MASS: A Multi-Axis Storage Structure for Large XML Documents

Kurt Deschler Elke Rundensteiner
Dept. of Computer Science
Worcester Polytechnic Institute
{desch,rundenst}@cs.wpi.edu

ABSTRACT

Effective indexing for XML must consider both the query requirements of the XPath language and the dynamic nature of XML. We introduce MASS, a Multiple Axis Storage Structure, to provide scalable indexing for XPath expressions with guaranteed update performance. We describe the components of MASS and how they together provide an efficient indexing solution for XML documents. Our experimental results demonstrate that MASS can outperform other state-of-the-art XML indexing solutions, even with constrained system resources.

Categories and Subject Descriptors
H.3.2 [Information Storage and Retrieval]: Information Storage - File organization

General Terms: Algorithms, Performance, Design

Keywords: XPath, XML, Indexing, Clustering, Encoding

1 INTRODUCTION

XML provides an attractive alternative to relational databases due to its expressive modeling power and versatility for representing data with diverse data structure. Achieving high performance for both queries and updates of XML data will be critical for the adoption of XML into many real-world applications.

Several structures have been proposed recently [4,12,13,16] to speed evaluation of path expressions. These structures use B-trees or hash indexing to accelerate path traversal for the child and descendant axes, often relying on large main-memory caches for performance. Only recently have XISS [2] and the XPath Accelerator [3] been proposed, to support evaluation of all Path axes. However, we find that these structures do not provide equal performance for all XPath axes, mainly due to their use of a single structural index.

Update performance is another challenging problem for XML databases. Existing proposals for XML indexing [2,3,12,13,16] have failed to demonstrate deterministic update performance, requiring significant portions of the index to be re-labeled upon insertion of a single document node. The root of this problem lies in the use of fixed length numerical quantities for encoding document order [6].

A side effect of extensively indexing XML is that the indexed data can be substantially larger than the original document. This adds to the already high degree of redundant tag data inherent in XML documents and presents a problem both in terms of disk space and cache hit rates. Compression techniques can exploit this redundancy to reduce the size of indexes [17]. However, the CPU-related costs incurred can easily outweigh the I/O savings.

As we demonstrate in this paper, the complex expressions that are possible in the XPath language [15] require a novel index structure for efficient evaluation. Furthermore, the orthogonal problem of facilitating index updates must be addressed before we can claim that a given index is a viable solution for real-world applications.

We propose a new structure called MASS (Multi-Axis Storage Structure) that provides an efficient means of evaluating all types of XPath expressions involving document structure, while also facilitating efficient updates. MASS introduces *FLEX keys*, *Node Clustering*, and *Cluster Compression* to address the aforementioned query, update, and size issues.

2 THE MASS INDEXING STRUCTURE

MASS is a highly integrated solution for indexing XML documents. Although each component of MASS can be applied separately to other indexing techniques, they have been designed as complementary pieces that integrate particularly well into one complete indexing solution.

2.1 FLEX Keys

We now propose a versatile organization for encoding document order called FLEX Keys (Fast Lexicographical Keys). FLEX Keys can be compared more efficiently than Dewey keys [11] and avoid the cost of re-labeling during incremental updates [2,3,6]. FLEX keys allow the application to determine node ordering, making them useful for establishing multiple orderings for document nodes. This is indeed the foundation for our clustering scheme presented in Section 2.2.

A FLEX key has a stepped organization where each ancestor from the root node is represented by a step. Instead of using numbers to represent order as in [2,3,11], FLEX keys use variable length byte strings that grow as needed to facilitate insertions without re-labeling. These byte strings can be compared efficiently using hardware-optimized memcmp() routines and sorted using radix algorithms [19].

FLEX keys can be compared to determine all relationships between nodes. The comparison properties of FLEX keys are useful for both for indexing and for filtering during intermediate query processing. The rules for comparing FLEX keys are as follows:

1. If the FLEX key for node *X* is a prefix of the FLEX key for node *Y*, then *X* is an ancestor of *Y*.
2. If two nodes have identical FLEX key ancestor components, then the nodes are siblings.

CIKM '03, November 3 8, 2003, New Orleans, Louisiana, USA.

3. If the longest prefix of the FLEX key of node *X* is equal to the FLEX of node *Y*, then Y is the parent of *X*
4. If components of FLEX keys are compared lexicographically, the lesser key is preceding in document order.

FLEX keys are constructed by first generating a byte string for each node that implies the correct ordering among siblings. This is then concatenated with the byte strings of all ancestor nodes using a delimiter that is smaller than all values used for keys. To facilitate incremental insertions, strings that terminate with the lowest or highest values in the alphabet are disallowed. This behavior is similar to the *Extended Prefix* scheme given in [6] for binary trees.

