WPI-CS-TR-02-23 June 2003

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

by

Kurt Deschler

Elke A. Rundensteiner

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's semistructured data model. This is particularly true for large documents, where query and update performance is governed by index efficiency.

We introduce MASS, a Multiple Axis Storage Structure, to provide scalable indexing for XPath expressions with guaranteed update performance. We describe the building blocks of MASS, namely, FLEX Keys, node clustering, and Cluster Compression. FLEX keys can be used to determine all node relationships while never requiring re-numbering. Node clustering guarantees scalable I/O performance for XPath node tests, positional predicates, and node-set aggregates, even with a small cache. Cluster Compression dynamically compresses both document data and FLEX keys to control data explosion while still supporting fast retrieval and incremental update of individual nodes.

We have implemented MASS in C++ and measured the performance of index materialization, query, and update operations. Our experimental evaluation illustrates that MASS scales well for a wide variety of query types as well as updates. When compared to other state-of-the-art XML indexing solutions, MASS can evaluate XPath expressions up to 7x faster, even with constrained system resources.

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.

The key to providing scalable performance for XML databases is robust and efficient storage and indexing. Lightweight applications, such as would be encountered on portable devices for example, require efficient indexing to process queries with little system memory. Likewise, data-intensive applications such as data warehouses require efficient indexing to reduce I/O for complex queries.

As we will demonstrate in this paper, the complex expressions that are possible in the XPath language [12] 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.

1.1 Indexing XPath Expressions

The XPath specification [12] defines 13 axes for navigation of XML document trees. There is no single index organization that can provide optimal performance for all XPath axes. Fortunately, the navigational patterns for some of the XPath axes are similar enough so that one single index can effectively support several axes. As an example, the *following-siblings* axis for a given node is a subset of its parent node's *child* axis.

In order to guarantee performance for ad-hoc queries on large documents, an indexing strategy must consider the many possible combinations of axis, node test, and predicates possible in each XPath location step. The importance of location step selectivity is illustrated by Figure 1. The first expression *"/at_bat/*"* must perform an efficient scan since it selects all children of the *at_bat* node, while the second expression *"/at_bat/ball[2]"* is very selective and returns only the second *ball* child of the *at_bat* node. We strive to develop indexing to support both classes of access patterns.

```
<game>
  <inning id="i1">
    <at_bat>
      <ball/>
      <strike/>
      <out/>
    </at_bat>
    <at_bat>
      <strike/>
      <hit bases="1"/>
    </at_bat>
    <at_bat>
      <ball/>
      <hit bases="2"/>
    </at_bat>
  </inning>
</game>
```

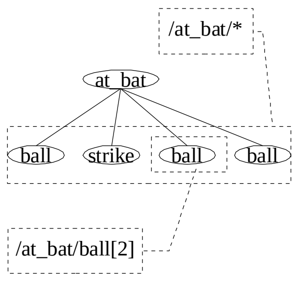


Figure 1: XPath Node Test and Predicate Selectivity

1.2 XPath Indexing Solutions

Many index structures have been proposed recently [12,16,18,19,20] to speed evaluation of path expressions.

These structures mainly use traditional B-tree or hash indexing to accelerate path traversal, with many of them relying on large main-memory caches for performance. Large performance improvements have been demonstrated using these structures, but only for expressions involving the child and descendants axes. More robust and general indexing will be needed to efficiently support XPath.

Only recently have two structures been proposed, namely the XPath Accelerator [12] and XISS [20], for supporting evaluation of all axes in XPath expressions. However, we find that these structures do not provide equal performance for all XPath axes, mainly due to their use of a single encoding. The efficiency of the XPath Accelerator depends on the structure's ability to narrow its multi-dimensional query window, which is not possible for all expressions. XISS is optimized for querying the descendant and child axes when the node test is not “*”, and does not appear to efficiently support the remaining axes.

The solution presented here provides equal and scalable I/O performance for all XPath axes. Furthermore, unlike the prior work, our solution also provides integrated indexing support for XPath Node tests, position predicates, and count aggregates. Furthermore, as required by the XPath specification [15], nodes are always returned in document order, independent of which axis was queried and the presence of selective predicates. This minimizes the need for time-consuming sorting in query processing. That is, even for both example queries in Figure 1 we will guarantee efficient ordered retrieval despite the difference in selectivity and in access patterns for the two expressions.

1.3 Update Performance

Update performance is another challenging problem for XML databases. Since XML documents are ordered, indexes must facilitate efficient updates to the ordering. Indexes must also be tolerant to structural variations and deep nesting inherent in XML data.

Existing proposals for XML indexing [12,16,18,19,20] have failed to demonstrate deterministic update performance, potentially 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]. Fixed length encodings work well when a document is initially labeled since nodes can be assigned increasing numeric values. However, incremental insertions will quickly run out of possible values since a numeric quantity can only be divided a finite number of times.

As we will demonstrate, the index structure presented here provides a novel solution for guaranteeing document update performance without ever requiring node relabeling and while readily facilitating irregularly structured data.

1.4 Index Size

A side effect of extensively indexing XML is that the indexed data can be substantially larger than the original document. This presents a problem both in terms of disk space and the overall percentage of document data that can remain in the database cache. Compression can be used to help reduce the size of indexes [17], hence reducing I/O. However, the CPU-related costs of decompression during queries can easily outweigh the savings due to reduced I/O while compression costs can reduce update performance.

We propose an organization for index pages that supports efficient compression and decompression of individual nodes and significantly reduces index size. Most importantly, our scheme allows for index pages to remain compressed when read into memory, increasing the amount of data in the cache and making decompression costs proportional to the amount of data actually accessed.

1.5 The MASS Approach

In this paper, 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 the following features to address the aforementioned query, update, and index size issues.

- **FLEX Keys** – A flexible scheme for encoding document structure and order that always allows insertion and deletion of nodes without renumbering and evaluation of all document relationships.
- **Node Clustering -** Facilitates efficient document ordered access and evaluation of proximity position predicates and count aggregates for all 13 XPath axes with both selective and non-selective location steps. This functionality is extended to large documents by exploiting B+ Tree ranking extensions to operate on node sets that span multiple disk pages.
- **Cluster Compression -** Exploits redundancy in MASS' clustered organization to facilitate high performance compression that is localized to individual disk pages to avoid the I/O costs of external lookups. This scheme allows compression and decompression of individual nodes and is shown to reduce the size of MASS indexes by 75%.

