementation from Berkeley DB (libdb) toolkit.

4. Performance Study

In the section, we study the performance of the four storage strategies described in Section 3 plus an approach in which the XML documents are stored in ASCII files (labeled ASCII). We used the DBLP [DBLP] database as our test data set. The DBLP database contains about 60MB ASCII data. It contains about 140,000 bibliographical entries and 1.5 million XML elements in total. The original DBLP dataset has each bibliographical entry in a separate file, and is organized into multiple directories according to whether the publication is from a conference or a journal. The disk usage to store these small files exceeds 150MB. We combine about every 10 records in each directory into one XML file to achieve better disk space usage for ASCII approach. After being loaded into database, the amount of space consumed by each strategy is shown in Table 4.1. Notice that the data size of the B-Tree* approach includes both the data and the path index since the path index is in-lined in the B-tree.

	Edge	**Object**	**B-tree***	**FSO**	**ASCII**
Data Size	127MB	98MB	101MB	89MB	65MB
Index Size	109MB	114MB	94MB	118MB	

Table 4.1 Space Usages of Different Storage Strategies

We also constructed an index on all (tag, text) pairs. As shown in Table 4.1, the size of the index is fairly large. Alternatively, users could instead selectively build an index only on the most frequently queried tags. For example, in the DBLP database, perhaps it is worthwhile to only build an index on the author and title tags.

We conducted our experiments on a Pentium III 500 running Linux 2.2, with 256MB memory. We implemented the relational edge approach on DB2 and the other two approaches on Shore. Both DB2 and Shore were configured to use a 30MB memory buffer pool (the maximum shared memory segment size under Linux 2.2).

We also implemented the file system approach. ASCII files are first parsed into an in-memory DOM tree using IBM XML4C parser [XML4C], then queries are run over the in-memory DOM tree. There is no buffer pool with this approach, and the application uses as much physical memory as available (256M). The whole database is broken into multiple small files so that each parsed DOM tree will fit in memory.

The space occupied by each DOM tree is released as soon as a query no longer needs to access the DOM tree any more in order to minimize memory thrashing.

When appropriate, we conduct our experiments with both warm and cold memory. In order to measure performance from a cold start, we first shutdown DB2 or Shore and then flush the operating system buffer cache by doing a recursive "*grep*" on the /usr directory.

4.1 Experiment 1: Reconstruct Original Document

The first experiment reconstructs the original documents from the database and writes them to the file system. This experiment measures the performance of the various strategies under a typical navigational workload, since reconstructing the original document requires traversing from the root to all of its descendants. It is also a necessary step when constructing the answer to an XML-QL query. The results are shown in Table 4.2.

Warm					**Cold**				
Edge	**Object**	**B-Tree**	**FSO**	**ASCII**	**Edge**	**Object**	**B-Tree**	**FSO**	**ASCII**
335s	60s	127s	72s	250s	353s	63s	130s	74s	260s

Table 4.2 Performance of Reconstructing Document

As expected, the B-tree approach is slower than the object approach because each element lookup goes through the B-tree. The relational approach is the slowest, because the cost of fetching records from DB2 through the CLI interface is slower than performing a B-tree lookup in Shore. Linux does a fairly good job of caching for the FSO approach. We also evaluated parsing ASCII XML file into a DOM tree and printing out the DOM tree using IBM XML4C [XML4C] parser. We can observe that the execution time of storage manager approaches (object and B-Tree) are better than using a file system to store XML, while the RDBMS approach is slower. Taking the ASCII/Parser approach as a referential "benchmark", we conclude that the object and B-tree approach out perform ASCII file implementation in almost all applications except those that involve shipping the whole XML file statically. The RDBMS implementation will be slower in applications that require a traversal of DOM tree.

