Efficient XML Storage based on DTM for Read-oriented Workloads

Makoto Yui†, Jun Miyazaki†, Shunsuke Uemura‡, and Hirokazu Kato†

† Graduate School of Information Science, Nara Institute of Science and Technology
‡Faculty of Informatics, Nara Sangyo University
{makoto-y|miyazaki|uemura|kato}@is.naist.jp

Abstract

We propose an XML storage scheme based on Document Table Model (DTM) which expresses an XML document as a table form. When performing query processing on large scale XML data, XML storage schemes on secondary storage and their access methods greatly affect the entire performance. For this reason, we developed an XQuery processing scheme in which an XML document is internally represented as a set of DTM blocks and can be directly stored on secondary storage. Our scheme is tailored for read-oriented workloads, and an XML document is stored on disks as arrays of nodes. We analyzed the actual data access patterns to DTM appeared in processing XML queries, and employed the combination of informed prefetching and scan-resistant buffer management based on the analysis. Our experimental results showed that our storage scheme outperforms competing schemes with respect to I/O-intensive workloads, and our sophisticated prefetching and caching increase overall throughput without significant drawbacks.

1 Introduction

Recently increasing use of XML has heightened the need for storing and querying large amount of XML data efficiently. Previous researches have mainly been focused on indexing paths and optimizing XML queries. On the other hand, an underlying storage representation significantly impacts on XML query processing, and thus, it is important to explore storage schemes for XML documents.

Several storage schemes have been proposed for XML documents[14, 11, 10, 16]. However, to the best of our knowledge, there has been no careful study on the actual data access patterns of XML query processing (e.g., XQuery). It has still been an open issue as to which strategy is suitable for XQuery processing.

In designing our storage scheme, we made the following architectural decisions.

- Our scheme aims to be tailored for read-only and/or read-most workloads. This is based on the fact that XML databases are often required for managing existing XML documents received from other organizations (e.g., Electric Data Interchanges). In such situation, node-level updates (i.e., DML operations) are not always required. However, document-level updates (i.e., bulk insertions/deletions and document replacement) are considered to be required. Read-optimized database design has been suggested for relational database systems, such as [7], but not well studied for XML database systems. Therefore, we propose an efficient XML database system optimized for read-oriented workloads.
- We focus on iterative XQuery processing in which an operator tree consists of iterators. This is because many XQuery processors[5, 12], including our implementation, employ an iterator model and a *tuple-at-a-time* semantics instead of a *set-at-a-time* semantics.

Considering the above aspects, we propose an XML storage scheme based on Document Table Model (DTM) which expresses an XML document as a table form. A DTM table is internally represented as an array of DTM blocks, so that it can be directly stored on secondary storage. This straightforward approach enables effective prefetching of DTM blocks. However, it is known that there are interactions between prefetching and caching, and traditional cache replacement policies like LRU do not work well with prefetching[4]. It is because prefetched disk blocks need to be stored on the cache, and prefetched entries can potentially compete for (hot) cache entries. On the other hand, the benefit of prefetching and caching can coexist by using a scan-resistant cache replacement policy in certain situations[3]. To deal with this problem, we conducted the combination of informed (i.e., directed) prefetching and scan-resistant caching.

We have implemented a native XML database system, named *XBird*[1], using the proposed storage scheme. While XBird supports indices, in this paper, we focus on XML storage schemes and their access methods without indices

[1] XBird will be released at: http://db-www.naist.jp/~makoto-y/proj/xbird/

to evaluate the performance. Since XML queries require them for string-value calculation and serialization, the performance often depends on the basic access methods.

Our experimental results showed that our storage scheme outperforms competing schemes under I/O-intensive workloads, and our sophisticated prefetching and caching increase overall throughput without significant drawbacks.

The rest of the paper is organized as follows: in Section 2, we introduce a logical design of DTM. In Section 2.3, we give an analysis of data access patterns in XQuery processing. Section 3 presents our physical storage scheme and its access methods. In Section 4, we provide experimental results and their evaluation. We introduce related work in Section 5 and conclude in Section 6.

