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

by

Kurt W. Deschler

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

_____

May 2002

APPROVED:

_____

Professor Elke A. Rundensteiner, Advisor

_____

Professor Micha Hofri, Head of Department

_____

Professor Carolina Ruiz, Reader

Abstract

Due to the wide acceptance of the Word Wide Web Consortium (W3C) XPath language specification, native indexing for XML is needed to support path expression queries efficiently. XPath describes the different document tree relationships that may be queried as a set of axes. Many recent proposals for XML indexing focus on accelerating only a small subset of expressions possible using these axes. In particular, queries by ordinal position and updates that alter document structure are not well supported. A more general indexing solution is needed that not only offers efficient evaluation of all of the XPath axes, but also allows for efficient document update.

We introduce MASS, a Multiple Axis Storage Structure, to meet the performance challenge posed by the XPath language. MASS is a storage and indexing solution for large XML documents that eliminates the need for external secondary storage. It is designed around the XPath language, providing efficient interfaces for evaluating all XPath axes. The clustered organization of MASS allows several different axes to be evaluated using the same index structure. The clustering, in conjunction with an internal compression mechanism exploiting specific XML characteristics, keep the size of the structure small which further aids efficiency. MASS introduces a versatile scheme for representing document node relationships that always allows for efficient updates. Finally, the integration of a ranked B+ tree allows MASS to efficiently evaluate XPath axes in large documents.

We have implemented MASS in C++ and measured the performance of many different XPath expressions and document updates. Our experimental evaluation illustrates that MASS exhibits excellent performance characteristics for both queries and updates and scales well to large documents, making it a practical solution for XML storage. In conjunction with text indexing, MASS provides a complete solution from XML indexing.

Acknowledgments

I would like to thank Dr. Elke Rundensteiner for her help and guidance in transforming MASS from a pile of ideas to a well structured system. I would also like the thank my finance, Melissa and my mother, father, and grandmother for their support and encouragement over my course of study.

Contents

List of Figures

List of Tables

1 Introduction

XML databases have several fundamental differences from their relational counterparts. Relational databases [14] require an a-priori defined schema that has been normalized and optimized to suit specific queries. The flat representation model found in relational databases requires a database designer to manually fragment the data they wish to model and then carefully indexed by a database administrator since little change to the schema is anticipated. Conversely, XML has a semistructured data model [1] where the structure is embedded in the document. The storage for the XML document must be derived from the document's structure. If the document has no Document Type Definition (DTD) or XML Schema, the structure of the document is not known until the document is parsed. These differences make XML storage a complex problem that requires a much more dynamic solution than relational databases can provide.

Effective storage for XML documents must consider both the dynamic nature of XML data and the stringent requirements of XQuery [16], the World Wide Web Consortium (WC3) proposed query language for XML. XQuery allows querying of documents both by the structure of the data and the data values themselves using the XPath [15] expressions. Indexing is thus required on both document structure and textual values in order to be able to evaluate many of these queries efficiently. Likewise, the storage must be efficient in adapting to changes in both the structure and data content of documents. Value-based indexing is an old problem with B-Trees and inverted indexes [22] being widely accepted as effective solutions. Structural indexing for XML documents, which is the focus of this research, is a new and difficult problem. A novel storage solution is required that can efficiently adapt to XML document structural changes while providing the specific interfaces need to

evaluate XPath expressions.

In this paper, we propose a new index structure called MASS (Multi-Axis storage Structure), that provides efficient means of evaluating XPath expressions involving document structure, while minimizing the cost of incremental updates. The main contributions of MASS are:

- A versatile system for encoding document order and structure based on lexicographical sequence keys. These keys can be used to determining all node relationships and guarantee insertion or deletion of new nodes in constant time. They can also be used to uniquely identify document nodes for tasks such as join processing or determining intersections between node sets.
- A compressed inlining scheme that provides full inlined paths for all document nodes while internally compressing path information. Individual nodes can be accessed during query processing without having to decompress other unrelated data. The compression is shown to reduce the size of the inlined data by up to 75%.
- A set of four key clusterings that allow efficient query of all XPath axes. The clusterings efficiently support different combinations of XPath axes, node tests, and predicates.
- Integration of a ranked B-Tree [22] to speed evaluation of sibling queries. The ranked B-Tree greatly reduced the number of key comparisons for range searches and can evaluate document position queries and count node sets in logarithmic time.

The remainder of this thesis is structured as follows. Section 2 provides a brief introduction to XPath expressions, while Section 3 explains the requirements for XPath indexing in detail. Sections 4 describe the various building blocks of MASS. Section 5 explains query processing in MASS, while section 6 explain updates

operations. Sections 7 provides results from the implementation or MASS and subsequent and performance measurement. Section 8 covers related work in XML indexing and section 9 provides conclusions.

2 XPath Expressions

The XPath language is used for addressing nodes in a document or matching sets of nodes by their relationships to a context node [15]. The expressions used to select nodes are called location paths. Location paths can select nodes by node type, tree relationships, unique ID, and value. Each location path is composed of one or more location steps. The first location step selects selects nodes with respect to the root node while subsequent steps select nodes with respect to the previous location step. The example in Figure 1 depicts a representative XML document (c) and annotated DTD (a) for an inning of a baseball game along with an XPath expression (b) with two location steps.

```
<?xml version="1.0"?>
<!DOCTYPE game [
<!ELEMENT game  (inning* )>
<!ELEMENT inning  (at_bat, at_bat, at_bat+ )>
<!ATTLIST inning id ID #REQUIRED >
<!ELEMENT at_bat  (ball | strike | hit | out )*>
<!ELEMENT ball  (#PCDATA )>
<!ELEMENT strike  (#PCDATA )>
<!ELEMENT hit EMPTY>
<!ATTLIST hit bases CDATA #IMPLIED>
<!ELEMENT out  (#PCDATA )>
]>
```

(a)

```
/game/at_bat[2]
```

(c)

```
<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>
```

(b)

Figure 1: (a) DTD, (b) XML Document, (c) XPath Expression

Each location step of an XPath expression consists of an axis, node test, and possibly predicates. The axis describes the tree relationship to the context node. The possible XPath axes are child, descendent, parent, ancestor, following-sibling, preceding-sibling, following, preceding, attribute, namespace, self, descendent-or-

self, and ancestor-or-self. The node test is either the type name of the nodes to select from the axis or "*" to select all nodes in the axis. Any additional predicates further refine the selection at the location step. In the previous example, the first location step has selected the game children nodes of the root, while the second step selected the 2nd at_bat child of each game node.

Figure 2 gives some more examples of XPath expressions with brief descriptions of what is queried. Note that there is a shorthand for some relationships, For example, "//" selects the descendant axis.

Expression	***Meaning***
`/game//at_bat[2]`	*2nd at-bat of the game*
`/game/inning/at_bat[hit]/..`	*at_bats that had a hit*
`/game/inning[9]/*`	*Statistics for the 9th inning*
`/game/inning[9]::following::*`	*After the 7th Inning Stretch*

Figure 2: XPath Expressions

There is currently no standard syntax for updating XML documents, although there will likely be soon since the need obviously exists. Hence, we will only discuss the insertion and deletion of nodes in this paper.

3 XML Storage Requirements

XML storage must not only provide efficient retrieval for queries, but must also adapt to the complex structural variation of XML data. Storage costs should remain proportional to document size, while update and query performance should be bounded for large documents.

The efficiency of a database is largely governed by the capabilities of its indexes. Even with very large documents, proper indexing should allow many queries to be evaluated with little system resources. Indexing capabilities should match those of the XPath language so that most XPath constructs can be evaluated efficiently.

Efficient XML storage should provide a replacement for the relational database

backends currently employed. Custom storage should be capable of evaluating many expressions faster than relational databases by avoiding costly joins and data fragmentation issues[10]. Likewise, document updates should be more efficient with native storage, especially when they alter the structure of the document.

3.1 Query Operations

To effectively support XPath expressions, indexing must be available to support each XPath axis. Efficient lookup should be available both with and without a node type in the node test. Additionally, since it is common for an ordinal position or an attribute to be supplied as a predicate, these should also be indexed.

The two examples in Figure 3 demonstrates queries with different node test and predicate combinations that have very different indexing requirements. The first expression, which selects the at_bat children of the inning nodes, requires indexing by node type for efficient evaluation. The 2nd expression, which selects children of inning nodes by position, requires indexing on the order of child elements for efficient evaluation.

```
/Game/Inning/at_bat
/Game/Inning/*[1]
```

Figure 3: Queries with Different Indexing Requirements

Indexing should provide useful statistics for XPath expressions. The search axes described in XPath [14] have inherently different selectivities which largely influence query performance. The *parent* and *ancestor* axes are relatively cheap to evaluate. Conversely, the *sibling*, *following,* and *descendant* axes can contain any number of nodes from zero to the total number of nodes in the document. It is important that the size of these axes can be determined accurately and efficiently when formulating a query plan. If these statistics are not accurate, then the query processor may have to

choose more conservative query plans such as document scans.

As an example, the queries in Figure 4 have different selectivities. The first query is not very selective for the XML document data given in Figure 1 since it retrieves all descendants of the inning node. The second query is much more selective since it returns at most one at_bat node for each inning node. However, without statistics on the number of innings, the actual selectivity remains unknown to the query processor.

```
Game/Inning[@ID="I1"]/*
Game/Inning/At-Bat[2]/Hit[1]
```

Figure 4: Queries with Different Selectivity

XML indexing should facilitate efficient expression evaluation on deeply nested data. Deeply nested data presents a problem analogous to multi-way joins in a relational database. When there are many intermediate results, queries become difficult to optimize and evaluate. A simple example is a fully qualified path as shown in Figure 5. Iteratively processing of the four location steps in this example could be very expensive since many nodes may be matched at each step.

```
/game/inning/at_bat/hit
```

Figure 5: Fully Qualified Path Expression

3.2 Lossless Storage

XML storage should eliminate the need for flat XML files. XML data stored in flat files is impractical for both queries and updates. For query processing, the flat file cannot facilitate any of the XPath axes. This is mainly due to the fact that the structure of the XML document is not known until the tags for each element are reached. Flat files can also be impractical for updates since operating systems generally do not support insertion and deletion from the middle of the file. Hence, files must be

completely re-written after updates.

Lossless XML storage must fully represent the document ordering, structure, and content, regardless of the size or complexity of the data. XML data consists of various node types including element, CDATA, and processing instructions. Lossless indexing must preserve both the node type and original node value to facilitate document reconstruction.