We have implemented MASS in C++ and evaluated performance using data from the XMark Benchmark [14] and Shakespeare's plays [21]. Our experimental results demonstrate the efficiency of MASS' clustered organization and compression, outperforming both the XPath Accelerator [12] and XISS [20] using identical queries. We also provide promising performance results for document loading and incremental update.

The remainder of this paper is structured as follows. Section 2 describes the various building blocks of MASS. Sections 3 and 4 explain query and update processing in MASS, respectively. Section 5 provides our performance measurements. Section 6 covers related work in XML indexing and Section 7 provides conclusions.

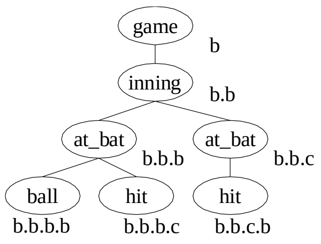
2 The MASS Indexing Structure

MASS is a highly integrated solution for indexing XML documents. Although each component of MASS, namely FLEX keys, Node Clustering, and Cluster Compression 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. FLEX keys also allow the application to determine node ordering, making them useful for establishing multiple document orderings. This is indeed the foundation for our clustering scheme as can be seen in Section 2.2.

A FLEX key has a stepped organization where each step corresponds to nodes in the document tree along the path from the root to that node. While similar to the Dewey encoding in spirit, we now propose to use variable length byte strings for each key step rather than numbers. FLEX keys can be compared efficiently using the memcmp() routine, which is optimized for specific hardware in most operating systems. Each byte string is generated such that it implies the correct ordering among siblings. A new FLEX key is constructed by appending this generated string to the vector of strings from the parent node's FLEX key.



FLEX Keys
b
b.b
b.b.b
b.b.b.b
b.b.b.c
b.b.c
b.b.c.b

Figure 2: FLEX Key Assignment

FLEX keys can be assigned with a single pass over the document by maintaining a single flex keys as the context. Steps are added or removed from the FLEX key at the start and end of each element, respectively. This is depicted in Algorithm 1. FLEX key assignment for the data from Figure 1 is demonstrated in Figure 2.

Algorithm 1: Labeling Algoritm

```
StartElement:
  Get next sibling key.
  Append sibling to FLEX key
  Copy FLEX Key to element
EndElement:
  Remove last FLEX key step
```

FLEX keys are a practical solution for labeling order in XML document trees that overcomes the problems inherent to numeric encodings. Fixed-length numeric encodings [11] can be compared efficiently, but fail to provide adequate support for incremental insertions, quickly running out of possible keys and requiring re-labeling. Variable length numeric encodings allow for incremental insertions, but cannot be compared efficiently (i.e. “10.1” is less than “2.1” when compared byte-by-byte). FLEX keys overcome these problems using generated character keys that can grow as required to facilitate insertions. This scheme guarantees that keys can be compared efficiently using memcmp() and that both insertions and deletions can be performed without re-labeling.

FLEX keys play a critical role in facilitating evaluation of the parent, ancestor, and ancestor-of-self axes. These axes cannot be efficiently indexed for all context nodes since the results between context nodes overlap in non-deterministic patterns. Numeric encodings can only be used to determine the parent and ancestor relationships between node pairs, requiring comparison of all document node pairs. Conversely, FLEX keys can be used to identify the ancestor or parent nodes directly by computing the prefixes of the FLEX key.

FLEX keys can be used to establish orderings other than document order by changing the order in which key steps of are compared. The order in which steps are compared depends on the desired sort order for nodes. For document ordering, steps are compared lexicographically, left to right. Other useful orderings are demonstrated later in this paper (as clusterings). Alternative orderings can be facilitated without copying by comparing individual steps of the FLEX key using multiple memcmp() comparisons. Alternately, steps can be copied to a new key that is byte-comparable if extra delimiters are inserted where the comparison order changes. The delimiter can be any character that is lexicographically smaller value than all keys in the key's alphabet. Byte-comparable keys can be compared using memcmp() and sorted using radix sort algorithms [4]. The example in Figure 3 demonstrates an encoding that groups keys with the same parent (prefix).

Original Key	*Reordered Key*	*Byte-Sorted Keys*
b.b	b..b	b..b
b.b.c	b.b..c	b..c
b.c	b..c	b.b..c

Figure 3: Alternate Node Ordering

New FLEX keys can always be generated and inserted into an ordered set without requiring modification of existing keys. The alphabet for byte strings can either be the character range (a-z) for more human readable keys or the full byte range (0-255) to minimize key length. In order to always allow insertions, we propose that FLEX keys disallow byte strings that terminate in the highest or lowest characters of their alphabet ('a' and 'z'). Once all keys of a given length have been exhausted, the length must increase for newly generated keys. Since two values are reserved for each byte, is possible to generate at most 253^n unique keys for a byte string of length *n bytes*. This behavior is similar to the *Extended Prefix* scheme given in [6] for binary trees.

An example of incrementally inserted FLEX keys is shown in Figure 4. The new nodes (shown in bold) are assigned the keys "b.b.b.ab" and "b.b.b.bb" rather than the more intuitive keys "b.b.b.a" and "b.b.b.ba", respectively. If the latter keys had been inserted, no further insertions immediately preceding the new nodes would be possible without relabeling. 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.

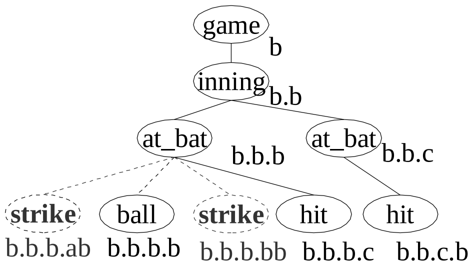


Figure 4: Incrementally Inserted FLEX Keys

FLEX keys can be compared to determine all relationships between nodes. The comparison properties of FLEX keys are useful for both filtering during node selection and 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.
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.