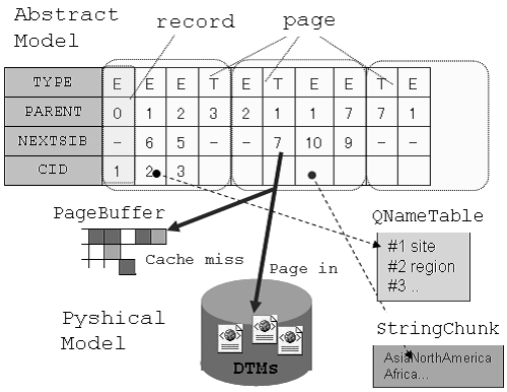

Figure 1. Internal organization of XBird

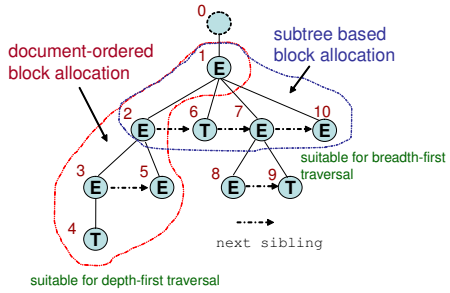

Figure 2. An XML tree labeled in depth-first order and its fragmentation examples

2 Logical Data Structure

2.1 Document Table Model

An XML document is represented as a variant of Document Table Model (DTM) in our proposed scheme. DTM was originally used in Apache Xalan XSLT processor[2]. It expresses an XML document as a table form, while previous Document Object Model (DOM) regards an XML tree as an object tree.

Since DTM table consists of primitive data types, DTM can avoid the footprint of objects, such as object instantiation and memory consumption, which are mandatory to DOM. Therefore, popular XQuery/XPath processors[12, 2] adopt either DTM or a similar internal data structure to DTM. However, these processors do not consider use of secondary storage to manage large scale XML data. To deal with this issue, we aim at a natural extention of DTM by taking account of secondary storage.

2.2 Internal Organization

Figure 1 shows an overview of our system organization. Figure 2 depicts an example of an XML tree labeled in depth-first search manner. In Figure 2, the symbols E and T indicate element nodes and text nodes, respectively. The upper half of Figure 1 represents the DTM associated with the XML tree illustrated in Figure 2.

A DTM table consists of four integer arrays (see Figure 1). An index of these arrays indicates a node handle, and thus, XBird requires 16 bytes per an XML node when an integer is 4 bytes long.

The first $TYPE$ row represents a node type, i.e., either E or T, and the following attributes: FIRST_ENTRY flag, LAST_ENTRY flag, HAS_CHILD flag, the number of namespace declarations, and the number of child elements. The FIRST_ENTRY and LAST_ENTRY flags are used to determine whether a sibling node exists on the left or right. The HAS_CHILD flag denotes whether the node has child node(s) or not. All these data are compacted and stored into an integer in a bitwise manner.

The second $PARENT$ and the third $NEXTSIB$ rows represent the index to parent nodes and that to the next sibling nodes, respectively. The fourth row keeps a content ID (CID) which indicates a unique identifier for QName and character string.

Character strings in XML data are converted into chunks. A CID is attached to each string, and then the strings are managed by the string management module, named *StringChunk*. If the length of a string exceeds the system threshold (512 bytes by default in our current implementation), the string is compressed by LZF algorithm[1] so that memory consumption can be suppressed.

QNames are managed in a unit of a collection which corresponds to the directory of file system to improve space efficiency. We call the QName management module *QNameTable*. A QNameTable generally fits in memory. Thus, it is mapped to memory while processing the corresponding collection.

2.3 Access to DTM

The access to a DTM table is based on operations to acquire a numerical value related to a specified node, such as

a CID or a parent value. For example, QName and character string are acquired from QNameTable and StringChunk, respectively, by using a CID as a key. Axis processing is based on offset calculation by referring to attributes such as parent and next-sibling values. Thus, the logical data structure, i.e., DTM, and its operations are equipped in the query processor.

All axis operations can easily be implemented by the combinations of the following five core functions in our DTM variant. The functions *firstChild*, *lastChild*, *nextSibling*, *parent*, and *previousSibling* are used to obtain youngest child, eldest child, next elder brother, parent, and younger brother, respectively. The algorithms of frequently used functions *firstChild*, *nextSibling*, and *parent* are shown in Figure 3. Using these functions, axis processing, such as parent, child, next-sibling, etc, can be implemented as a simple offset calculation.