3.3 Document Order

XML storage must preserve document order and structure to facilitate document reconstruction. A flat XML file contains no explicit representation of document order or structure. Rather node relationships are only implied by the physical ordering of tags in the file. XML storage must implicitly or explicitly represent these relationships.

The representation for document order should be useful for query processing. Efficient access to nodes in document order is required for efficient document reconstruction. Furthermore, since the result of a path expression must be in document order [16], it is also important that document ordering can be maintained efficiently during query processing. Indexes should provide document ordering whenever possible to minimize the need for sorting after each location step. Additionally, it should be efficient to compare ordering between individual elements for the case when sorting is needed to order a result correctly.

An effective solution for document ordering should also allow for efficient insertion and removal of individual document nodes. Document ordering must be managed efficiently since there may be an indeterminate number of elements whose ordinal position may be affected by an update. XML storage should also allow for efficient insertion of elements that cause structural change, such as removing all

nodes of a given type or adding a new type of node.

4 The MASS Indexing Structure

We propose MASS as an index structure to meet the stringent requirements for XML indexing outlined in Section 3. MASS is a native XML indexing structure that facilitates efficient querying of all XPath axes and allows efficient document update. Furthermore, the lossless storage in MASS enable it to serve as primary XML storage, eliminating the need for flat files.

MASS encapsulates five different concepts that together satisfy the requirements for query processing, update, and lossless storage. The concepts are inlining, ordering, compression, clustering, and persistent storage. Performance is the key consideration for the entire design of MASS. The result is an effective and scalable solution for XML path indexing. We will now examine the five key concepts behind MASS one by one in detail.

4.1 Inlined Paths

Inlining is a well known technique that reduces the cost of traversing a document graph by storing multiple edges of the graph together in a concatenated form. Inlined paths can also be used to determine node types for ancestor nodes without having to access the ancestor nodes directly. The example in Figure 6 demonstrates the inlined expansion of a document graph.

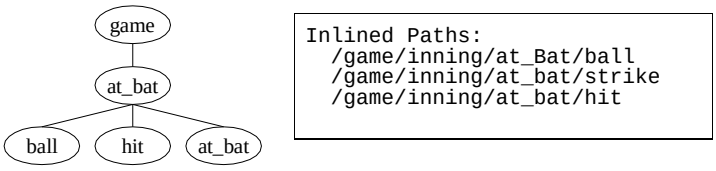


Figure 6: Example of Inlined Data

Prior work [4,7,10] has repeatedly shown that inlined paths can significantly reduce the cost of partial and fully qualified path expressions by reducing the number

of index I/Os. Figure 7 demonstrates queries that can be evaluated efficiently by comparing the path expression with the inlined expansions from Figure 6. The second example is considered a *partial match* since the node test "*" matches all node types.

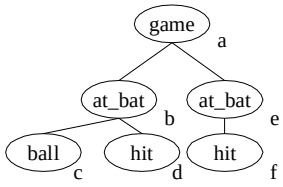
```
What were all the hits in the game?
  /game/inning/at_bat/hit

What happened in all of the at_bats?
  /game/inning/at_bat/*
```

Figure 7: Querying for Inlined Paths

Inlined paths alone have limited effectiveness since they can only test node types. For example, the nodes returned from the expressions in Figure 7 may be siblings, but this information is not present in the inlined paths. A solution is needed that also captures the structural information from the document.

4.2 Inlined Order Keys

We first look at a simple numbering scheme. In the example in Figure 8, a pre-order traversal is used to traverse the document graph in document order and assign a number to each node. This simple numbering scheme is of limited use to queries since the relationships between nodes cannot be determined without examining the entire set of nodes. A more robust solution is clearly needed to satisfy our XPath query requirements.

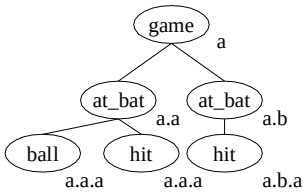


Order	Path
a	game
b	game/at_bat
c	game/at_bat/ball
d	game/at_bat/hit
e	game/at_bat
f	game/at_bat/hit

Figure 8: Simple Ordering Example

To encode document structure in MASS, we propose a scheme that complements inlined paths with *inlined order keys*. For each document node a relative key is assigned that describes the order with respect to sibling nodes. This relative key is

concatenated with the relative keys of all ancestor nodes to form the inlined order key. This assumes that the ancestor nodes have already been inserted. Note that since the parent node has the same key organization, the concatenation of all ancestor keys is equivalent to the inlined order key of the parent node. The example in Figure 9 demonstrates inlined order keys



Inlined Order Key	*Inlined Path*
a	game
a.a	game/at_bat
a.a.a	game/at_bat/ball
a.a.b	game/at_bat/hit
a.b	game/at_bat
a.b.a	game/at_bat/hit

Figure 9: Inlined Order Key Example

The inlined order key for any two nodes can be compared to determine all of the axis relationships between the nodes. Let us review this with a few examples. If the inlined order key for one node is a prefix of the inlined order key for another node, then it is an ancestor of that node. If the longest prefix of the inlined order keys for two nodes is identical, then the nodes are siblings. If the longest prefix of the inlined order key for one node is equal to the inlined order key of another node, then it is the parent of that node. If the inlined order key for one node is lexicographically greater than that of another node, then it precedes that node in document order.

The inlined order key also serves as a unique identifier for document nodes. The relative portion of an inlined order key for a given node is unique among the siblings of that node. Inlined order keys are also unique across all documents since the component of the inlined order key that corresponds to the root node is unique for each document. The uniqueness of these keys allows search for individual nodes by exact match.

Inlined order keys are also useful for intermediate query processing. The example in Figure 10 demonstrates evaluation of an expression containing the

disjunction *"strike or foul"* using the date from Figure 9. After separately evaluating the two sides of the disjunction, a single node set in document order must be produced from the union of these two results. The inlined order key can be used to order the result in document order and to determine if there are duplicates in the union that must be removed. If both sides of the disjunction arrive in document order, then a single merge can be used to perform both of these tasks efficiently.

```
Which At-Bats had either a hit or a ball?
  /game/inning/at_bat/*[hit or ball]/..

(1)At-bats with hits: (in document order)
  /game/inning/at-bat      [a.a]
  /game/inning/at-bat      [a.b]

(2)At-bats with balls: (in document order)
  /game/inning/at-bat      [a.a]

Calculate Union of results (1) and (2):
  /game/inning/at_bat/foul [a.a]
  /game/inning/at_bat/foul [a.a]
```

Figure 10: Inlined Order Key Used to Evaluate Disjunction

The ancestor information stored with each document node can be used to evaluate partial match expressions, establish document ordering, and check node relationships without actually retrieving the ancestors of a node. The I/O saved by not retrieving the ancestors can substantially improve performance of these queries. The example in Figure 11 demonstrates this optimization.

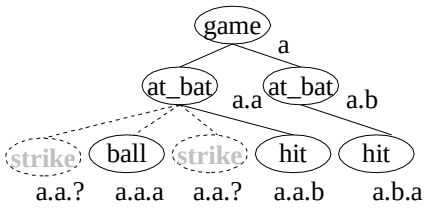
```
Was At-bat was the last strike thrown in?
  /Game/Inning/At-Bat/Strike[last()]/..

Intermediate Result:
  /Game/Inning/At-Bat/Strike [B.B.C.B]

Truncate Path and order to get parent node:
  /Game/Inning/At-Bat        [B.B.C]
```

Figure 11: Retrieving Specific Ancestors from Inlined Path

4.3 Lexicographical Sequences

Although the inlined order key scheme described in Section 4.2 works well with queries, it does not allow for efficient updates. This problem is illustrated in Figure 12. A new node cannot be added to the left of node "a.a.a" since a relative key with

value less than 'a' cannot be generated. A new node cannot be added to the right of node “a.a.a” since a relative key that is greater than 'a' and less than 'b' cannot be generated.



Inlined Order Key	Inlined Path
a	game
a.a	game/at_bat
a.a.a	game/at_bat/ball
a.a.b	game/at_bat/hit
a.b	game/at_bat
a.b.a	game/at_bat/hit

Figure 12: Document Update Problem

We propose generated lexicographical sequences to facilitate node insertions and deletions in MASS. Generated lexicographical sequences are variable length byte strings whose values are lexicographically ordered. Strings are compared character by character unless one string is the prefix of the other, in which case the shorter string has a smaller value.

The algorithm used to generate lexicographical sequences guarantees that lower and higher keys can always be generated. To ensure this, we define a lower bound and upper bound for the range of characters that are legal for use in keys. These limits are set to the alphabetic characters 'a' and 'z' in this paper. Neither of these characters will ever be generated by the algorithm since that would prevent the subsequent generation of smaller or larger keys respectively. To generate a new lowest key, the existing lowest key is supplied as the high key and the lower bound is supplied as the low key. Likewise, to generate the new highest key, the existing highest key is supplied as the low key and the upper bound is supplied as the high key.

Lexicographical sequences allow new document nodes to be added or existing document nodes to be removed from an index without the need to update other nodes. Given two keys *A* and *C*, the generator can always create a key *B* such that $A < B < C$

and that can be used in place of *A* or *B* to generate subsequent keys. Keys other the highest or lowest are generated using a divide by 2 algorithm as shown in Algorithm 1. Sample results for this algorithm are given in Table 1.

Algorithm 1: Lexicographical Sequence Generator

```
Inputs:
   S1, S2: strings S1 < S2
   UpperBound, LowerBound: lowest/highest byte values
Output: S3, L3: string S3 of length L3. S1 < S2 < S3

set S3 = CommonPrefix( S1, S2 )
if( Length( S1 ) ≤ Length( S1 ) and Length( S3 ) ≤ Length( S2 ))
   set MidByte ← Average ( LastByte ( S1 ), LastByte ( S2 ) )
   if ( MidByte != LastByte ( S1 ) )
      set LastByte ( S3 ) ← MidByte
   else
      set S3 ← S1
      Append( S3, Average(  UpperBound, LowerBound ) )
else
   set S3 ← S2
   set LastChar( S3 ) ← LowerBound
   Append( S3, Average(  UpperBound, LowerBound ) )
```

Low Input	*High Input*	*Output*
A	Z	M
M	Z	S
S	Z	X
X	Z	Y
Y	Z	YM
Y	YM	YH

Table 1: Cascading Lexicographical Key Generation

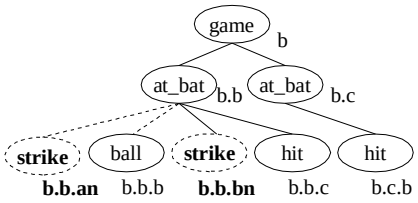
The worst case length of the keys increases as a function of the upper bound and lower bound. In a database however, these limits are set to the minimum and maximum unsigned byte values (0 and 255) to maximizes the fanout and produce shorter keys. The worst case growth of the key size occurs when consecutive values are generated. It is given as:

where: $K = N / \log_2(U - L)$

N - # of insertions
K - Length of key generated
U - Upper bound value of each character
L - Lower bound value of each character

Figure 13 demonstrates that the node insertions that were not possible in Figure

12 are possible using generated lexicographical sequences.