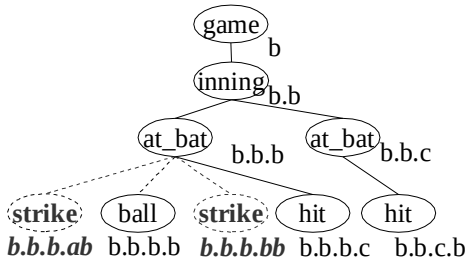

Figure 1: Incrementally Inserted FLEX Keys

Figure 1 Provides an example of FLEX key assignment including incrementally inserted nodes (shown in bold). Note that future incremental inserts will not produce longer strings since legal keys of the same length such as “b.b.b.ac” and “b.b.b.bc” are possible.

2.2 Clustered Organization

We now present the clustered encodings used by MASS to provide efficient evaluation of XPath node tests, position predicates and count aggregates for all XPath axes. The axis and node test combinations supported by each clustering are shown in Table 1. The ancestor and ancestor-of-self axes are supported using the inlined FLEX key data since these axes cannot be clustered efficiently.

Axis / Node Test	*“*”*	*not “*”*
descendant, descendant-or-self, preceding, following	CL1	CL3
child, following-sibling, preceding-sibling, attribute, namespace	CL2	CL4

Table 1: Mapping of Axis to Node Clusterings

Each clustering guarantees minimal I/O for each location step by ensuring that all nodes produced by a given axis, context node and node test are grouped together in adjacent index entries and returned in document order. Key compositions for the four MASS clustered indexes are given in Table 2. Examples of the CL2 and CL3 clusterings are shown in Figures 2 and 3, respectively.

Clustering	*Key Order*
CL1	Document
CL2	Sibling

Clustering	*Key Order*
CL3	Node Test, Document
CL4	Node Test, Sibling

Table 2: MASS Clusterings

MASS' clustered organization supports efficient evaluation of position predicates since ordinal positions correspond directly to index entries for every context node. Likewise, node set size can be determined efficiently by calculating the distance between two clustered entries. These fast counting capabilities are useful for query planning, yet are absent from several recent indexing proposals [2,3,12,16].

FLEX Key	*Node Type*
d	game
d.d	inning
d.d.a	id
d.d.d	at_bat
d.d.e	at_bat
d.d.f	at_bat
d.d.d.d	ball
d.d.d.e	strike
d.d.d.f	out
d.d.e.d	strike
d.d.e.e	hit
d.d.e.e.a	bases
d.d.f.d	ball
d.d.f.e	hit
d.d.f.e.a	bases

Figure 2: CL2 Clustering

Node Type	*FLEX Key*
at_bat	d.d.d
at_bat	d.d.e
at_bat	d.d.f
ball	d.d.d.d
ball	d.d.f.d
game	d
hit	d.d.e.e
hit	d.d.f.e
inning	d.d
out	d.d.d.f
strike	d.d.d.e
strike	d.d.e.d

Figure 3: CL3 Clustering

2.3 Indexing Large Documents

For large documents, where data is larger than the main memory, a paged index such as a B+ tree must be used to cluster and store data. We propose the use of ranking extensions [19] to extend the capabilities of the B+ tree to support MASS' query operations. The *ranked B+ Tree* facilitates the random access needed for position predicate evaluation with logarithmic I/O complexity. Furthermore, it facilitates distance calculation between any two index entries with logarithmic I/O complexity, which can be used to determine node set size for large (multi-page) node sets.

2.4 Cluster Compression

MASS introduces a simple yet effective scheme for compressing nodes called *Cluster Compression*. Cluster Compression exploits both XML tag data and flex key data that is redundant between adjacent index entries. Furthermore, it allows for compression and decompression of individual nodes to facilitate efficient queries and incremental updates. Unlike LZ and LZW encoders [8,9], there is no dictionary to maintain and offsets are stored only once with each piece of compressed data, rather than at each reference to the data. The number of nodes visited during compression and decompression of each node is bounded by the height of the document tree. Additional I/O is never incurred since compression chains never span index pages. Figure 4 demonstrates compression by a factor of five between two highly redundant adjacent entries.

Logical Index Entries

Entry#	FLEX KEY	PATH
1	d.d.e	/game/inning/at_bat
2	d.d.f	/game/inning/at_bat

Physical Index Entries

Ent#	Candidate	FLEX	Path	Size
1	-	[1]d [2]d [3]e	[1]game [2]inning [3]at_bat	19
2	1	[3]f		1

Figure 4: Cluster Compression Example

3 XPATH EXPRESSION EVALUATION

MASS facilitates efficient evaluation of location paths through iterative evaluation [5] of location steps. Iterative evaluation is very efficient using MASS since indexes are read sequentially for all XPath axes. MASS can be also be used in conjunction with structural joins [7] to facilitate efficient bottom-up and hybrid [5] evaluation strategies.