Algorithm Algorithm of primary axis accesses

```
1. const BLOCKS_PER_NODE = 4;
2. const PARENT_OFFSET = 1;
3. const NEXTSIB_OFFSET = 2;
4. function firstChild(curnode) {
5.     code := getCol(curnode);
6.     if(!hasChild(code))
7.         return nil;
8.     namespaces := getNamespaceCount(code);
9.     attributes := getAttributeCount(code);
10.    addr := curnode + ((namespaces + attributes) + 1)
11.                * BLOCKS_PER_NODE;
12.    return addr;
13. }
14. function nextSibling(curnode) {
15.    return getCol(curnode + NEXTSIB_OFFSET);
16. }
17. function parent(curnode) {
18.    return getCol(curnode + PARENT_OFFSET);
19. }
```

Figure 3. Algorithm of primary axis accesses

Note that, in Figure 3, the function *getCol* acquires an array element specified by a node handle. The function *hasChild* judges whether the node has a child or not by using the bit flag. The functions *getNamespaceCount* and *getAttributeCount* acquire the number of namespace declarations and attributes, respectively. The constant BLOCKS_PER_NODE shows the number of array elements consumed by each node. The remaining constants PARENT_OFFSET and NEXTSIB_OFFSET represent the offsets to identify the location of the parent and next-sibling values, respectively.

In order to design an efficient XML storage and their access methods, we preliminarily analysed access patterns to DTM. The key issues to design the physical layout for effective XML query processing are:

- Mapping DTM to secondary storage for iterative XQuery processing, and
- Paging strategy between main memory and secondary storage.

Our solution to the former issue is to store nodes in document order, and for the second issue is to employ aggressive prefetching I/Os. Here after, we discuss these key issues by using the experimental results of 20 XQuery queries in XMark benchmark[13].

In the experiment, as we set page size to 2KB, a page has 512 DTM rows (see Figure 1). As for the experimental data, we used 113MB of the XML document generated by XMark where the scale factor (SF) is 1.

Figure 4 shows an overview of page access patterns of XMark queries. The inset figure within Figure 4 depicts the data access patterns of Q9 and Q10 both of which show peculiar page access patterns.

The vertical and horizontal axes of the figure denote the page ID that is to be accessed and time where accessing one page is a unit of time. Since page IDs are numbered in document order, the page with the lowest ID contains the document root and its neighbors, and the last node is allocated in the page with the highest ID. In other words, the document order is reflected on the vertical axis. Seeing in Figure 4, access patterns from lower left to upper right are a common tendency among 18 queries, which means that pages are requested in document order. Note that this sort of access pattern is not unique to XMark queries, but XBench[15] queries (both TC/SD and DC/SD) had also indicated similar access patterns (details are omitted due to lack of space).

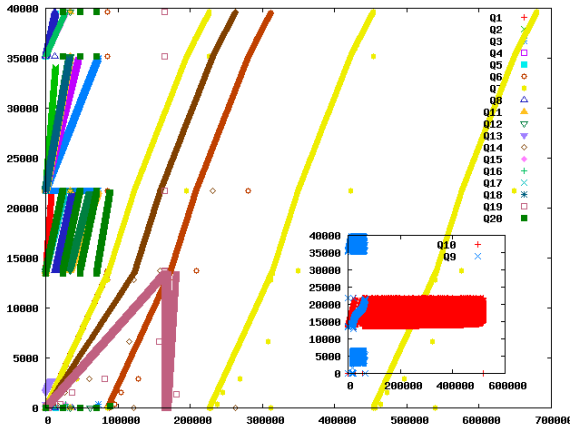

Figure 4. Page access patterns of XMark queries

2.4 Page Replacement Policy

Q9 contains a triple nested-loop, and thus, its locality of reference tends to be low. Therefore, we could not make the best use of the locality of reference, and it took long time to process Q9 due to paging overhead. A natural question is what algorithm is suitable for the buffer management of XML query processing. LRU is widely used as a buffer

replacement policy even in XML databases[11, 10]. However, it is known that LRU does not work well for sequential scans and large Inter-Reference Gaps (IRGs), while such sequential scans often appear in serialization and string-value calculations in XML query processing. Moreover, standard LRU is uncongenial to prefetching[4].