Inlined Order Key	*Inlined Path*
b	game
b.b	game/at_bat
b.b.an	game/at_bat/strike
b.b.b	game/at_bat/ball
b.b.bn	game/at_bat/strike
b.c	game/at_bat
b.c.b	game/at_bat/hit

Figure 13: Document Update Problem with New Nodes Added

4.4 Compression

MASS provides an effective compression mechanism that exploits the redundancy found in XML documents. Most XML data contains a limited number of node types that is determined either by constraints in the DTD or by the number of real world entities modeled by the data. As an example, the DTD in Figure 1 limits the document content to nine different node types (including text nodes). The limited number of node types alone guarantees some degree of redundancy in XML data. If the number of node types remains fixed and the document size increases, then the amount of redundancy will increase.

To further the case for compression, the inlined paths and order keys presented in Sections 4.1 and 4.2 guarantee that much more redundant data will be present in MASS than in the original document. This guarantee of redundancy ensures that some degree of compression is possible. This is the premise for our compression scheme. The high degree of redundancy for inlined data is illustrated in Figure 14. Note that all strings in this example occur in more than one location.

Inlined Order Key	*Inlined Path*
b	game
b.b	game/at_bat
b.b.an	game/at_bat/strike
b.b.b	game/at_bat/ball
b.b.bn	game/at_bat/strike
b.c	game/at_bat
b.c.b	game/at_bat/hit

Figure 14: Redundant Inlined Data

Compression in MASS is performed on a per-node basis. Nodes are compressed as they are inserted into MASS indexes. Individual nodes can be decompressed whenever they are accessed by a query without decompressing other nodes not needed by the query.

Since compression must be efficient in terms of both space and time, MASS limits compression opportunities to nodes stored in physically adjacent index entries on the same data page. Limiting the search for redundant data to adjacent nodes minimizes the complexity of the compression algorithm and keeps the compression mechanism independent of the physical node clustering. Limiting compression to entries on the same page eliminates the need for any additional I/O to compress or decompress nodes.

The compression scheme in MASS facilitates compression by linking entries that share redundant path or order key components, effectively forming a graph of compressed data. Before an entry is inserted, the entries it will be adjacent to are examined to determine if they contain any redundant path or order key components. The adjacent entry with the most redundant path components is selected as the 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 that entry has less redundancy than the candidate entry. If another entry is found to have more redundancy, then it becomes the compression candidate. This process guarantees that the depth of the compression graph is bounded and is discussed below in the context of decompression.

The example in Figure 15 demonstrates selection of a compression candidate. The *match size* is the sum of the lengths of strings that are redundant between a compression candidate and a new node.

	Inlined Order Key	Inlined Path	Match Size (bytes)
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 15: Compression Candidate Selection

If a node is found that is a good candidate for compression of the new node, then a bit vector is created to determine which components of the new node will need to be stored and which components will be shared. This bit vector, called the *compression mask*, is added to the index entry for the new node along with any non-shared components when the new node is finally inserted. The physical offset of the compression candidate on the data page must also be stored so that shared components can be located for decompression.

The compression scheme used by MASS indexes requires little overhead. The mask and compression offset are stored in a very compact format that typically adds between two and four bytes of overhead storage per index entry. This compression strategy greatly reduces the overhead of storing inlined node paths and order keys. The example in Figure 16 demonstrates compression reducing storage size by more than a factor of five.

Logical Index Entries

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

Physical representation of entries

MASK: 0=>Both Foreign, 1=>Path Foreign, 2=>Order Foreign, 3=>Both Local

Entry#	Foreign #	MASK	Local Strings	Size (bytes)
1		3333	d root d game d inning e at_bat	36
2	1	1	f	7

Figure 16: Physical Compression Representation

To decompress a node, a recursive lookup is used to retrieve all path and order

key components. The compression mask for each entry is used to determine which components are stored explicitly in the current node entry. If any path or order key components are instead part of another entry, then that other entry is read to determine which components are stored there. This process repeats itself until all path and order key components have been retrieved. Since each entry read will contain at least one path or order key component, 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 as long as other entries are present that produce better compression.

The compression scheme in MASS minimizes the performance penalty and fragmentation typically associated with compressed data by implementing compression at a granular level. Since individual entries can be decompressed, there is no up-front cost to access index entries. Entries that are not accessed are never decompressed. External fragmentation occurs when space cannot be used because it too small for the required allocation. Compression can cause external fragmentation in database structures since the variable size of compressed units is unlikely to align with the fixed sized database blocks. Since the size of each compressed entry in MASS is small with respect to the page size, the amount of external fragmented space will remain limited.

4.5 Node Clustering

Node clustering optimizes each MASS index for specific XPath axes. The concepts described up to this point were applicable to all MASS indexes. Node clustering defines the unique ordering for nodes in each MASS index. Each clustering is tuned to allow efficient evaluation of a subset of the XPath axes.

Node clustering provides nested ordering similar to the that of clustered

multicolumn indexes found in relational databases [14]. Each node clustering defines a sequence of keys that provides nested ordering for document nodes. Each prefix of the key sequence of length n can be used to perform a range search that matches the first n keys of the sequence. MASS guarantees that key sequences uniquely describe nodes and hence nodes can be located by exact match for all clusterings.

The first MASS clustering, CL1, is demonstrated in Figure 17. This clustering orders nodes in document order. Since the inlined order key can be used to determine document ordering, the key sequence for CL1 is simply <inlined order key>. Note the additional nesting that is inherited from the inlined order key.

Inlined Order Key	Inlined Path
d	/root
d.d	/root/game
d.d.d	/root/game/inning
d.d.d.d	/root/game/inning/at_bat
d.d.d.d.d	/root/game/inning/at_bat/ball
d.d.d.d.e	/root/game/inning/at_bat/strike
d.d.d.d.f	/root/game/inning/at_bat/out
d.d.d.e	/root/game/inning/at_bat
d.d.d.e.d	/root/game/inning/at_bat/strike
d.d.d.e.e	/root/game/inning/at_bat/hit
d.d.d.f	/root/game/inning/at_bat
d.d.d.f.d	/root/game/inning/at_bat/ball
d.d.d.f.e	/root/game/inning/at_bat/hit

Figure 17: CL1 Clustering Example

To demonstrate the efficiency of the CL1 clustering, we consider the location step in Figure 18, which selects all descendants of the context node. The descendant axis is retrieved by searching for the first and last descendants, then scanning the range between these points, which can only contain descendant nodes.

Context Node:	d.d.d.d /root/game/inning/at_bat
Location Step:	descendant:*

Figure 18: Descendant Axis Location Step

The first descendant will always be the node immediately following the context node in document order. The first descendant is located by searching for the first node greater than the context node. Next, the inlined order key of the first node is compared

with the inlined order key of the context node to determine if it is indeed a descendant. If the first node is not a descendant, then there are no descendant. Otherwise, we proceed to locating the last descendant node.

The last descendant node in the CL1 clustering is the last node in document order having an order key prefix equal to the order key of the context node. This node can be located by appending the lexicographical sequence reserved upper bound key ('z') to the inlined order key of the context node. The last descendant can be located by searching for the first node that proceeds the generated key (d.d.d.d.z) in document order.

The CL1 clustering also facilitates efficient query of the descendant-or-self, following, and preceding axes when the node test is “*” (matches all node types). Attribute nodes are excluded from the CL1 clustering since they are excludes from the descendant, descendant-or-self, following, and preceding axes. The CL1 clustering is the most efficient clustering for document extraction since elements can be retrieved in document order. However, attribute nodes must be merged in since they are node present in this clustering.

The CL2, CL3, and CL4 clusterings provide for efficient evaluation of the remaining combinations of of the XPath axes and node tests. The steps to query there clusterings are nearly identical to the example provided for the descendan axis. Sample data is provided for each of these clusterings to demonstrate the node ordering produced.

The CL2 clustering orders nodes with their siblings in document order. The key sequence for CL2 is <parent order key, relative order key>. This clustering provides efficient evaluation of the child, preceding-sibling, following-sibling, and attributes axes when the node-test is “*”.

The CL2 cluster stores stores all nodes, including attributes. Since attributes do not have relative order, attributes in the CL2 clustering are effectively stored in document order. Since they are in document order, these attributes can be efficiently merged with elements from the CL1 clustering for document extraction.

Inlined Order Key	***Path***
d	/game
d.d	/game/inning
d.d.a	/game/inning/id
d.d.d	/game/inning/at_bat
d.d.e	/game/inning/at_bat
d.d.f	/game/inning/at_bat
d.d.d.d	/game/inning/at_bat/ball
d.d.d.e	/game/inning/at_bat/strike
d.d.d.f	/game/inning/at_bat/out
d.d.e.d	/game/inning/at_bat/strike
d.d.e.e	/game/inning/at_bat/hit
d.d.e.e.a	/game/inning/at_bat/hit/bases
d.d.f.d	/game/inning/at_bat/ball
d.d.f.e	/game/inning/at_bat/hit
d.d.f.e.a	/game/inning/at_bat/hit/bases

Figure 19: CL2 Clustering Example

The CL3 clustering orders nodes by their node type in document order. The key sequence for CL3 is <relative location path, order key>. This clustering is optimized for evaluation of the descendant, preceding, and following axes when the node type is supplied in the node test since all nodes of the same type are in adjacent index entries.

Relative Path (node type)	***Inlined Order 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 20: CL3 Clustering Example

The CL4 clustering order entries by node type in sibling order. The key sequence for CL4 is <relative location path, parent order key, relative order key> This clustering provides efficient evaluation of the child, preceding-sibling, following-sibling, and attributes axis when the node type is supplied in the node test since sibling nodes of the same type are in adjacent index entries.