The following steps are performed internally by MASS to locate the node set for each location step.

1. Select the appropriate index clustering (CL1-CL4) from Table 1 using the axis and node test as selection criteria.
2. Compose the search keys used to locate the first and last node in the requested axis.
3. Locate the first and last nodes of the node set

Query operations are very efficient using MASS. Once the first and last node in the result are located, arbitrary ranges can be retrieved sequentially and node set counts can be calculated, both without additional key comparisons. Position predicates can be evaluated by simply advancing forward the desired number of nodes. By keeping the current index page in memory and comparing adjacent keys before performing a full binary search, MASS can perform iterative processing that is nearly as efficient as the merge style processing in [2,7].

4 DOCUMENT UPDATES

MASS allows for efficient incremental document updates since nodes can easily be individually inserted or removed from its compressed storage. Unlike previous proposals [2,3], an insert will never require relabeling other nodes. A new FLEX key can always be generated that is ordered between existing FLEX keys. Likewise, individual nodes can always be removed without relabeling other nodes.

5 EXPERIMENTAL RESULTS

We have implemented MASS in C++ and extensively tuned the implementation for optimal query performance. With the exception of the Xerces SAX Parser [10] used to parse XML input files, the entire implementation of MASS was done from scratch. This was done primarily to integrate the ranked B+ tree enhancements discussed in Section 2. We also implemented a custom storage manager that allows for strict control of buffer cache size.

5.1 Experimental Setup

Except where noted, tests were performed on a 333MHz Sun Ultra 10 with a buffer cache size of 256kb and 8k data pages. The OS buffer cache was bypassed using raw disk partitions or Solaris directio() (except where noted). Most of our experiments use documents of size 0.1MB to 100MB generated using the XMark generator [14].

5.2 Load/Compression Performance

Load performance was evaluated both in terms of time and space required. Documents of increasing size were loaded both with and without compression enabled. LZW compression was evaluated using UNIX *compress.*

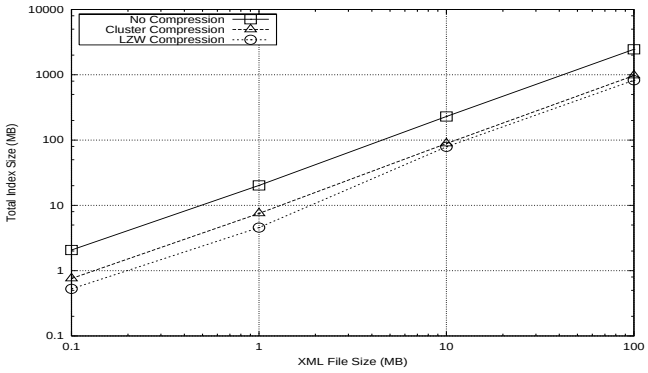

Figure 5: Index Size

The plot in Figure 5 shows that Cluster Compression decreases the index size by 70%, which is nearly as effective as the LZW compression for large documents. The reduced index size improved load times by up to 20%.

5.3 XMark Queries

Queries from the XMark benchmark [14] were used to evaluate the scalability of MASS for large documents.

XMark queries 2 and 3 demonstrate the performance of MASS' iterative evaluation and position predicate support. The plot in Figure 6 demonstrates the linear scale-up for these queries, which is expected due to the linear increase in result size.

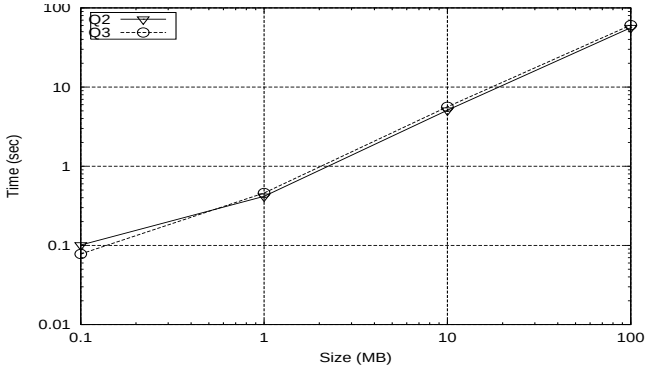

Figure 6: XMark Queries Q2, Q3

XMARK Queries 6 and 7 demonstrate performance of MASS' count aggregate evaluation. The logarithmic scale-up for these queries is demonstrated in Figure 7.