We compared the efficiency of LRU with that of scan-resistant 2Q[9] with prefetching. The result showed that 2Q brought 10% on average and a maximum of 23.5% better performance than LRU where SF is 10 for the XMark benchmark. Therefore, we use 2Q in XBird as a page replacement policy.

3 Physical Storage

3.1 Storage Scheme

XBird internally treats XML documents as DTM. If DTM is applied to the fundamental data structure of an XML database, it needs to be persistent. In order to have DTM persistent, it needs to be extended to secondary storage. A simple solution is to decompose the DTM into multiple blocks. Then, paging these blocks has only to be performed between main memory and secondary storage, as shown in the lower half of Figure 1.

In our proposed system, each block of DTM is stored on the secondary storage in document order. Here after, we call our proposed scheme *persistent-DTM*, *pDTM* for short, which follows well-known persistent-DOM (PDOM).

Since the block allocation policy of pDTM is based on document order, it does not always place adjacent nodes to the same or nearby block. A certain parent and its second or higher children might be taken apart on secondary storage. The reasons why we chose this allocation policy are as follows:

- According to the analysis in Section 2.3, access patterns in document order appear frequently.
- Node allocation in document order is suitable for string-value calculation and serialization which are mandatory for XML query processing.

Our XQuery processor employs an iterative query processing model based on an iterator tree, which is similar to BEA/XQRL XQuery processor[5] and Saxon[12] in which pipelined processing operators deal with loops, axis accesses, etc. Since queries are processed in operators in a tuple-at-a-time fashion, linear accesses in document order are found in Figure 4.

For example, during a query evaluation of 'site/regions' pipelined XQuery processor accesses to nodes are made in the following order: 'site[1]/regions[1]', 'site[1]/regions[2]', .., 'site[last()]/regions[last()]'. Thus, document-ordered storage model, i.e., pDTM model, fits for iterative query processing more efficiently than other strategies for most queries. In contrast, subtree-based storage model is not suitable for depth-first traversal (see Figure 2).

3.2 Physical Layout

The DTM table explained in Section 2.2 is internally formed as a two-dimensional array (more specifically, Iliffe vector[8]). The arrays of the second dimension consist of pages. Each element of the first dimension holds a pointer to each page. Since the pages that are not paged-in to main memory are expressed as *null* values, the skeleton, i.e., the first dimension, does not waste memory space.

The physical structure of DTM is divided into three layers as shown in Figure 5. A DTM row stores a record which is identical to a node information. A page is the minimum accessible data-unit of an I/O operation, which corresponds to a disk block.

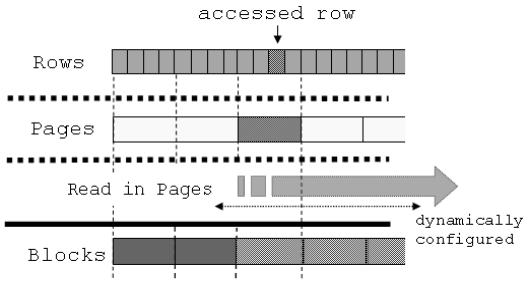

Figure 5. Physical structure of pDTM and its prefetching

3.3 Physical Access

A page-in operation is performed only when a requested DTM page does not exist in the page buffer (see Figure 1). For a simple implementation of paging, a hook is inserted just in front of *getCol* function, shown in Figure 3. The algorithm of the hook for paging is shown in Figure 6.

The following four paging profiles are currently defined. FORWARD profile is the default strategy that looks-ahead the forward direction. REVERSE profile is used for reverse-axis traversals which request nodes in reverse document order (e.g., preceding). INDEX profile is for index lookups. SERIALIZE profile is for retrievals of subtrees at serialization.

4 Experimental Evaluation

We implemented XBird in Java. In order to reveal the potential performance of XBird and compare it to competing schemes, we used the XMark benchmark suite[13] for evaluation. The experimental setting commonly used in this paper is as shown in Table 1.