Algorithm 2: Lexicographical Sequence Generator

```
Inputs:
    $S_1$, $S_2$: strings $S_1 < S_2$
    UpperBound, LowerBound: lowest/highest byte values
Output: $S_3$, $L_3$: string $S_3$ of length $L_3$. $S_1 < S_2 < S_3$
set $S_3 \leftarrow$ CommonPrefix( $S_1$, $S_2$ )
set mchar $\leftarrow$ Average( UpperBound, LowerBound )
while( (Length($S_1$) < Length($S_2$))
   Append($S_1$, LowerBound)
while( (Length($S_2$) < Length($S_1$))
   Append($S_2$, UpperBound)
set MidByte $\leftarrow$ Average ( LastByte ( $S_1$ ), LastByte ( $S_2$ ) )
if ( MidByte != LastByte ( $S_1$ ) && MidByte != LastByte ( $S_2$ ))
    set LastByte ( $S_3$ ) $\leftarrow$ MidByte
else
    Append( $S_3$, mchar )
```

We present a simple algorithm for generating FLEX key byte strings in Algorithm 2. This algorithm uses a median approach to generate new keys (i.e. inputting “a” and “c” yields “b”). If a key must grow in length to be unique, the middle key from the alphabet is appended (i.e. inputting “b” and ”c” yields ”bm”).

The median algorithm from Algorithm 2 is effective when the number of sibling nodes is small and for small incremental inserts. However, if a node has many siblings, the length of generated byte string can grow rapidly. The worst case growth of Algorithm 2 is given as:

$$KeysPerByte = \log_2(256) = 8$$

$$Length = \frac{NKeys}{KeysPerByte} = \frac{NKeys}{8}$$

To prevent generating excessively long byte strings for bulk inserts, Algorithm 2 can be modified so that the length of new keys is doubled whenever all keys of shorter length have been exhausted. This modification bounds the length of strings for sequentially inserted nodes to:

$$NKeys = 254^{\frac{Length+1}{2}}$$

$$\log_{254}(NKeys) = \frac{Length+1}{2}$$

$$Length = 2 * \log_{254}(NKeys) - 1$$

2.2 Clustered Organization

We now propose a set of four clustered encodings that can be used to efficiently evaluate location steps for all XPath axes. Each clustering will be used to encode an index that services a subset of the XPath axes. The clusterings guarantee minimal I/O for location path evaluation by ensuring that all nodes produced by a given context node are grouped together in adjacent index entries and returned in document order. Furthermore, they facilitate efficient evaluation of XPath node tests, position predicates and count aggregates.

Our first proposed clustering (CL1) is document ordered. With this clustering, it is possible to efficiently traverse the axes which partition the document, namely the *descendant*, *descendant-or-self*, *preceding*, and *following* axes. Attribute nodes are excluded from the CL1 clustering since they are excluded from each of these axes. The exact steps for navigating the clustered organization will be described later. An example of a CL1 clustering is shown in Figure 5.

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

Figure 5: CL1 Clustering

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 6: CL2 Clustering

Our second proposed clustering (CL2) places sibling nodes in adjacent locations by comparing the parent ordering for each node first followed by the relative ordering among siblings. This clustering, demonstrated in

Figure 6, addresses traversal of the axes that traverse nodes with the same parent, namely the *child*, *following-sibling*, *preceding-sibling*, *attribute*, and *namespace* axes. Here we exploit the FLEX key property that sibling nodes have identical prefixes to identify the extent of each axis. Note that we cleverly use the reserved string “a” in the FLEX key for an attribute so that it is ordered before their containing node's children, as specified by XPath.

Our final clusterings, shown in Figures 7 and 8, cluster first by node type and then by the orderings defined for CL1 and CL2, respectively. These clusterings allow for efficient traversal of each axis while only visiting nodes of a given type. They address location steps where the node test is not "*", that is, the node type is specified as a constraint.

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 7: CL3 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
bases	d.d.e.e.a
bases	d.d.f.e.a
game	d
hit	d.d.e.e
hit	d.d.f.e
id	d.d.a
inning	d.d
out	d.d.d.f
strike	d.d.d.e
strike	d.d.e.d

Figure 8: CL4 Clustering

The mapping shown in Table 1 summarizes the rules used to select the appropriate clustering depending on the axis and node test of the location step being evaluated.

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

MASS' clustered organization also facilitates efficient evaluation of position predicates and count aggregates. This is possible since nodes for each axis are stored in physically adjacent locations. Position predicates can be evaluated by locating the first node in the axis, then advancing forward the desired number of nodes. Likewise, node counts can be determined by locating the first and last nodes in the axis, then calculating the number of entries between these two nodes. Inexpensive evaluation of count aggregates is not only useful for aggregation queries, but also in optimizing queries, where accurate statistics can be vital to effective query planning. Without clustering, these operations would require costly fetching and aggregation of entire axes.

It would be possible to generate a subset of the MASS clusterings to support a subset of the XPath axes. However, query processing is greatly simplified using the full set of clusterings since indexing is available regardless of which axis is queried or presence of a node test. Indexes that are not accessed will simply be paged out to disk and will not affect query performance.

2.3 Extending MASS Query Operations to Large Documents

The clustered organizations proposed for MASS facilitate efficient ordered retrieval and calculation of *position predicates* and *count aggregates* when data is in contiguous memory. However, for large documents, where we expect data to be dynamically paged to disk, additional support is required at the index level to support node sets that span across index pages.

Clearly, a B+ Tree would provide for efficient point and range access needed for locating context nodes and iterating over node sets. However, the traditional B+ Tree does not support random access to index entries and distance calculation between index entries, which are required for evaluating positional predicates and count aggregates respectively without scanning.