4.2 Experiment 2: Selection Queries

In the second experiment, we ran four XML-QL selection queries against the DBLP database. The XML-QL queries are manually translated to equivalent SQL queries for the Edge approach. The queries and their selectivity are

SQ_1: Select the authors for a given title. The number of results is 8.

```
<*><title>Mariposa: A Wide Area Distributed Database System
</title> <author>$a</author> </>
construct $a
```

SQ_2: Select all papers authored by Michael Stonebraker. The number of results is 154.

```
<*><title>$t</title>
<author>Michael Stonebraker</author></>
construct $t
```

SQ_3: Select all papers authored by Michael Stonebraker or Jim Gray. The number of results is 222.

```
<*><title>$t</title>
<author>$a</author></>
$a = ‘Michael Stonebraker’ or $a = ‘Jim Gray’
construct $t
```

SQ_4: Select all papers published between 1990 and 1994. The number of results is 40,189.

```
<*><title>$t</title> <year>$y</year></>
$y >= 1990 and $y < 1995
construct $t
```

We measure the time of executing the query and writing the results to a file on disk. Table 4.3 contains the results of these tests.

	Warm					**Cold**				
	Edge	**Object**	**B-Tree**	**FSO**	**ASCII**	**Edge**	**Object**	**B-Tree**	**FSO**	**ASCII**
SQ_1	0.01s	0.001s	0.002s	0.0007s	117s	0.38s	0.2s	0.16s	0.23s	126s
SQ_2	0.13s	0.03s	0.05s	0.028s	119s	2.2s	2.0s	1.7s	2.3s	128s
SQ_3	0.15s	0.04s	0.07s	0.03s	119s	2.9s	2.6s	2.0s	2.8s	127s
SQ_4	23.1s	9.1s	12.6s	6.1s	138s	40.5s	37s	27.8s	36s	157s

Table 4.3 Selection Query Performance Results

For the ASCII approach, the execution time is dominated by the time required to parse the XML files, which took about 123 seconds with a cold buffer cache and 114 seconds with a warm buffer cache. With a warm cache set up, we can see that object approach has the best performance. The FSO version performs better than the Shore version because it does not perform any locking. The B-tree approach has similar performance to that of the Object approach while the execution time of Edge is about two to five times that of the Object and B-tree approaches.

With a cold cache, the B-Tree approach performs better than Object approach. This is because the path index is in-lined with *lw_objects*. Bringing both the *lw_object*s and the path index from disk together compensates for the cost of a B-tree look-up. It is interesting to note that the performances of the Edge and object/B-tree approaches are much closer when the cache is cold.

With the Edge approach the XML-QL selection queries are translated to SQL join queries. The DB2 explain tool shows that the DB2 optimizer chooses an index nested loop join. Using an index nested loop join in fact is equivalent to navigation in the object approach using OIDs.

4.3 Experiment 3: Join Queries

For these experiments, we used three join queries.

JQ_1: Select all papers by Jim Gray that are quoted by Michael Stonebraker. The number of results is 5

```
<* key=$k><title>$t1</title><author>Jim Gray</author></>
<*><title>$t2</title>
<author>Michael Stonebraker <cite>$c</cite></>
$k = $c
construct $t1, $t2
```

JQ_2: Select all papers by Michael Stonebraker quoted by papers published between 1990 and 1994. The number of results is 614

```
<* key=$k><title>$t1</title>
<author>Michael Stonebraker </author></>
<*><title>$t2</title><year>$y</year><cite>$c</cite></>
$y>=1990 and $y<1995, $c=$k
construct $t1, $t2
```

JQ_3: Select all pairs of papers that cite one another. The number of results is 68,024. This kind of query is common in data mining applications.

```
<* key=$k><title>$t1</title></>
<*><title>$t2</title><cite>$c</cite></>
$k = $c
construct $t1, $t2
```

For the Object, FSO and B-tree approaches, we implemented JQ_1 and JQ_2 using an index nested loop join, and JQ_3 using a hash join. The hash table was built in user space and contains strings as hash keys and OIDs as hash values. It consumes only about 20 MB of memory. For the ASCII approach, all three join queries were implemented using the hash join algorithm since there is no index available. The execution time includes the time to write the results to disk. The performance results are contained in Table 4.4.