Relative Path (node type)	*Inlined Order 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 21: CL4 Clustering Example

We now consider the influence of particular clusterings on the effectiveness of compression. Recall that only nodes in adjacent index entries are compressed. The outer keys of each clustering key sequence determine which nodes occur in adjacent index entries. Consider the cluster example in Figures 17, 19, 20, and 21. Schemes CL1 and CL2 cluster by document order and therefore always share common path and order key prefixes. Schemes CL3 and CL4 rely on there being several instances of a given node type to achive compress of location paths, but this is common in XML data. The secondary document orderings of CL3 and CL4 cause adjacent nodes of the same type to share common order key prefixes. Adjacent nodes of different types will always share at least the first component of the inlined order so compression is always possible to some degree.

4.6 Persistent Storage

The clusterings described in the previous section have excellent query properties that we would like to allow for large documents. Persistent storage is needed for these clustering that allows efficient retrieval of both individual nodes and ranges of nodes from each cluster. Furthermore, because predicates in XPath can specify a relative position, we also need efficient random access to nodes in each clustering.

The ranked B+ tree [22] meets all of these requirements and plays an important

role in both the scalability and functional abilities of MASS. The ranked B+ tree provides efficient point and range access like that of the conventional B+ Tree. In addition, it also provides efficient random access to any entry by relative position and can efficiently calculate the distance between any two entries, as explained below.

The ranked B+ tree facilitates random access by maintaining subtree item counts in interior pages of the B+ tree. Each interior page has a counter for the total number of items in all subtrees and a counter for each individual subtree. Figure 22 provides an example of a ranked B+ tree that is ordered using the CL1 Clustering.

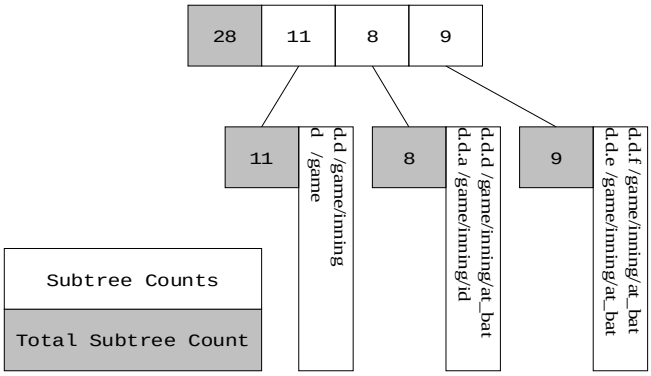


Figure 22: Ranked B+ Tree Organization

The ranked B+tree functions identically to a conventional B+tree for key lookups and range traversals. What is unique about the ranked B+ tree is its ability to seek backward or forward for an arbitrary number of entries or count the distance between any two entries in logarithmic time. This behavior is necessary to guarantee performance of document lookups by position. Consider the query in Figure 23, which selects the 30th at-bat of the game.

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

Figure 23: Query Accelerated by Ranked B+ Tree

Without random access, this query may have to scan the 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 in logarithmic time. For this example, the

appropriate clustering is CL4 since we are accessing the child axis and the node test is *at_bat*. The 30th node is accesses in the ranked B+ tree by searching for the first node by exact match then advancing 29 nodes in the index.

Another benefit of the ranked B+ tree tree is the ability to calculate the distance between two index entries in logarithmic time. Since there is a one-to -one correspondence with index entries and nodes in an axis, MASS can determine the number of nodes in each axis in logarithmic time. With these exact counts, a query processor can more effectively determine the best query plan. These statistics can be computed for all nodes in an axis, or for all nodes of a given type in an axis.

5 XPath Expression Evaluation

MASS facilitates XPath expression evaluation by providing optimized interfaces for each of the XPath axes. To query an axis, a context node, axis name and node test must be provided to MASS. The caller can then query the axis as desired. If the node test selects a node type, then only nodes of that type will be returned, otherwise all node types will be returned.

Each axis in MASS supports three query operations: *Fetch*, *Fetch* N^{th}, and *Count*. The semantics of these operation are as follows. *Fetch* returns the next node from the axis in document order. *Fetch* N^{th} returns the node at a specified ordinal position in the axis if the position is within the size of the axis. *Count* returns the total number of nodes in the axis.

Internally, the challenge to MASS is in mapping the requested axis and node test to one of the four clusterings. To set up for query of an axis, the following three steps are always performed:

1. Select the appropriate clustering using the axis and node test and open the corresponding index. The axis and node test combinations supported by each

clustering are described in Section 4.5.

2. Compose the search keys used to locate the first and last node in requested axis. Refer to the example below for detail.
3. Locate the first and last nodes in the axis. Store the locations of these nodes as the endpoints of the axis

As an example, we will now consider the location step in Figure 24, which selects all hit node children of the context node. The first step is to select a clustering, which is CL4 since the axis is child and the node test is not “*”. The search key for the first child is composed by appending the node type to the absolute path of the context node and appending the reserved lower bound key ('a') to the inlined order key of the context node. The search key for the last child is similar, except that the reserved upper bound key ('z') is appended to the absolute order key instead. The resulting search keys are shown in Figure 24. The first node is located 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 that same index for the first entry less than the last child key. The child relationship and node test must be checked for the first node to determine if any children of that type exist. This is done by determining if context node order key is the longest prefix of the order key and that the node test is true for the first node. For this example, d.d.e is the longest prefix of d.e.e.e and the child type is hit, so the node returned is in fact the first child of type hit. Note that since the first and last child are the same entry, there is exactly one node in the result.

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

Figure 24: Child Axis Location Step Example

The descendant axis is retrieved by searching for the first and last descendants, then scanning the range between these points, which can only contain descendant nodes. In the CL4 clustering

The first descendant will always be the node immediately following the context node in document order. The first descendant is located by searching for the first node greater than the context node. Next, the inlined order key of the first node is compared with the inlined order key of the context node to determine if it is indeed a descendant. If the first node is not a descendant, then there are no descendant. Otherwise, we proceed to locating the last descendant node.

Query operations are very efficient in MASS. Once the endpoints of the axis are located, no further searching is required. The Fetch operation can efficiently iterate the index up to the endpoint without the cost of comparing index keys. The Fetch Nth, and Count operations can re-use the endpoints of the axis for their evaluation.

6 Document Update

MASS is inherently designed to allow for efficient incremental document update. Incremental inserts in MASS are nearly identical to the initial construction of the indexes since XML data is loaded into MASS via an event-based SAX parser to begin with. The only dependencies between nodes are order keys and compression and these are both determined based on the current nodes in the indexes. In effect, every node loaded into MASS is an incremental insert.

To insert a new node, the inlined path and inlined order key must be determined. Nodes are always inserted after their parent node has been inserted. If the new node has existing siblings, then the relative order key must be calculated by applying the lexicographical sequence key generation algorithm from Section 4.3 to the relative order keys of the siblings. Otherwise, the relative order key can be any suitable key, such as 'n'. The inlined order key for the new node is formed by appending the relative order key for the new node to the inlined order key of the parent node. Likewise inlined path for the new node is formed by appending the node type for the new node to the inlined path of the parent node. The node can then be inserted into each of the indexes in MASS.

Node insertions are complicated by the per-node compression scheme. Since the size of each index entry is not known until the actual index entry is created, the entry must be constructed outside of the page and then copied to the page only if there is room. If the entry does not fit, then the page must be split and a new entry must be re-created since any entries that it referenced for compression purposes may have moved. Furthermore, when an index page splits, all nodes from the split page must be re-inserted into the new pages.

Node deletions are also complicated by the compression scheme. Since compressed index entries are linked together in an indeterminate order, there is no means to physically delete entries without completely re-creating data pages. Entries can only be logically deleted such that they do not appear in searches and do not effect performance. Space used for these logically deleted entries can be recovered during inserts by re-creating full pages that contain deleted items instead of splitting them. This allows MASS to amortize costs over multiple updates. This algorithm is shown in Figure 25.

7 Experimental Results

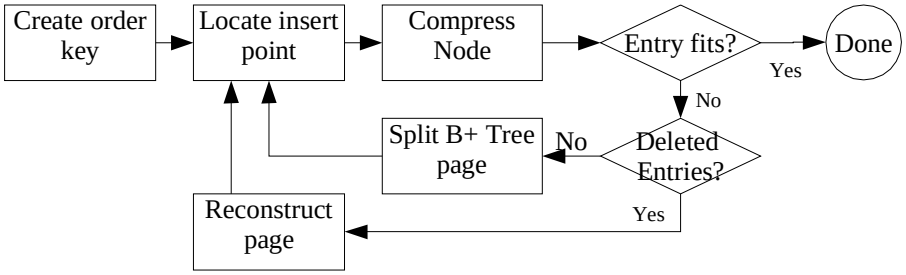


Figure 25: Modified B+ Tree Insert Algorithm

We have implemented MASS in C++ and extensively tuned the implementation for optimal query performance. With the exception of the Xerces SAX Parser [23] used to parse XML input files, the entire implementation was done from scratch.

7.1 MASS Architecture

Figure 26 represents the high-level architecture of MASS. At this abstraction, MASS is a set of indexes with interfaces for loading and querying documents. The interface for loading documents uses the Xerces C++ SAX parser [23]. The interface for querying documents is a set of C++ classes that represent the various XPath axes.

The role of the SAX parser is to scan the XML document and create document events as it encounters constructs such as the start of a new element. Each event must be handled before the parser can continue scanning. The SAX parser maintains minimal state information and thus requires little overhead, even for parsing large documents.

The SAX Handler inserts nodes into the indexes based on document events from the SAX parser. The SAX Handler registers callback functions with the parser to handle each document event. The current position in the document hierarchy is maintained by storing the inlined path and inlined order key for the last node inserted. The SAX Handler generates order keys for new nodes as they arrive and inserts these

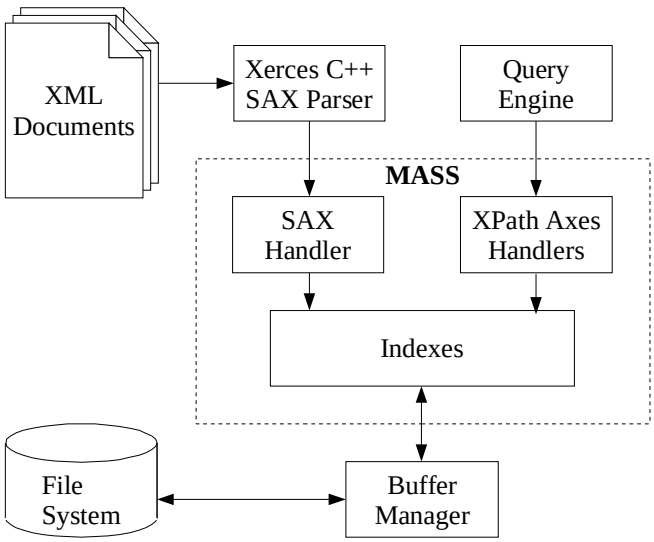


Figure 26: MASS Architecture

nodes into each of the indexes.