To facilitate the desired operations we propose the use of ranking extensions [4] to extend the capabilities of the B+ tree, which we will now refer to as the *ranked B+ Tree*. Since clustering is inherently ordered, an ordinal position *n* can be evaluated by finding the *n*th largest key in the clustering. The ranked B+ Tree allows the *n*th largest key to be determined with a logarithmic number of I/Os. The ranked B+ Tree also facilitates calculation of the distance between any two index entries with a logarithmic number of I/O operations. The ranked B+ Tree facilitates random positional access by maintaining subtree item counts in its interior pages. Each interior B+ Tree page has a counter for each individual subtree and a counter for the total number of items in all subtrees. Figure 9 provides an example of a ranked B+ tree that is ordered using the *CL1* Clustering.

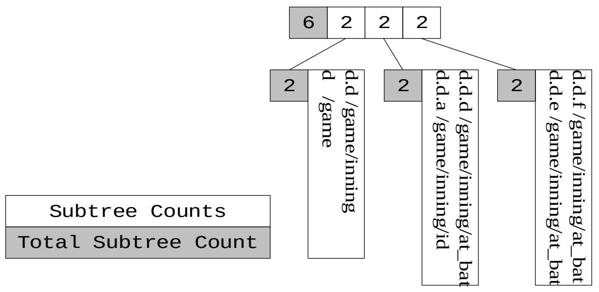


Figure 9: Ranked B+ Tree Organization

The query in Figure 10 demonstrates the effectiveness of the ranked B+ tree. Without random access to nodes, this query must scan all *at_bat* nodes to determine the 30th node. However, using the ranked B+ tree, it is possible to locate the correct data page and index entry with a logarithmic number of I/Os and key comparisons. The 30th *at_bat* node is located by searching for the first *at_bat* node in the CL4 clustering, then seeking forward 29 nodes using the ranked B+ Tree subtree counts. If the nodes span index pages, then only the pages containing the first and last nodes will be read.

```
//at_bat[30]
```

Figure 10: Query Accelerated by Ranked B+ Tree

2.4 Cluster Compression

We propose a simple yet effective scheme for compressing MASS' clustered indexes called *Cluster Compression*. One novelty of our Cluster Compression is that it exploits not only the redundancy between nodes in adjacent node entries that is inherent in clustered data, but also the high probability that any given node will have FLEX key components in common with its adjacent nodes. Cluster Compression allows for compression and decompression of individual nodes to facilitate efficient queries and incremental updates.

Cluster Compression shares both redundant FLEX key and path components with adjacent index entries on the same index page while storing non-redundant components locally. Figure 11 demonstrates cluster compression between two highly redundant adjacent entries. Each step of the FLEX key and path is compared with those of another node from the same page called the *compression candidate*. If the step does not match, both its depth and values are stored locally. Otherwise, its values are not stored and will instead be read from the compression candidate during decompression. For this example, the entry is compressed by a factor of five. Since offsets are only stored for the path components that are actually stored, entries for compressed entries are very small.

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 #	*Compression Candidate*	*Local Paths*	*Local Order*	*Size (bytes)*
1	-	[1]d [2]d [3]e	[1]game [2]inning [3]at_bat	19
2	1	[3]f		1

Figure 11: Physical Compression Representation

Cluster compression can provide compression ratios superior to both dictionary and prefix compressors for clustered data. Dictionary based compressors [9] require additional overhead to store dictionary offsets and control information that can be expensive for deep paths. As stated earlier, Cluster Compression does not need to store offsets for compressed data. Prefix compressors generally do not compress redundant data after the first mismatch and therefore provide limited compression opportunities. Cluster Compression can compress not only prefixes, but any redundant steps, making it much more effective than simple prefix compression.

To compress a node, an index entry from the same index page that shares common key components must first be located. We define the *match size* as the sum of the lengths of strings that are redundant between a *compression candidate* and the new node being inserted. The entry adjacent to the insert location with the largest match size is selected as the first *compression candidate*. In order to keep the maximum depth of the compression graph bounded, any link from the *compression candidate* to another entry must be followed until an entry having a smaller *match size* is found. If another entry having a larger *match size* is found, then it becomes the new *compression candidate*. Note that this search will not result in additional I/O since compression chains never span index pages. Once the ideal *compression candidate* is found, each component that is not redundant with the adjacent entry must be stored in the new entry along with its depth in the document tree.

The example in Figure 12 demonstrates selection of a compression candidate for a newly inserted node. Node b.b.b is chosen as the compression candidate since its match size is bigger than b.c.

	FLEX Key	*Path*	*Match Size*
Compression Candidate→	b.b.b	/game/at_bat/ball	12
New Node→	b.b.bn	/game/at_bat/strike	
	b.c	/game/at_bat	11

Figure 12: Compression Candidate Selection

Decompression is facilitated by following the compression chain, copying missing components from the current entry until all components for the path have been retrieved. Since each entry read will contain at least one path or order key component, and likely more, the worst case number of entries read is equal to the total number of key components. However, this worst case can only occur for a very limited number of entries since these combinations of entries will not be chosen once other entries are present that produce better compression. Recall that I/O will not be performed since compression chains do not span disk pages.

The potential for compression is excellent for all four MASS clusterings. Consider the clustering examples in Figures 5-8. The outer keys of each clustering key sequence determine which data is redundant in adjacent index entries. Adjacent *CL1* and *CL2* entries will have common FLEX key prefixes since these clusterings are primarily document ordered. Adjacent *CL3* and *CL4* entries will have common prefixes whenever there are several instances of a given node type, which is quite common in XML data.

2.5 Balanced Architecture

The combination of FLEX keys, Clustering, and Cluster Compression strike a perfect balance. FLEX keys readily facilitate MASS' clusterings. Clusterings groups together redundant FLEX keys, maximizing the effectiveness of Cluster Compression. In turn, Cluster Compression allows us to facilitate all four clusterings with reasonable disk requirements.

3 XPath Expression Evaluation

MASS facilitates efficient evaluation of location paths through iterative top-down evaluation [5] of location steps. Iterative evaluation is very efficient since our clusterings guarantee that indexes are read sequentially for all XPath axes. Since MASS produces node sets at each location step, it could also be used in conjunction with structural joins [22] to facilitate efficient bottom-up and hybrid [5] evaluation strategies.