	Warm					**Cold**				
	Edge	**Object**	**B-Tree**	**FSO**	**ASCII**	**Edge**	**Object**	**B-Tree**	**FSO**	**ASCII**
JQ_1	0.57s	0.06s	0.09s	0.04s	119s	4.0s	4.1s	3.6s	3.7s	128s
JQ_2	3.2s	0.7s	0.9s	0.4s	124s	16.3s	10.1s	9.9s	10.3s	135s
JQ_3	196s	62s	70s	51s	55min	212s	71s	82s	74s	56min

Table 4.4 Join Query Performance Results

For JQ_1 and JQ_2, the ASCII approach is again dominated by the time to parse the two input documents. For JQ_3, the ASCII approach runs out of physical memory and starts thrashing. We can see that the performance of the Object and B-tree strategies is very similar when memory is cold while the Object and FSO approaches again perform best in the warm memory case. The DB2 optimizer chooses an index nested loop join on JQ_1 and JQ_2 and a sort-merge join on JQ_3. When memory is warm, the Object/B-tree approach is still two to five times better than the RDBMS Edge approach. Object/B-tree for JQ_2 and JQ_3 also show much better results than Edge when memory is cold.

4.4 Experiment 4: Update

We test update performance by inserting 1000 bibliographical record into the database. Each record consists of one title, two authors, five cites, one year, one URL and one cross-ref child. Thus a total of 12,000 elements corresponding to 420 KBytes of data are inserted into the database. The delete query deletes these records from the database.

	Edge	**Object**	**B-tree**
Insert	82s	67s	73s
Delete	80s	12s	14s

Table 4.5 Insertion and Deletion Performance Results

The insertion performance of the three approaches is very close. For deletion, the Object approach is fast because it only marks an element as invalid. The B-tree approach is also quite fast as only the unused space needs to be marked as available for use by subsequent insertions.

4.5 Discussion

These experiments have demonstrated that the object and B-tree strategies have similar performance across the entire range of queries. 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. While the performance of the Edge approach is usually two to five times slower than the other approaches, it was by far the easiest approach to implement. In addition to providing maturity and stability, using a relational database system has several other advantages including portability and scalability. In addition, since a majority 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. Given all these advantages, we believe that using a RDBMS will continue to be a viable option despite of its lower performance.

The performance of the relational approach depends on the effectiveness of the system’s query optimizer. Bad execution plan results can significantly degrade the performance. Semantically equivalent queries expressed as different SQL queries can cause relational database to choose different execution plans. Our results represent "correct plans" (most noticeably, choosing an indexed nested-loop join when navigating from one node to another), though not necessarily optimal. It is an open question whether or not a query optimizer would generally choose the correct plan in a ***real*** system. However, since the quality of commercial database system optimizers is fairly high, and since queries on XML documents exhibit predictable access patterns when the mapping schema is fixed, it should be possible to design and implement a *hint* mechanism indicating what mapping strategy has been employed. In the future we intend to explore the generation of optimal query plans given a mapping schema from XML data to relation tables.

Implementing the object approach using the file system (FSO) instead of the Shore storage manager has a number of advantages. First it avoids the overhead of locking and logging that the Shore implementation incurs. Second, since it uses the operating system buffer cache as its buffer pool, the overhead of buffer management is avoided. We expect that this approach will be useful for those applications that do not require concurrency control or recovery services and that are required to process relatively small collections of XML documents.

We have also run these queries against XML documents stored as ASCII XML files in the file system. This approach suffers from two significant problems. First, the time spent parsing the XML documents being queried dominates the total execution time for both selection and join queries. Second, when the parsed DOM tree exceeds the limit of the available physical memory, the system begins to thrash, as shown by JQ_3. This approach is clearly not a viable alternative.

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 storage manager and evaluated the performance of each strategy using a set of selection, join, and update queries. Our results clearly indicate that storing the XML documents as ASCII files is not a viable option for a query intensive environment. The results of our experiments also show that using an object manager to store XML documents provides the best overall performance, while using a relational database system has the worst performance. Generally the relational database system is about half or less as fast due to the overhead of the relational database layer above the storage manager. For certain applications this performance difference may be tolerable, especially since this approach makes it easy to support queries that span both XML documents and existing operational data sets stored in a relational DBMS. On the other hand, if one wants to build an XML database system with performance as the primary goal using an object manager is clearly the best.

Acknowledgments

Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908.

References

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

[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

[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

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

[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

[DBLP] DBLP maintained by M. Ley. http://www.informatik.uni-trier.de/~ley/db/index.html

[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

[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)

[OBS] M. Olson, K. Bostic, and M. Seltzer. *Berkeley DB.* Proceedings of the 1999 Summer Usenix Technical Conference, Monterey, California, June 1999

[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

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

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