The indexes facilitate persistent storage of document nodes. There is one index for each of the four clusterings described in Section 4.5. When loading documents, these indexes transform the in-memory representation of nodes from the SAX Handler to the compressed representation on disk. When querying documents, the indexes decompress nodes into an in-memory representation suitable for queries.

The XPath Axis handlers map XPath axis requests to primitive index operations. This allows the query engine to request nodes from a specific axis without any knowledge of the internal representation of MASS. The axis handler maintains the current location in the index so that nodes can be fetched sequentially without iterative lookups.

7.2 Implementation

The child, ancestor, descendant, following, and following-sibling axes were fully implemented and tested. The remaining axes had trivial differences such as reverse document order (i.e. preceding vs. following) or disjunction with the context node (i.e. descendant vs. descendant-or-self) that were deemed unnecessary for evaluation.

The implementation of MASS is compact at approximately 5,000 lines of C++ source code. The key to such a compact implementation was the extensive use of the C++ Standard Template Library (STL) [25]. MASS implements STL iterators as the interface to each of its collection classes. The generic iterators can then be used with STL template algorithms. Figure 27 demonstrates the key comparison function for the CL3 clustering, which uses the template algorithm *lexicographical_compare_3way*. This same algorithm is used in all other functions that perform lexicographical comparisons.

```
// Function: static NodeHelper::CL3Compare
// Purpose:  Key comparison for the CL3 clustering
// Inputs:   Nodes to compare
// Return:   true if lhs < rhs

bool NodeHelper::CL3Compare(const NodeHelper& lhs, const NodeHelper& rhs)
{
// Compare the relative path (node type)
// rpbegin() returns an iterator to that last segment of the inlined path
  int rc = lexicographical_compare_3way( lhs.rpbegin(),
                                         lhs.rpbegin() + 1,
                                         rhs.rpbegin(),
                                         rhs.rpbegin() + 1);
// If the relative path is the same, compare inlined order keys (document ordering)
// obegin() returns an iterator to the first string in the inlined order key
// oend() returns an iterator beyond the last string in the inlined order key
  if(0 == rc)
    rc = lexicographical_compare_3way( lhs.obegin(),
                                       lhs.oend(),
                                       rhs.obegin(),
                                       rhs.oend());
  return rc < 0;
}
```

Figure 27: Source Code Using STL Iterators

7.3 Experimental Setup

Data from the XMark [24] XML generator was used for all performance testing. Data from the XMark generator models an online auction website. The data produced by this generator has over 40 element types and large structural variation. The data is approximately 70% text and 30% markup. Table 2 provides additional heuristics about the XMark data used.

Document Size(kb)	*Element*	*Attributes*	*Text Nodes*
113	4898	357	3169
557	23952	1974	15434
1134	48211	3919	31089
5636	235051	19249	151518
11396	472684	38265	304819

Table 2: Test Data Heuristics

Tests were run for each of the document sizes given in Table 2. 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. Tests were run on a Sun E4500 with 8 400MHz CPUs and 8GB Ram. MASS is single threaded and only utilizes one CPU, but the large amount of memory was required since out test configuration keeps all indexes in memory and simulates I/O operations.

7.4 Load Performance

Load time was measured for each document size. For each load, the wall time, number of I/Os and number of key comparisons was measured. We then repeated these experiments with compression turned off.

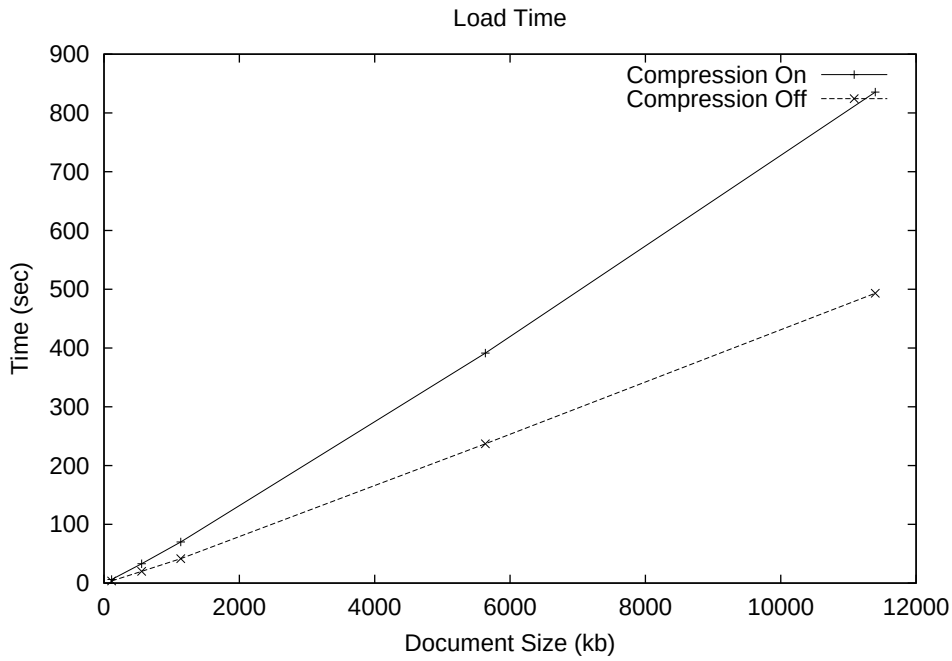


Figure 28: Load Performance

The results in Figure 28 demonstrate that the loading capabilities of MASS scale linearly with document size. These measurements include the time to load all four MASS indexes. Using this data in conjunction with data from Table 2, we can calculate that MASS can load 2,356 compressed nodes or 3,990 uncompressed nodes per second respectively. Load times are consistently 70% slower with compression turned on.

7.5 Index Size

The index size for each of the four node clusterings in MASS was recorded for each document size. This measurement was then repeated with compression turned off.

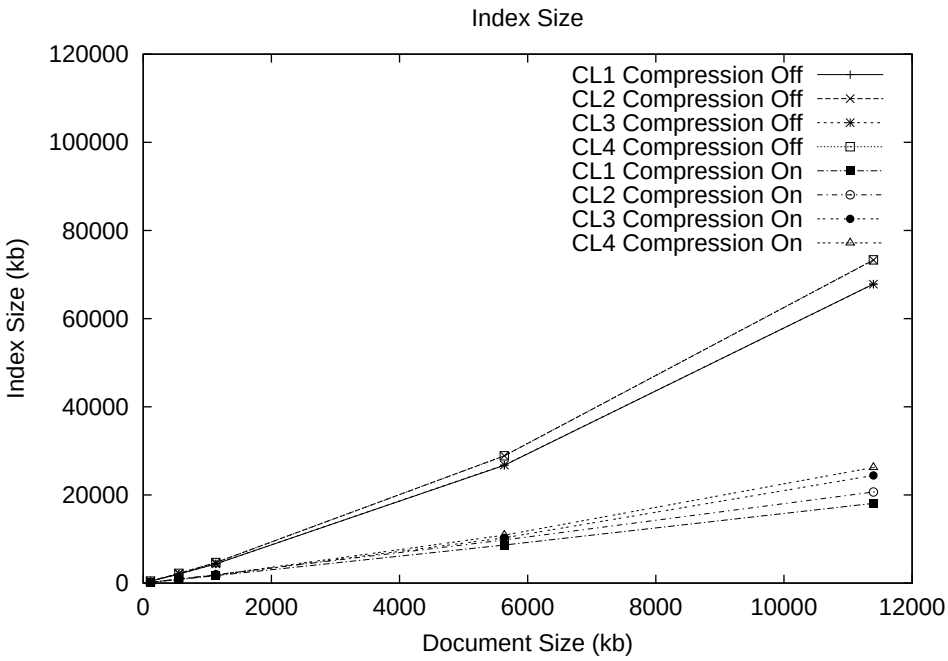


Figure 29: Index Size

The results in Figure 29 demonstrate that compression in MASS can reduce the size of indexes by 60-70%. With compression off, the CL1 and CL3 clusterings are slightly larger since these indexes do not store attributes. With compression on, the CL1 and CL2 indexes are slightly smaller. Compression is likely better for the CL1 and CL2 clusterings because their primary ordering is based on document ordering whereas CL3 and CL4 are primarily ordered by node type.

7.6 I/O Performance

To measure I/O performance of query operations, we have instrumented the buffer cache to count the number of logical I/Os. The B+Tree is MASS is designed to keep the current leaf page in memory so that sequential fetching only creates I/O when page boundaries are crossed. Therefore, the logical I.O actually represents required physical I/O.

I/O performance was measured for three different operations. The *Fetch* operation reads all nodes in the axis. The *Count* operation counts the number of nodes

in the axis. The *Fetch N*th operation fetches the node in the middle of the axis. I/O tests were run against each of the five implemented axes at each of the five document sizes. The number of I/Os performed for the five axes tested are shown in Figures 30-34.