To evaluate a location step, a context node, axis name and node test must be provided. Once these three pieces of information are provided, MASS can locate the node set in one of the clustered indexes. The following steps are performed internally by MASS to locate the node set corresponding to the supplied 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 using the generated keys. The remaining nodes will not be materialized unless they need to be fetched.

To demonstrate querying an axis using MASS, we will now consider the location step on line 1 of Figure 13, which selects all *hit* children. MASS first selects a clustering using the information in Table 1. *CL4* is selected since the axis is *child* and the node test is not "*". The search key for the first node in the axis, depicted on line 3 of Figure 13, is constructed by appending the node type to the absolute path of the context node and appending the lower bound key ('a') to the FLEX key of the context node. The search key for the last node, line 4, is constructed in a similar manner, except that the reserved upper bound key ('z') is appended to the FLEX key. MASS locates the first result node by searching the index corresponding to the *CL4* clustering for the first entry greater than the first child key. The last node is located by searching the same index for the first entry less than the last child key. The child relationship and node test are then checked using the node type and FLEX key for the first node to determine if any children of that type exist. For this example, d.d.e is the longest prefix of d.d.e.e and the child type is hit, so the node returned is in fact the first child of type *hit*. Since the first and last child are the same entry, there is exactly one node in the node set.

Location Step	`child:hit`
Context Node	`/game/inning/at_bat     [d.d.e]`
First Child Search Key	`/game/inning/at_bat/hit [d.d.e.a]`
Last Child Search Key	`/game/inning/at_bat/hit [d.d.e.z]`
First Child Found	`/game/inning/at_bat/hit [d.d.e.e]`
Last Child Found	`/game/inning/at_bat/hit [d.d.e.e]`

Figure 13: Child Axis Location Step Example

Query operations are very efficient in MASS. Once the two endpoints of a node set are located, no further searching is required. MASS can sequentially retrieve the remainder of the node set without additional key comparisons. MASS can also retrieve arbitrary ranges from the node set (to support position predicates) and calculate the node set size (for count aggregates) without performing additional key comparisons. Another key feature of our solution is that clusterings ensure that node sets will be returned in document order.

Figure 14 demonstrates the evaluation of a complete XPath expression. MASS is optimized to keep the current index page in memory and to compare adjacent keys before performing a full binary search. With these optimizations, iterative processing is as efficient as the merge style processing in [20], as confirmed by our experimental study.

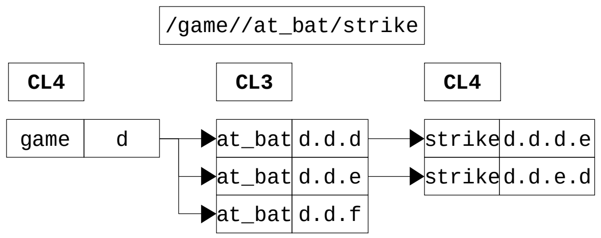


Figure 14: Expression Evaluation

4 Facilitating Document Updates Without Relabeling

MASS is inherently designed to allow for efficient incremental document updates since nodes can easily be individually inserted or removed. Unlike previous proposals [20,12], an insert will never require relabeling other nodes. Likewise, individual nodes can always be removed without relabeling other nodes.

4.1 Node Insertion

To insert a new node, the FLEX key must be determined for the new node. Nodes are always inserted after their parent node has been inserted, which is the pattern we expect when scanning an XML document. If the new node has existing siblings, the immediate sibling nodes must be retrieved so that their relative ordering keys are available as inputs to Algorithm 2. Otherwise, the upper bound or lower bound characters ('a' and 'z') are substituted as inputs to the algorithm. The FLEX key for the new node is formed by appending the relative order key for the new node obtained by Algorithm 2 to the FLEX key of the parent node. The node is then ready for insertion into the MASS indexes.

Insertion into the Ranked B+ tree is much like the traditional B+ tree [4], with the exception that subtree counts on interior index pages are updated. The cost of updating the subtree counts is negligible for large updates since the interior pages will generally remain in memory.

To achieve maximum page fill with our proposed per-node compression scheme, we have modified the traditional B+ tree insert algorithm to compress new nodes in temporary space, then only insert the node if there is indeed room on the page. If the entry does not fit, then the page is split and the new entry is re-created since any entries that it referenced for compression purposes may have moved. When an index page splits, all nodes from the split page must of course be re-inserted into the new pages. However, splits occur infrequently in B+ trees, especially with large data pages. This insert algorithm is shown in Figure 15.

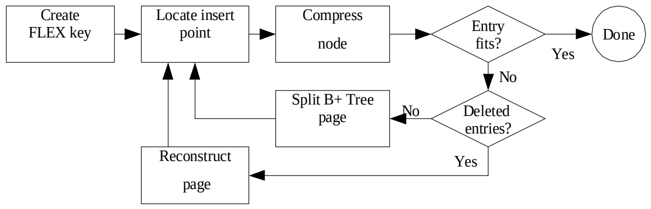


Figure 15: Modified B+ Tree Insert Algorithm

4.2 Node Deletion

FLEX keys allow nodes to be deleted without relabeling other nodes. Before a node can be removed, all descendants must be removed to avoid violating XML document semantics by creating orphaned nodes. This can be performed easily by using MASS to query the descendants axis. FLEX keys from deleted nodes can be re-used by subsequent inserts to the same locations to avoid creating longer keys with the same lexicographical ordering.

Space used by deleted nodes may not be reused immediately since it may be shared by many compressed nodes. Instead, the nodes can be logically deletes such that they do not appear in searches. Space used by these logically deleted entries can be recovered during inserts by re-creating pages that contain deleted items when these pages become full instead of splitting. Re-creating pages also has the benefits of defragmenting free space and optimizing compression for the remaining 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, since all structures are proprietary except the ranked B+ Tree. We have implemented our own ranked B+ tree mainly to include the enhancements discussed in Section 3 for reducing I/O and key comparisons for sequential fetching.