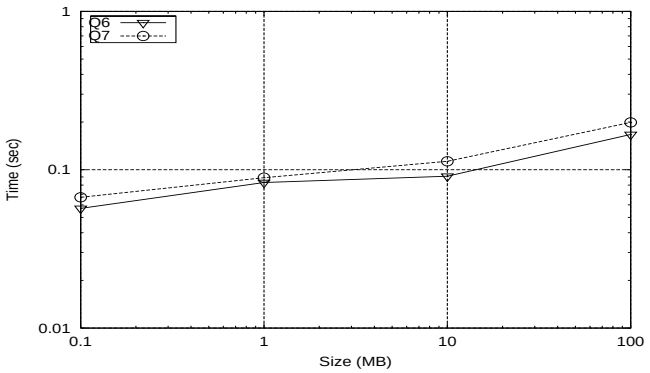

Figure 7: XMark Queries Q6, Q7

5.4 Update Performance

To measure MASS' update performance, we measured the time to add 100 bidders to an auction in the XMark data. Figure 8 demonstrates that the time for this fixed-size update scales logarithmically with document size.

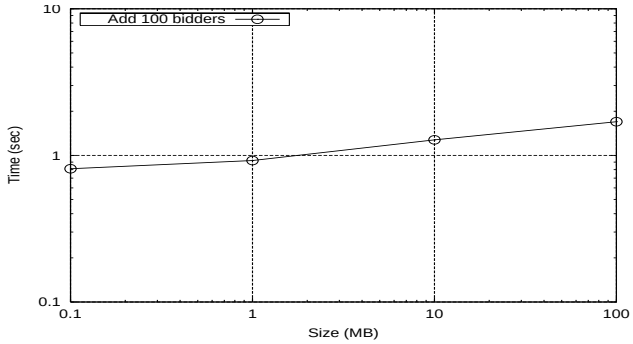

Figure 8: Incremental Update Performance

5.5 XPath Accelerator & XISS Comparison

To compare MASS against both XISS [2] and the XPath Accelerator [3], we loaded the XML version of Shakespeare's plays [18] into MASS and measured performance of the expression *"//act//speech"*. We performed this experiment on a Sun Ultra 2 using the directio() routine to bypass the Solaris buffer cache as in [2]. We then repeated the experiment without directio() to quantify the results of file caching.

System	***Time (sec)***
MASS with directio()	0.58
MASS without directio()	0.28
XISS	0.7
XPath Accelerator	1.15

Table 3: Comparison of Query: //act//speech

The results in Table 3 demonstrate that MASS outperforms both XISS and the XPath accelerator for queries on the descendants axis. This is particularly interesting since XISS and the XPath Accelerator are both optimized for queries of the descendants axis, whereas MASS does not favor any particular axis.

6 CONCLUSIONS AND OUTLOOK

The MASS indexing structure proposed here provides efficient Xpath indexing that readily supports document updates. Our experimental results demonstrate that MASS requires little system resources and scales well with document size.

We are currently building a complete XPath engine based on MASS to study query optimization and performance of complex expressions.

REFERENCES

1. MASS: A Multi-Axis Storage Structure for Large XML Documents WPI TR 03-23. ftp://ftp.cs.wpi.edu/pub/techreports/pdf/03-23.pdf
2. Q. Li and B. Moon: Indexing and Querying XML Data for Regular Path Expressions. VLDB 2001: 361-370
3. T. Grust: Accelerating XPath location steps. SIGMOD Conference 2002: 109-120
4. T. Milo and D. Suciu: Index Structures for Path Expressions. ICDT 1999: 277-295
5. J. McHugh and J. Widom: Query Optimization for XML. VLDB 1999: 315-326
6. E. Cohen, H. Kaplan, and T. Milo: Labeling Dynamic XML Trees. PODS 2002: 271-281
7. D. Srivastava, S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, and Y. Wu: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE 2002
8. J. Ziv and A. Lempel: A Universal Algorithm for Sequential Data Compression. IEEE Trans.actions on Information Theory 23(3): 337-343 (1977)
9. T. Welch: A Technique for High-Performance Data Compression. IEEE Computer 17(6): 8-19 (1984)
10. Xerces C++ parser. The Apache XML Project. http://xml.apache.org/xerces-c/index.html
11. I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang: Storing and querying ordered XML using a relational database system. SIGMOD Conference 2002: 204-215
12. B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon: A Fast Index for Semistructured Data. VLDB 2001: 341-350.
13. R. Goldman and J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB 1997: 436-445
14. Xmark-The XML-Benchmark Project. http://monetdb.cwi.nl/xml/index.html
15. J. Clark and S. DeRose. XML Path Language (XPath) http://www.w3.org/TR/xpath.html
16. C. Chung, J. Min, and K. Shim: APEX: an adaptive path index for XML data. SIGMOD Conference 2002: 121-132
17. H. Liefke and D. Suciu. XMILL: An Efficient Compressor for XML Data. SIGMOD Conference 2000: 153-164.
18. John Bosak. XML markup of Shakespeare's plays. http://www.ibiblio.org/pub/suninfo/ standards/xml/eg/.
19. D. Knuth: The Art of Computer Programming: Volume 3, Sorting And Searching. Addison-Wesley, 1973