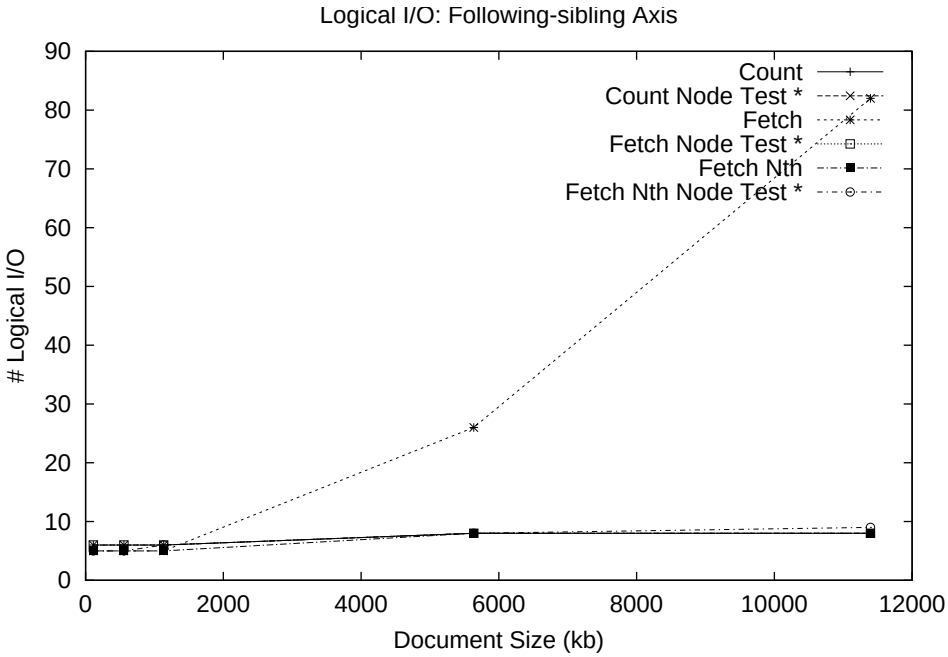


Figure 30: Following-sibling Axis I/O Performance

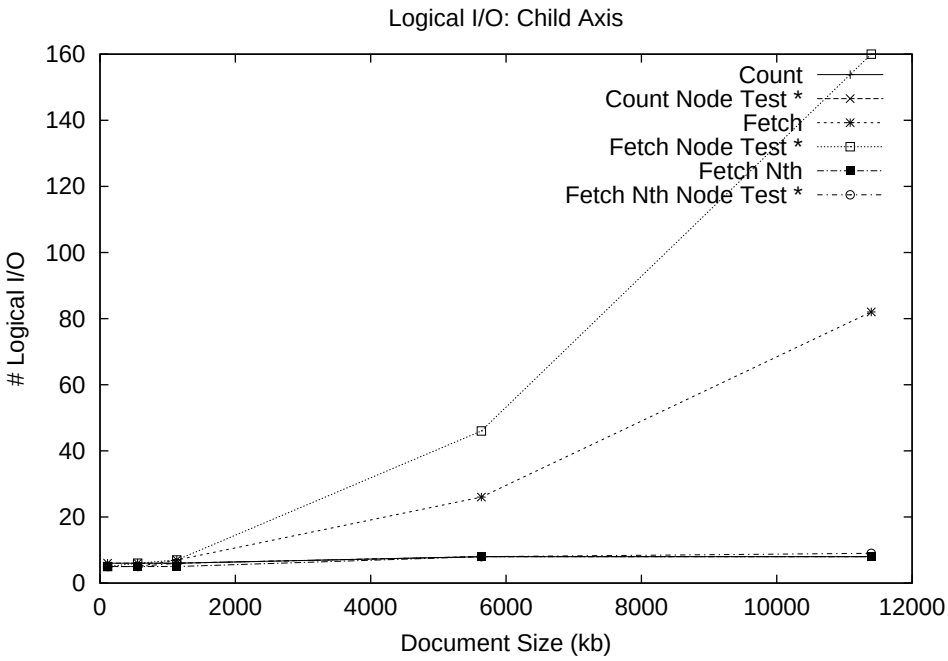


Figure 31: Child Axis I/O Performance

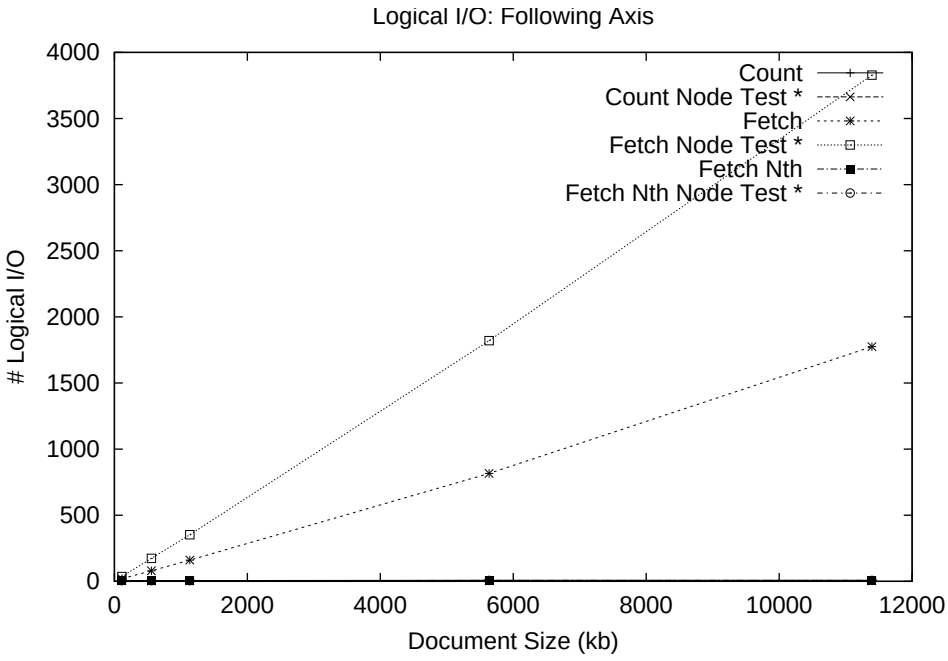


Figure 32: Following Axis I/O Performance

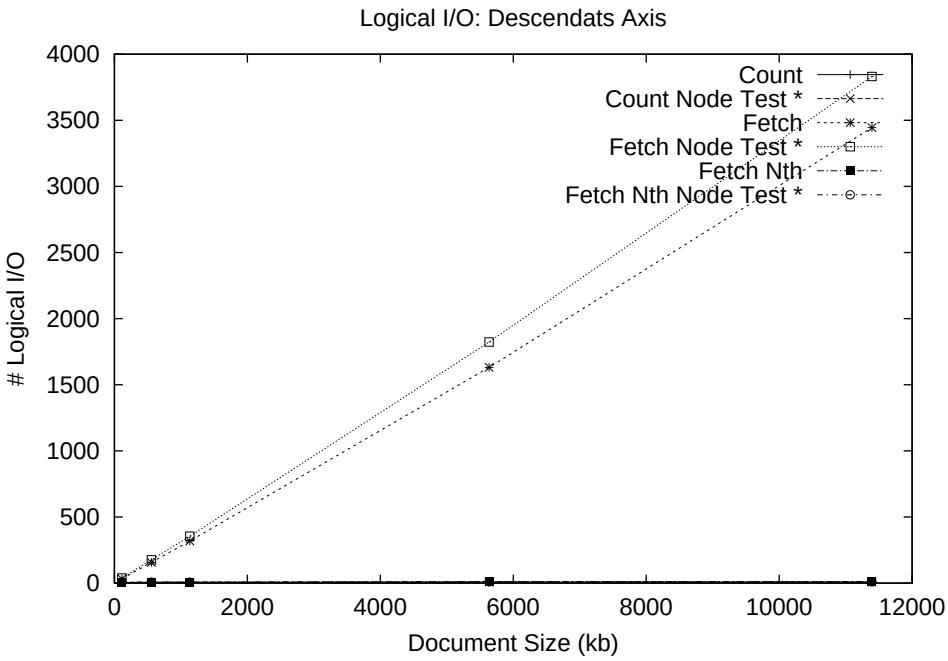


Figure 33: Descendant Axis I/O Performance

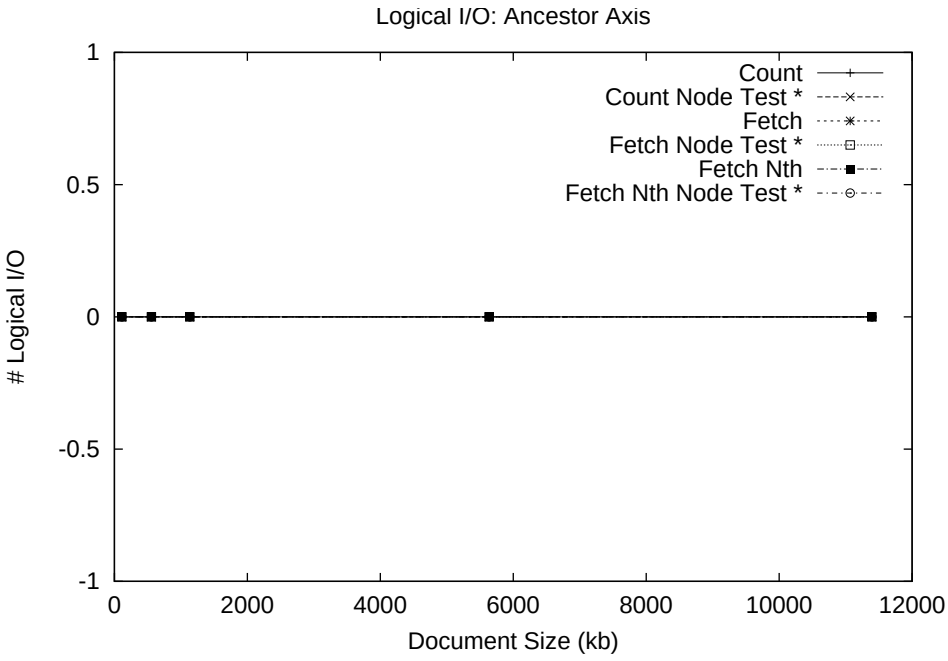


Figure 34: Ancestor Axis I/O Performance

All axes provided similar results with the exception of the ancestor axis, which reads node information from its inlined path and order key rather than performing

I./O. For the other axes, the number of I/Os for the *Fetch* operation increased with document size since all nodes in the axis were actually decompressed and read. The number of I/Os for the *Fetch* N^{th} and *Count* operations remained small because the ranked B+ tree was able to evaluate these operations efficiently.

7.7 CPU Performance

To measure CPU related performance of query operations, instrumentation was added to count the number of key comparisons performed by each query. The time to complete each operation was also measured and used to calculate the fetching throughput.

The number of key comparisons for the five axes tested are shown in Figures 35-39. Note that the number of key comparisons is bounded as document size increases for all of the axes. This is because MASS performs at most two B-Tree searches to locate the end points of the axis. The actual fetching never has to compare keys. Key comparisons are not performed for the ancestor axis since it is evaluated using information from the inlined path and inlined order key of the context node.