5.1 Experimental Setup

All tests were performed on a 333MHz Sun Ultra 10 with 256MB Ram. To control the amount of memory used for caching, we implemented a custom storage manager that has its own configurable buffer cache. We fixed the size of our buffer cache to 32kb for all queries. To avoid OS file caching, we ran all experiments (except where noted otherwise) on a raw disk partition

The page size for MASS indexes was fixed at 8k as this was found to be the smallest page size that could facilitate the large text nodes present in XMark data. This small page size allowed us to keep our cache size small and emphasize our I/O efficiency.

The majority of our experiments use data generated with the XML generator from the XMark benchmark [14]. We use this data set to demonstrate performance scalability of load, update, and several query operations. For each of these experiments, we increased the document size exponentially from 0.1MB to 100MB. We also use the Xmark data to compare performance with the XPath Accelerator. Our final experiment uses the XML version of Shakespeare's plays [21] to compare performance with XISS.

5.2 Load Performance

To evaluate Load performance, we loaded documents of increasing size and measured the elapsed time to load each document. We then repeated these experiments with compression turned off.

These measurements include parsing the document, sorting the document nodes for each clustering, load the four MASS indexes and flushing the indexes to disk. To help improve sort performance, the cache size for loads was set at 8MB for all loads. To facilitate efficient bulk loading, we have implemented a multi-phase paged radix sort that allows all indexes to be loaded sequentially.

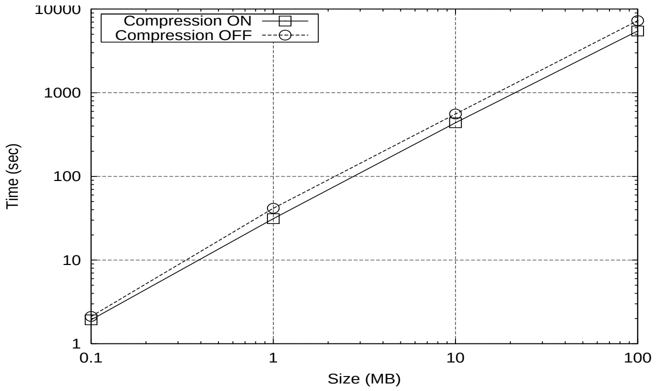


Figure 16: Load Performance

Figure 16 illustrates MASS' load performance both with and without Cluster Compression enabled. Load performance scales linearly with respect to document size even with a small cache. Due to the efficiency of our algorithm, load times improved by 20% with Cluster Compression turned on. This is especially apparent for the larger document sizes, where the size of the indexes impacts cache performance.

To evaluate the effectiveness of cluster compression, we compared the size of MASS indexes compressed using cluster compression with the size of indexes compressed using LZW compression [9]. Cluster compression was performed incrementally while each document was loaded. LZW compression was evaluated by disabling cluster compression and using the UNIX *compress* program to compress each index file after loading.

The plot in Figure 17 shows the combined size of all four MASS indexes with cluster compression, LZW compression, and without compression. With Cluster Compression, the index size is consistently 70% smaller than with no compression for all document sizes. Furthermore, the cluster compression is nearly as effective as the LZW compression for large documents. These results demonstrate the effectiveness of our Cluster Compression scheme.

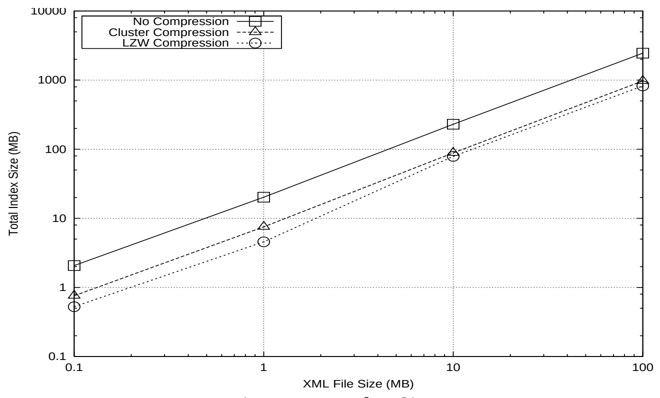


Figure 17: Index Size

5.3 XMark Queries

To evaluate performance of position predicates, count aggregates, and document reconstruction, we measured performance of five queries from the XMark [14] benchmark that test these operations with varying document sizes. While XMark queries only utilize the *child* and *descendants* axis, they demonstrate the scalability and performance of various MASS operations.

Query 2
FOR $b IN document("auction.xml")/site/open_auctions/open_auction
RETURN <increase> $b/bidder[1]/increase/text() </increase>

Query 3
FOR $b IN document("auction.xml")/site/open_auctions/open_auction
WHERE $b/bidder[0]/increase/text() * 2 <=
$b/bidder[last()]/increase/text()
RETURN <increase first=$b/bidder[0]/increase/text()
last=$b/bidder[last()]/increase/text()>

Figure 18: XMark Queries 2 and 3

XMark Queries 2 and 3, shown in Figure 18, test position predicates. MASS can efficiently evaluate position predicates due to its axis-specific clustering and Ranked B+ Tree. Unfortunately, Queries 2 and 3 do not use a large enough ordinal position to emphasize MASS' performance for these expressions, especially since the number of bidders for each auction is small. The plot in Figure 19 demonstrates the linear scale-up for these queries, which is expected due to the linear increase in result size.

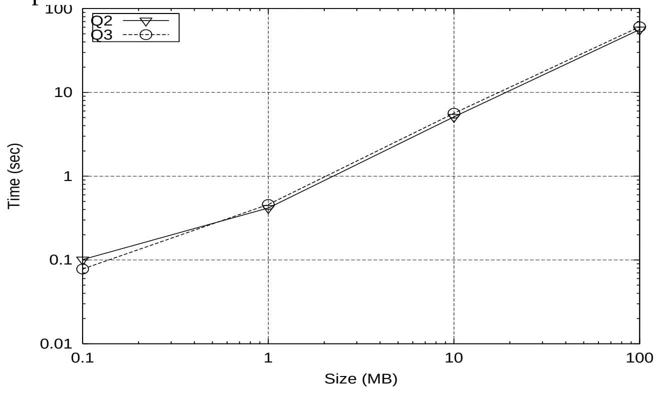


Figure 19: XMark Q2, Q3

Queries 6 and 7 of XMark stress the cost of counting node sets, which MASS can perform in logarithmic time. The logarithmic scale-up for queries 6 and 7 is demonstrated in Figure 21. These sub-second response times make the count aggregates practical for lightweight applications such as gathering statistics in the query planning phase.