Algorithm Page-in Algorithm

IN: rowId OUT: page

```
1. const PAGE_SHIFT := 9;
# following items of paging profile are dynamically configured
# by giving hints from the query processor.
2. readForwards := 32, readBackwards := 0;
# The hook to the getCol function.
3. pageAddress := rowId >> PAGE_SHIFT;(a)
4. page := PAGE_BUFFER.get(pageAddress);(b)
5. if(page == nil) page := readInPages(pageAddress);(c)
6. return page;
7. function readInPages(pageAddress)(d) {
8.    fromPage := pageAddress - readBackwards;
9.    toPage := pageAddress + readForwards;
10.   for(k:=fromPage; k<=toPage; k++) {
11.      page := readIn(k);
12.      if(i == pageAddress) requiredPage := page;
13.      PAGE_BUFFER.putIfNotFound(k, page);
14.   }
15.   return requiredPage;
16. }
```

a) Calculate a page address from the requested row ID.

b) Retrieve the requested page from the page buffer.

c) If 'page' value is *nil*, then call the function *readIn-Pages* to retrieve pages.

d) Retrieve pages from disk based on a paging profile. All pages that are read from disk are placed to the PAGING_BUFFER using a page address as a key.

Figure 6. Page-in Algorithm

CPU	Intel Pentium D 2.8GHz
OS	SuSE Linux 10.2 (Kernel 2.6.18)
RAM	2GB
Hard Disk	SATA 7200rpm
Java	Sun JDK 1.6
JVM option	-server -Xms1400m -Xmx1400m
Buffer size for DTM paging: 128MB	
Cache size used by StringChunk module: 32MB	
DTM page size	2KB
FORWARD	readForwards: 32, readBackwards: 0
SERIALIZE	readForwards: 64, readBackwards: 0

Table 1. Experimental setting

4.1 Comparison to Subtree-based scheme

In order to evaluate storage techniques themselves, we compare XBird with Natix[10](version 2.1.1) by using queries shown in Table 2 (These queries are introduced in XPathmark[6]). Natix is a native XML database system implemented in C++, and supports XPath 1.0. The unique feature of Natix is that it adopts a subtree-based storage block allocation strategy. XBird was configured not to use indices as to make it a fair comparison of XML storage methods, because Natix 2.1.1 does not have indices.

The summarized results where the SF is 5 and 10 are shown in Figure 7 and 8, respectively. Now, we focus on Q2-Q14 and Q17 for discussion.

An important difference appears in the results of Q6, Q7 and Q14 which contain '//'. Because '//' requests a lot of blocks in a depth-first manner, the efficiency of handling blocks, such as paging and buffer management, tends to appear remarkably. For Q2, Q5, Q14 and Q17 whose outputs are relatively large, pDTM achieves better performance than Natix because of the efficiency of serialization.

Natix shows a slightly good performance for Q4 which contains following-sibling axis, due to its subtree-based physical layout. Since child nodes are brought together in Natix, the following-sibling axis can be processed efficiently. Recall that subtree-based layout is suitable for breadth-first traversal of XML trees, but not for depth-first traversal.

5 Related Work

Only a few native XML storages have been studied. OrientStore[11] proposed a schema-guided storage method. Their strategy called Logical Partition-Based Clustering utilizes schema information, which clusters XML data into schema blocks to reduce I/Os required for path processing of XML queries. Though this storage technique is effective for path processing, it is not efficient for string-value calculation and serialization which require document ordering.

Natix[10] is a well-known native XML database which employs a subtree-based storage scheme. It divides an XML tree into subtrees based on the physical page size, so that each subtree fits into a page. Each page keeps the pre-order property of the subtrees on secondary storage.

Zhang et al. proposed a fast tree pattern matching algorithm, called *next-of-kin* (NoK) pattern matching, and a succinct XML string representation scheme, called *subject tree*[16]. In NoK, each page of subject trees is stored on secondary storage in the pre-order of XML trees. Though NoK supports simple parent-child queries, they have mentioned neither prefetching I/Os nor sophisticated buffer management.

On the other hand, non-native XML storages have been studied well, for example, in [14].

6 Conclusion and Future Work

In this paper, we proposed an efficient XML storage scheme based on DTM for iterative XQuery processing. We also analyzed access patterns that frequently appear for XML queries. Our experimental results showed that our storage scheme outperforms competing schemes in the certain situation where lots of pages are required such as queries contains '//' or when serialized results are relatively large. Furthermore, our enhancements (i.e., prefetching and scan-resistant buffer management) improved the performance of query processing by 10% on average and by 23.5% at maximum in our experiments. These results demonstrate the importance for XML database systems to take informed prefetching and scan-resistant caching into consideration.