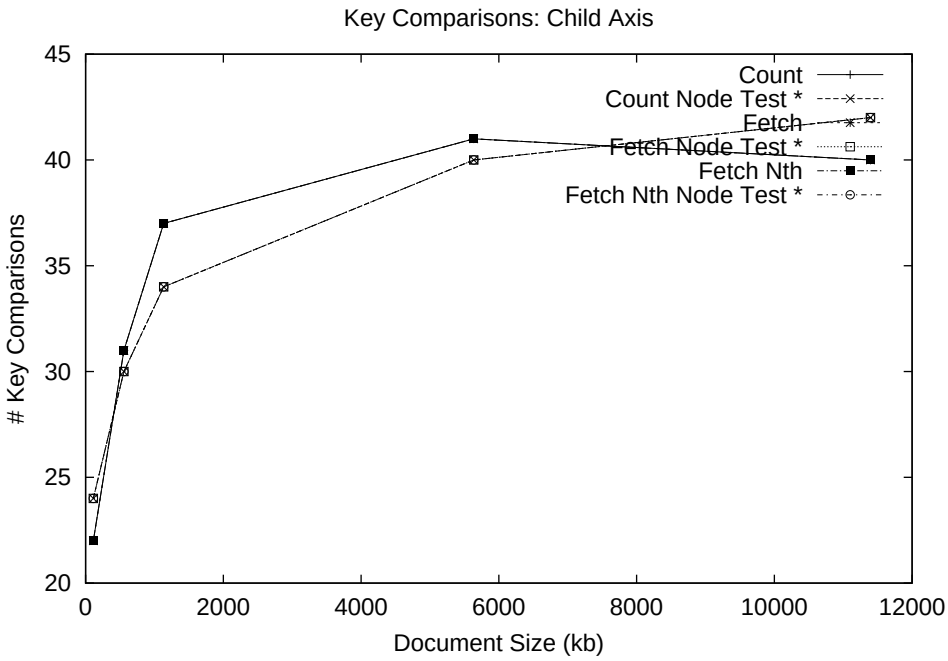


Figure 35: Child Axis Key Comparisons

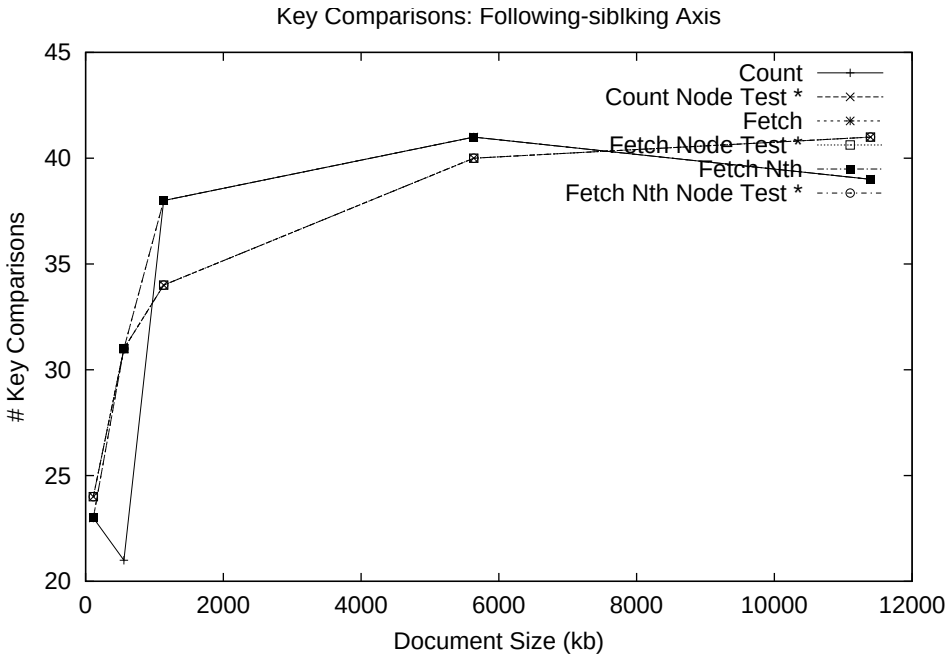


Figure 36: Following-sibling Axis Key Comparisons

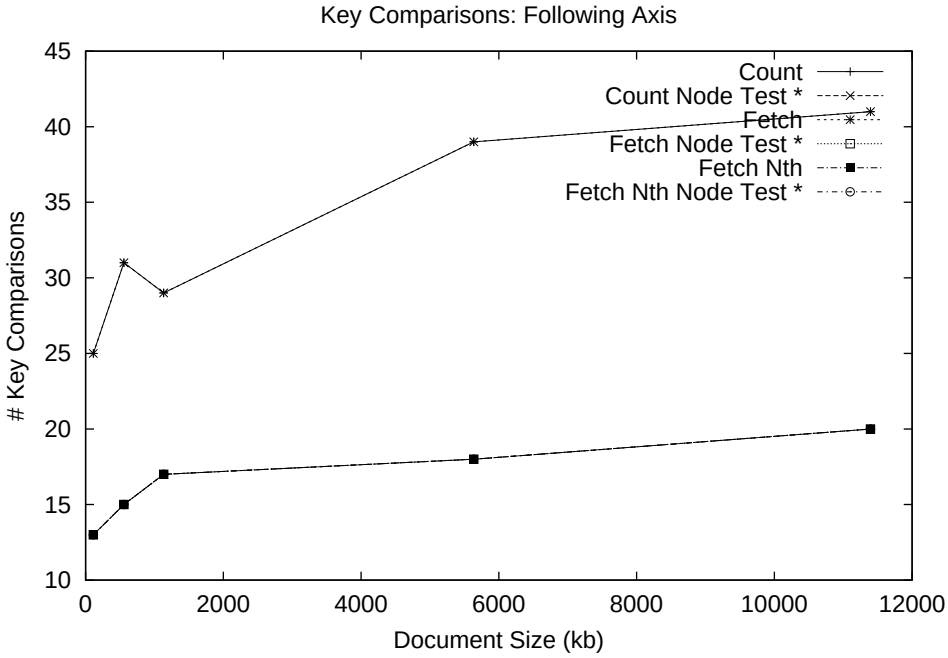


Figure 37: Following Axis Key Comparisons

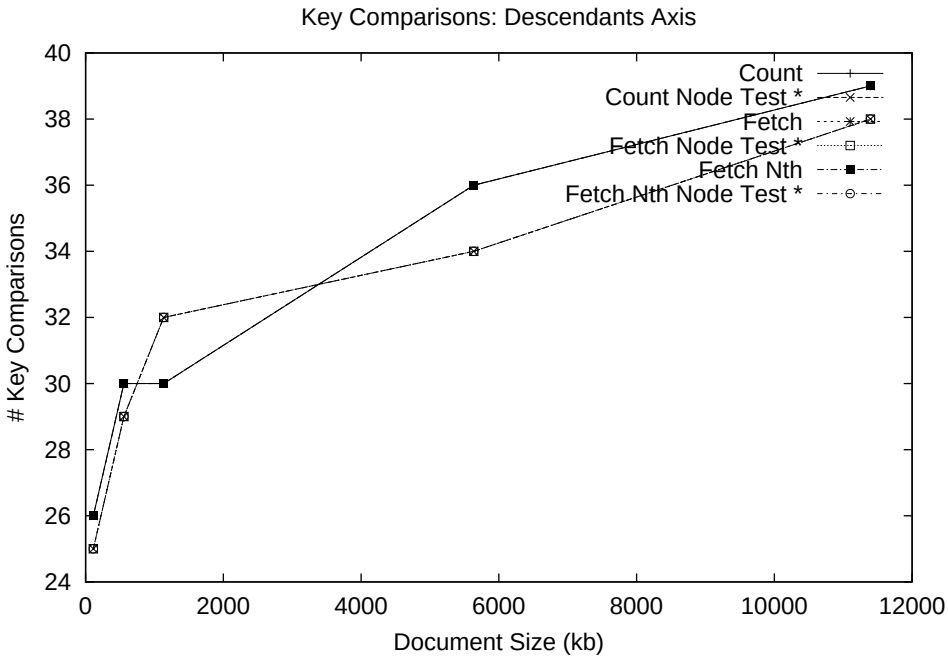


Figure 38: Descendant Axis Key Comparisons

Node throughput was calculated by dividing the number of nodes fetched in the

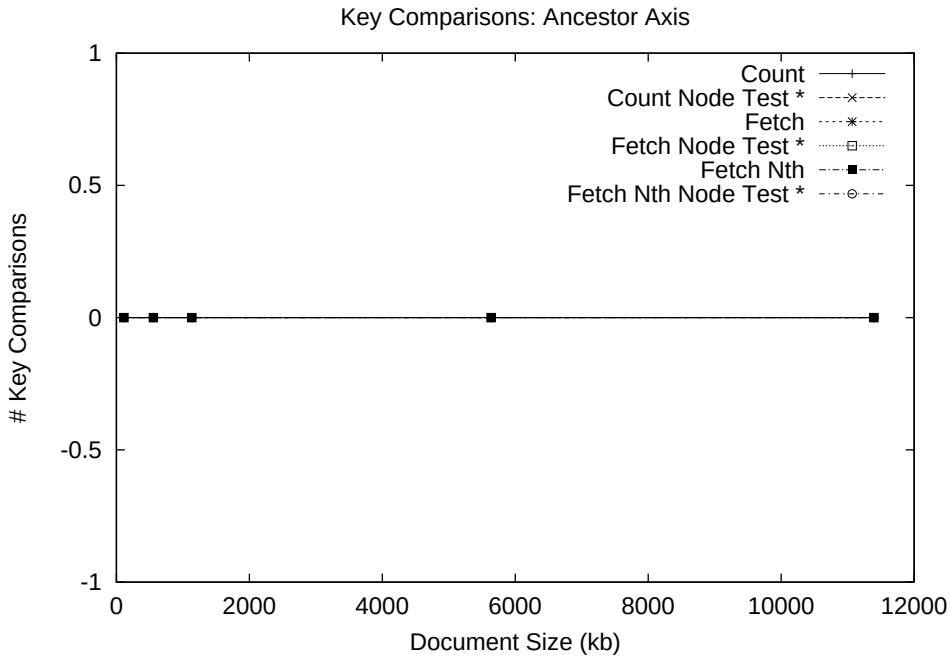


Figure 39: Ancestor Axis Key Comparisons

descendant axis by the wall time to fetch these nodes. The descendant axis was used for this measurement because it is always the largest axis in terms of node count (the descendants of the root node includes all non-attribute nodes in the document).

The chart in Figure 40 shows that compressed nodes can actually be retrieved faster than uncompressed nodes with large documents, even though we have not performed I/O in this experiment. One possible insight into this behavior is that the larger size of the uncompressed index may be adversely effecting the CPU cache.