Query 6
FOR $b IN document("auction.xml")/site/regions
RETURN COUNT ($b//item)

Query 7
FOR $p IN document("auction.xml")/site
LET $c1 := count($p//description), $c2 := count($p//mail)
, $c3 := count($p//email), $sum := $c1 + $c2 + $c3
RETURN $sum;

Figure 20: XMark Queries 6 and 7

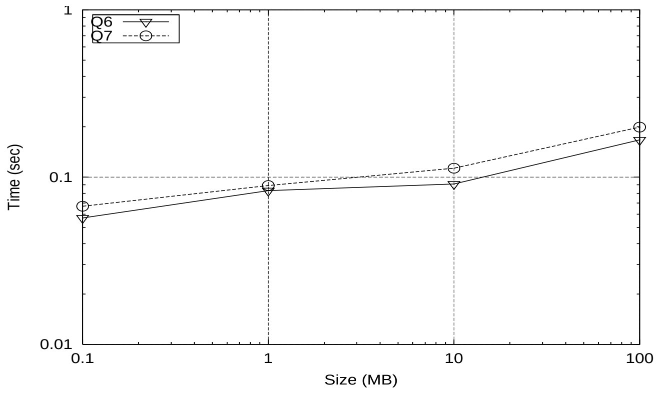


Figure 21: XMark Q6, Q7

5.4 Update Performance

To demonstrate MASS' efficient update performance, we have measured the performance of adding 100 bidders to an auction in the XMark data. Each bidder adds approximately ten new nodes to the document. Like the bulk load, nodes are sorted using a radix sort before being incrementally inserted.

The plot in Figure 22 demonstrates that performance of this fixed-sized incremental update scales logarithmically with document size. These times include the cost of sorting, inserting, and writing the updated indexes to disk.

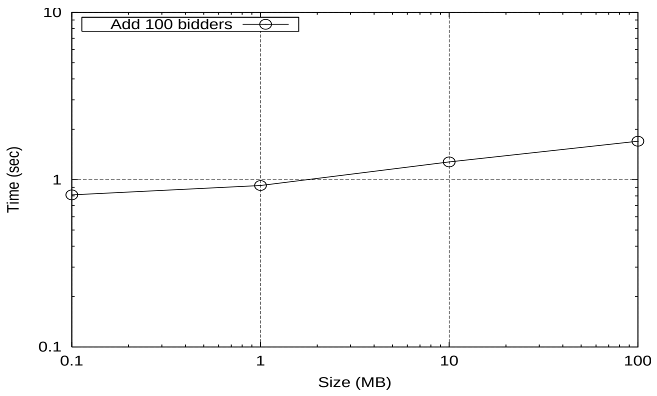


Figure 22: Incremental Update Performance

5.5 Comparison with XPath Accelerator and XISS

To compare MASS against results of the XPath Accelerator reported in [12], we measured the time to evaluate the expression "*//open_auction/description*" on the XMark. Although our machine was 3x slower (333MHz vs 1GHz), MASS was able to evaluate the expression 3-4x faster for the larger 10 and 100MB documents using raw disks. We also ran this query with MASS installed on an operating system file and recorded times 15-17x faster than the XPath Accelerator for the 10 and 100MB documents, respectively. Results for the XPath Accelerator were taken from [12]. These results, which are shown in Figure 23, demonstrate the superior I/O performance of the clustered MASS indexes in comparison to the XPath Accelerator.

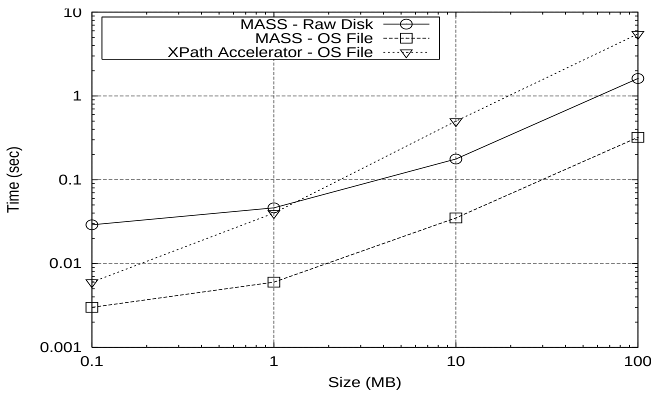


Figure 23: Comparison of MASS and Xpath Accelerator. Query: //open_auction//description)

To compare MASS against both XISS [20] and the XPath Accelerator [12], we loaded the XML version of Shakespeare's plays [21] 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 [20]. We then repeated the experiment without directio() to quantify the results of file caching.

System	***Time (sec)***
MASS with directio()	0.58
MASS w/o directio()	0.28
XISS	0.7
XPath Accelerator	1.15

Table 2: Comparison of MASS, XISS, and XPath Accelerator. Query: //act//speech

The results in Table 2 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 Related Work

6.1 Path Indexes

Many structures have been proposed for indexing regular path expressions. Earlier structures [2,3,5,13] were developed in the context of semistructured data. More recent proposals [12,16,19,20] discuss the problem in the context of XML. Until recently, research has focused almost exclusively on regular path expressions. However, regular path expressions are only a subset of the XPath language, representing the descendants and child axes. Many of the proposed path indexing structures [16,18,19] do not even address the issue of document ordering.

Several structures exist for facilitating fast traversal of document trees. The strong dataguide [18] stores full node paths to allow fast evaluation of prefix and fully qualified regular path expressions. The Index Fabric [19] also stores full paths, but in a more compact trie structure rather than a B-tree. A strong dataguide could be added to MASS simply by defining a clustering that is ordered by the full path of each node.