Issues to be explored include realization of automatic database tunings such as buffer replacement policy and

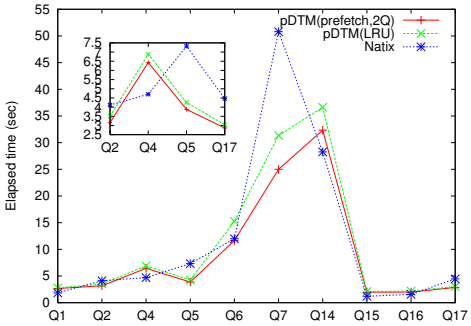

Figure 7. Performance of XMark SF=5

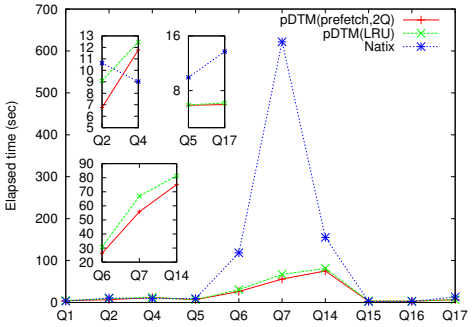

Figure 8. Performance of XMark SF=10

Query Id	Output size SF5 / SF10	Query expression
Q1	15 / 12	/site/people/person[@id = "person0"]/name/text()
Q2	241K / 482K	/site/open_auctions/open_auction/bidder[1]/increase/text()
Q4	0 / 0	/site/open_auctions/open_auction[bidder[personref/@person = "person20"]/following-sibling::bidder [personref/@person = "person51"]]/reserve/text()
Q5	666K / 1.3M	/site/closed_auctions/closed_auction[price/text() >= 40]/price
Q6	6 / 6	count(//site/regions//item)
Q7	6 / 6	count(/site//description | /site//annotation | /site//emailaddress)
Q14	149K / 297K	/site//item[contains(description, "gold")]/name/text()
Q15	42K / 80K	/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()
Q16	9.3K / 20K	/site/closed_auctions/closed_auction[annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()]/seller/@person
Q17	897K / 1.8M	/site/people/person[homepage/text()]/name/text()

Table 2. XPath queries converted from XMark queries

prefetching strategy based on the online analysis of access patterns.

Acknowledgements

We thank Matthias Brantner of Mannheim university for his help in carrying out the Natix evaluation. This base project is sponsored by Information-Technology Promotion Agency (IPA), Japan. This research was partly supported by CREST of JST and MEXT (Grant-in-Aids for Scientific Research on Priority Areas #19024058 and for Young Scientists (B) #19700094).

References

[1] lzf - an extremely fast/free compression/decompression-method. http://liblzf.plan9.de/.

[2] Apache Software Foundation. The Apache Xalan Project. http://xml.apache.org/xalan-j/.

[3] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proc. SIGMETRICS*, pages 157–168, 2005.

[4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proc. SIGMETRICS*, pages 188–197, 1995.

[5] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan. The bea streaming xquery processor. *VLDB Journal*, 13(3):294–315, 2004.

[6] M. Franceschet. Xpathmark - an xpath benchmark for xmark. Research report PP-2005-04, University of Amsterdam, Netherlands, 2005.

[7] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *Proc. VLDB*, pages 487–498, 2006.

[8] J. K. Iliffe. *The Use of The Genic System in Numerical Calculations*, volume 2 of *Annual Review in Automatic Programming*. 1961.

[9] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. VLDB*, pages 439–450, 1994.

[10] C.-C. Kanne and G. Moerkotte. Efficient storage of xml data. In *Proc. ICDE*, page 198, 2000.

[11] X. Meng, D. Luo, M.-L. Lee, and J. An. Orientstore: A schema based native xml storage system. In *Proc. VLDB*, pages 1057–1060, 2003.

[12] Saxonica Ltd. Saxon. http://www.saxonica.com/.

[13] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The xml benchmark project. Technical Report INS-R0103, CWI, 2001.

[14] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative xml storage strategies. *SIGMOD Record*, 31(1):5–10, 2002.

[15] B. B. Yao, M. T. Özsu, and N. Khandelwal. Xbench benchmark and performance testing of xml dbmss. In *Proc. ICDE*, 2004.

[16] N. Zhang, V. Kacholia, and M. T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in xml. In *Proc. ICDE*, pages 54–65, 2004.