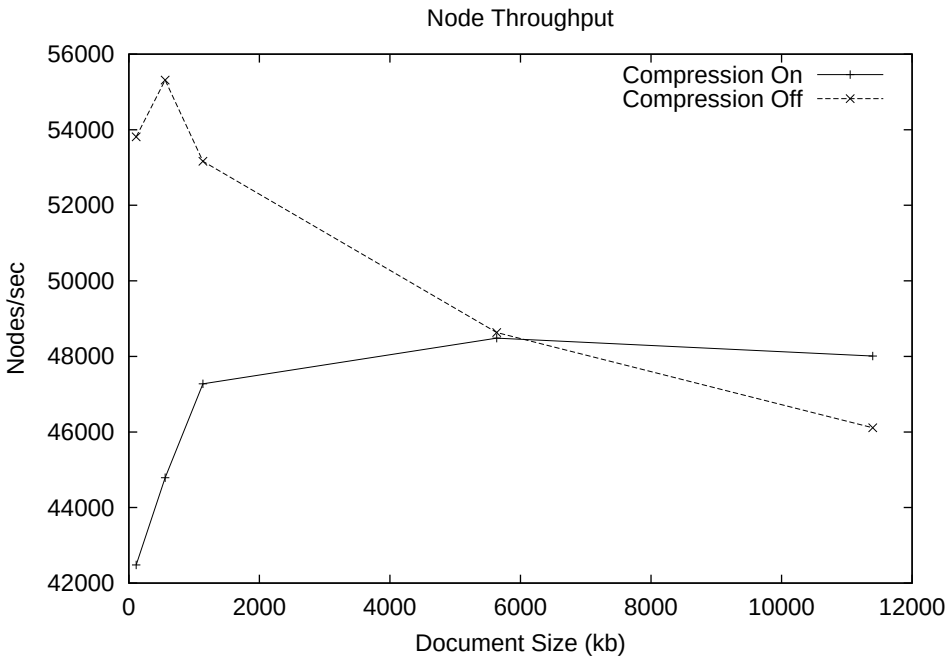


Figure 40: Node Throughput

8 Background

Results from previous research [2] suggest that there is much room for improvement in the design of XML storage and indexing. In its text form, an XML document cannot be used to efficiently evaluate any of the XPath axes. The two main approaches used thus far for XML storage are mapping document structure to relational database schemas and storing document structure in indexes designed specifically for semistructured data.

8.1 XML in Relational Databases

Efficient storage and query of XML data in relational databases has proven difficult to support. The relational data model is awkward for modeling complex nested XML constructs [4]. Problems such as differentiating attributes from sub-elements, facilitating document references, and assembling the original document are difficult with a relational mapping [7,10], even with additional domain information [5,6]. Mapping XML data to relational tables is difficult since the mapping may

inhibit query functionality or excessively fragment data, creating the need for complex joins to evaluate expressions [4]. Structural changes to relational mapped documents can result in creating, dropping, or altering tables, which is costly in any DBMS.

To query XML that has been mapped to a relational database, queries must be re-written to their SQL counterparts. Complex expressions that involve pattern matching or recursion may not even be possible to write in SQL. Since SQL does not support dynamic table names in join statements, an additional layer must be supported to assemble SQL statements. These queries are then executed ad-hoc. SQL translation may results in complex SQL queries that stress the capabilities of DBMS optimizers [5].

8.2 Native XML Indexing

Many structures have been proposed for indexing various types of path expressions. Earlier structures [2,3,5,13] were developed in the context of semistructured data. More recent proposals [19, 20, 21] discuss the problem in the context of XML. These structures create various decompositions of document graphs to accelerate expressions involving labeled paths. In [19], a compact index called the Index Fabric is proposed that uses a trie structure to store root to leaf paths. While this structure can efficiently supports prefix matching expressions, support for other expressions must be manually tuned by creating *refined paths* and document ordering is not maintained. A numbering scheme to encode document ordering is proposed for XISS [20]. This scheme pre-allocates number ranges to store the document structure and ordering. This scheme can evaluate ancestor-descendents relationships in constant time, but only as a filter once nodes have been selected. The system proposed in APEX [21] optimizes frequently accessed paths by storing them in a large hash table.

However, document ordering is not preserved and paths not found in the hash table must be evaluated recursively by joining individual extent, which is innefficient for long paths.

While the aforementioned proposals provide novel ways to evaluate path expressions involving labeled paths, these expressions are representative of only part of the XPath [14] language. These solutions do not provide adequate support for queries that select nodes based on document ordering. The examples in Figure 41 demonstrate path expressions that cannot be evaluated efficiently aforementioned proposals. Note that XQuery [15] requires that the result of a path expression is in document order, so the first expression may require a large sort to evaluate for indexes that do not maintain document order. Since the clusterings in MASS all return nodes in document order, no further sorting is needed.

```
/Game/*
/Game[1]/Inning/[9]
```

Figure 41: Queries Based in document Ordering

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 [19, 20, 21], none of them provide a cost model or meaningful statistics for a query processor. MASS is able to quickly provide the exact size of each axis so the that query processor can aggressively optimize complex expression queries.

The path inlining concepts in MASS are derived from dataguides [18] and from earlier proposals for relational database mapping [4,10]. The strong data dataguide [18] is similar in that encodes all root-to-leaf paths, but is ordered entirely based on

node type and is therefore only useful for selecting nodes by labeled path. A strong dataguide could be created in MASS simply by defining a clustering that is ordered by the inlined path. Efforts to employ inlined paths in relational databases have been hampered by the constraints of the relational storage. In a relational database, inlining is accomplished using either multicolumn indexes or by concatenating data as strings and storing the strings in a single varchar column. If multicolumn indexes are used, there is a large increase in storage size from the overhead to fill non-existent edges with nulls [4] and any word alignment enforced by the DBMS. Furthermore, structural changes may have to alter the relational schema or use overflow storage [10]. If varchar keys are used, there may be a large cost perform type conversion and decompose keys.

Compression of XML data is covered extensively by the XMILL [17] Research. While MASS follows on the concept of compressing redundant structural data, it also applies compression to the keys that are generated to infer document ordering. XMILL compresses entire documents and can therefore provide near-ideal compression. Conversely, MASS implements compression of individual document nodes and must do so as efficiently as possible.

Clustering has previously been considered for XML data, but not with regard to the XPath language. A prior study of XML query optimization [5] noted that clustering can be used to further optimize queries [5,7]. MASS simplifies query optimization in this regard by creating multiple clusterings that are directly correlated to the query language, eliminating the need for index selection.

9 Conclusion

In this paper, we propose MASS, a Multiple Axis Storage Structure. Mass facilitates efficient evaluation of XPath expression by supporting each XPath axis

using a limited number of clustered indexes. Mass incorporates five concepts into each of these indexes: ordering, compression, clustering, and persistent storage. MASS also provides lossless storage for XML documents by preserving all node relationships and data content, eliminating the need for XLM flat files.

MASS supports all XPath Axes, both for queries involving document order and for queries that are only selective by node type. Data is clusted in document order, allowing efficient query by document position and eliminating the need for sorting to order results. Query operations in MASS scale with document size and require little system resources due the clustering organization and Ranked B+ Tree used to store document nodes.

MASS provides for efficient document update while preserving document order using generated lexicographical sequences. These keys allow nodes to be incrementally added to or deleted from a document while never requiring update of other existing nodes. The same keys can be used to determine all node relationships and uniquely identify nodes for query processing.

Path expression cannot all be evaluated with a single index organization. MASS has taken the approach of creating efficient building blocks for indexes, then using this building blocks to construct efficient indexes that can together provide efficient path indexing for all of the axes in XPath. To our knowledge, this is the first proposal for structural indexing at this scope.

References

1. Dan Suciu. Semistructured Data and XML. AT&T Labs, Technical Report 1997.
2. Jason McHugh, Jennifer Widom, Serge Abiteboul, Quigshan Luo, and Anand Rajaraman. Indexing Semistructured Data. Stanford University Technical Report, 1998.
3. Tova Milo and Dan Suciu. Index Structures for Path Expressions, 1997.
4. Jayavel Shanmugasundaram, Kristen Tufte, Gang He, Chun Zhang, David DeWitt,
   and Jeffrey Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In Proceedings of the 25th annual VLDB Conference, 1999.
5. Jason HcHugh and Jennifer Widom. Query Optimization For XML. In Proceedings of
   the 25th annual VLDB Conference, 1999.
6. Jyh-Herng Chow, Josephine Cheng, Daniel Chang, Jane Xu. Index Design for Structured Documents Based on Abstraction. IBM Santa Teresa Labs Technical Report, 1998.
7. Daniela Florescu and Donnald Kossmann. Storing and Querying XML Data using an
   RDBMS. IEEE Technical Committee on Data Engineering, 1999.
8. Dongwook Shin, Hyuncheol Jang, and Honglan Jin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. ACM Digital Library, 1998.
9. Weidong Chen, Jyh-Herng Chow, You-Chin Fuh, Jean Grandbois, Michelle Jou, Nelson Mattos, Brian Tran, and Yun Wang. High Level Indexing for User Defined
   Types. In Proceedings of the 25th annual VLDB Conference, 1999.
10. Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. University of Pennsylvania.
11. Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, David Maier, and Dan Suciu. Querying XML Data. IEEE Technical Committee on Data Engineering, 1999.
12. Don Chamberlin, Jonathan Robie, and Daniela Florescu. QUILT: An XML Query

Language for Heterogeneous Data Sources. 2000.

13. Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A Database Management System for Semistructured Data. Stanford University Technical Report, 1998.
14. Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1992.
15. Jim Clark and Steve DeRose. XML Path Language (XPath), Version 1.0W3C Recommendation November 1999. http://www.w3.org/TR/xpath.html
16. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. Working Draft, http://www.w3.org/TR/2001/WD-xquery-20011220, 20 December 2001.
17. Hartmut Liefke, Dan Suciu: XMILL: An Efficient Compressor for XML Data. SIGMOD Conference 2000: 153-164.
18. R. Goldman and J. Widom. DataGuides: Enable query formulation and optimization in semistructured databases. In Proceedings of 23rd International Conference on Very Large Data Bases, pages 436 445, August 1997.
19. Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, Moshe 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", Proc. of 27th Intl. Conf. on Very Large Data Bases, 2001.
21. Chin-Wan, Chung, Jun-Ki Min, Kyuseok Shimy, “APEX: An Adaptive Path Index for XML Data”, ACM SIGMOD 2002.
22. D. Knuth: “The Art of Computer Programming: Volume 3, Sorting And Searching”, Addison-Wesley, 1973
23. Xerces C++ parser. The Apache XML Project. http://xml.apache.org/xerces-c/index.html
24. Xmark-The XML-Benchmark Project. http://monetdb.cwi.nl/xml/index.html
25. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/