Manual and automatic tuning has also been proposed for further accelerating path traversals. The Index Fabric [19] allows manually refined paths to be inserted into its trie structure to accelerate specific queries. The system proposed in APEX [16] places frequently accessed paths in a large hash table to reduce edge lookups for such frequently traversed paths. However, paths not found in the hash table must be evaluated recursively by joining individual extents as before, which is inefficient for long paths.

To our knowledge, XISS [20] and the XPath Accelerator [12] are the only proposals thus far that provide extensive support for XPath expressions. XISS utilizes three indexing components to allow name-based lookups and structural joins to evaluate node relationships. The XPath Accelerator maps all node relationships to a single multi-dimensional structure, relying on structural constraints to narrow the query windows for different search dimensions. Since these systems do not focus on issues such as clustering data, they are vulnerable to thrashing when data sets do not fit in the cache and poor performance with a cold cache. We challenge future systems to evaluate performance under these stringent conditions.

Physical clustering has previously been considered, but not with regard to the XPath language as is done in MASS. A prior study of XML query optimization [5] noted that clustering would be useful for optimizing queries since I/O costs could be estimated accurately. The I/O patterns produced by MASS are very deterministic due to the extensive use of clustering. Another study [7] suggests that application-specific clustering would benefit query performance. MASS defines clusterings that are directly correlated to the XPath language, allowing efficient query evaluation without application-specific knowledge.

6.2 Labeling Schemes

Several labeling schemes have been proposed for encoding document ordering and structure in XML data. In [11], these schemes are described generically as either having global, local, or Dewey encoding. [11] suggests that Dewey encodings, such as used by MASS, are the best for a mixture of queries and updates since updates only require re-numbering nodes in the sibling axis, while global encodings such as that used in the XPath accelerator [12] require re-numbering all nodes in the following axis. Our proposal of FLEX keys now overcomes the shortcomings of the Dewey encoding. Namely, FLEX keys use generated character strings to avoiding re-numbering sibling nodes. The labeling scheme employed by XISS [20] can defer re-numbering of nodes by pre-allocating number ranges to allow for incremental insertions. The generated sequences used in FLEX keys could also be be deployed in place of the integer keys of the global ordering scheme to eliminate the need for re-numbering after incremental insertions.

6.3 Statistics and Query Processing

Another problem that has been largely ignored in the literature is index statistics for path expressions. Early research from the Lore database [13] demonstrates the need for accurate statistics in query processing. However, if we look at three recent proposals for XML indexing [16,19,20], none of them provide a cost model or meaningful statistics for a query processor. MASS is able to quickly provide the exact size of node sets so that the query processor can aggressively optimize complex expression queries.

6.4 Compression

Most XML data contains a limited number of node types that is constrained by either the DTD or the number of real world entities represented by the data. The limited number of node types guarantees some degree of redundancy and hence the possibility for compression.

Compression of XML data is covered extensively by XMILL [17]. XMILL compresses entire documents in batch, therefore provide near-ideal compression. Conversely, MASS compresses nodes as they are inserted, allowing for efficient incremental maintenance. While MASS follows on the concept of compressing redundant structural data from XMILL, it also applies compression to FLEX keys, thus alleviating the perceived space problem with the Dewey encoding [11].

Compared to the dictionary-based LZ compressors [8], Cluster Compression has several key advantages that make is more practical for the dynamic XML environment. Whereas LZ compression stores dictionary offsets for all entries that reference shared data, Cluster Compression only stores offsets for the shared data itself. Furthermore, the dictionary for the LZ encoder is difficult to maintain on fixed size data pages which must already manage variable sized data. LZW encoders are impractical for updates since new data cannot be compressed without re-building the compression dictionary.

7 Conclusions

In this paper, we proposed MASS, a Multiple Axis Storage Structure. MASS supports efficient evaluation of both selective and non-selective XPath expressions on all XPath axes using a set of four clustered indexes. MASS' clustered organization also allows for efficient evaluation of positional predicates and count aggregates. Nodes are guaranteed to be returned in document order, eliminating the need for sorting to order results.

MASS requires little system resources due to its clustering organization and use of B+ Tree ranking extensions. MASS exhibits excellent I/O performance with small caches and scales well with document size. Our experimental results demonstrate that MASS can outperform other path indexing structures even with constrained system resources.

MASS provides for efficient document update while preserving document order using FLEX keys. FLEX keys allow nodes to be incrementally added to or deleted from a document without requiring update of other nodes. FLEX keys can be used to determine all node relationships and uniquely identify nodes for query processing.

Path expressions cannot all be evaluated with a single index organization. MASS has taken the approach of supporting XPath more thoroughly, using a set of robust and efficient indexes.
In the future, we plan to integrate MASS into an XPath engine and study query optimizations such as using structural joins to facilitate alternate evaluation strategies.

References

1. D. Suciu. Semistructured Data and XML. AT&T Labs, Technical Report 1997.
2. J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data. Stanford University Technical Report, 1998.
3. T. Milo and D. Suciu: Index Structures for Path Expressions. ICDT 1999: 277-295
4. D. Knuth: The Art of Computer Programming: Volume 3, Sorting And Searching. Addison-Wesley, 1973
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. Florescu and D. Kossmann: Storing and Querying XML Data using an RDMBS. IEEE Data Engineering Bulletin 22(3): 27-34 (1999)
8. J. Ziv and A. Lempel: A Universal Algorithm for Sequential Data Compression. IEEE Transactions 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. T. Grust: Accelerating XPath location steps. SIGMOD Conference 2002: 109-120
13. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. Stanford University Technical Report, 1998.
14. Xmark-The XML-Benchmark Project. http://monetdb.cwi.nl/xml/index.html
15. J. Clark and S. DeRose. XML Path Language (XPath), Version 1.0W3C Recommendation November 1999. 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. R. Goldman and J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB 1997: 436-445
19. B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon: A Fast Index for Semistructured Data. VLDB 2001: 341-350.
20. Q. Li and B. Moon: Indexing and Querying XML Data for Regular Path Expressions. VLDB 2001: 361-370
21. John Bosak. XML markup of Shakespeare's plays. January 1998. http://www.ibiblio.org/pub/suninfo/ standards/xml/eg/.
22. 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