Querying and Maintaining Succinct XML Data

Franky Lam William M. Shui Raymond K. Wong
National ICT Australia
and School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{flam, wshui, wong}@cse.unsw.edu.au

ABSTRACT

As XML database sizes grow, the amount of space used for storing the data and auxiliary supporting data structures becomes a major factor in query and update performance. This paper presents a new secondary storage scheme for XML data that supports all navigational operations in near constant time. In addition to supporting efficient queries, the space requirement of the proposed scheme is within a constant factor of the information theoretic minimum, while insertions and deletions can be performed in near constant time as well. As a result, the proposed structure features a small memory footprint that increases cache locality, whilst still supporting standard APIs, such as DOM, and necessary database operations, such as queries and updates, efficiently. Both formal analysis and experimental evidence demonstrate that the proposed structure is space and time efficient.

Categories and Subject Descriptors
H.3.2 [Information Systems]: Information Storage; H.2.4.n [Textual Databases]: XML Databases

General Terms
XML

Keywords
XML, Compact Storage, Storage Optimization, Query Processing

1. INTRODUCTION

The popularity of XML as a data representation language has produced a wealth of research on efficiently storing and querying tree structured data. As the amount of XML data available increases, it is becoming vital to be able to not only query and maintain this information quickly, but also store it compactly. We thus turn to the problem of finding a succinct representation for XML: a space-efficient representation of the data structure which also maintains low access and update costs for all of the desired primitive operations for data processing. The flexibility of XML makes finding a scheme which satisfies all these requirements at the same time extremely challenging. There are numerous reasons to maintain such a compact XML representation on secondary storage, for example:

- Reducing space requirements improves cache locality: Even in the current environment of enormous secondary storage capacities, reducing the space requirements for native XML databases is an important goal. A typical approach to representing XML in such databases is to keep at least four pointers per node, to the parent, first child, and immediate siblings. This approach can also be found in many XML tools such as libxml8. In the standard computational model, where a pointer takes $O(\log n)$ bits, using the above approach to represent the topology of $n$ nodes requires $\Theta(n \log n)$ space. For large XML documents, this representation becomes infeasible for many applications, particularly as the hidden constant in the space bound is relatively high. Furthermore, using more space also reduces the cache locality and has an adverse impact upon query performance.

- Indirection is expensive: There has been a large amount of work on the succinct representations of trees [14, 17, 18, 27–31], many of which come within a factor of the optimal lower bound on space. However, to achieve these lower bounds generally requires a significant amount of address indirection. Such schemes are not suitable for secondary storage due to the expensive cost of a random disk seek, which will generally be required upon each indirection. In general, there is a trade-off between space usage and indirection, which we will optimize for secondary storage devices in this paper.

When looking for a succinct storage scheme for XML, there are many important issues that need to be addressed:

- It must support fast navigational operations: Many XML applications, such as collaborative document editing systems, depend upon efficient tree traversal, using a standard interface such as DOM. Halverson et al [16] demonstrated that a combination of navigational and structural join operators is most effective for evaluating queries. Hence, it is imperative that the storage scheme supports fast traversal of the XML tree, in all possible directions, preferably in constant time or near constant time. Previous work, such as that of Zhang et al [36], has addressed the issue of succinctly representing XML, but at the cost of linear time navigational operations, which is not acceptable for many practical applications. Our structure efficiently supports tree navigation primitives in $O((\log n)/\log \log n)$ time, and also includes support for efficient structural joins.

- It must support efficient insertions and deletions: Several papers address the space issue by storing XML in compressed form [7,23,26,33]. They also support path expression queries

\footnote{See http://www.xmlsoft.org}

\footnote{In this paper, $\log n$ is the base 2 logarithm of $n$}
or fast navigational access but do not allow efficient update operations such as node insertion. This can be a critical concern in many real database applications. In this paper, we provide a scheme which allows near constant time for update operations in practice, with a theoretical worst case time of $O(\lg^2 n)$.

- **It must support efficient join operations**: Current query optimization techniques for XML such as work of Halverson et al [16], make heavy use of the structural join [2], which relies on a constant time operator to determine the ancestor-descendant relationship between two nodes. Thus, any general XML storage scheme should also support such an operator in near constant time. Our scheme supports ancestor-descendant queries in $O(\lg n / \lg \lg n)$ time.

- **It must be practical**: Many succinct tree representation schemes are elegant theoretical structures that unfortunately do not translate well into practice. Thus, while theoretical guarantees are important for any proposed structure, practical considerations should not be forgotten. In this paper, we focus on developing a practical storage scheme, using values that fit to the natural machine word size, block size and byte alignment, to allow our scheme to be used in real-world database systems.

- **It must be simple**: Ideally, as with B-trees, the basis of the data structure should be simple and clean enough to be used as material for an undergraduate course. Our scheme, while both extremely compact and efficient, is also amenable to simple implementation.

- **It should separate the topology, schema and text of the document**: All XML query languages select and filter results based on some combination of the topology, schema and text data of the document. To allow efficient scans over these parts of the document, it is natural to find a representation that partitions them into separate physical locations.

- **It should permit extra indexes**: Many applications may require addition specialized indexes to be built upon their data. Therefore, a general purpose database system is required to provide a storage representation, such that it is flexible enough to accommodate such need. More specifically, the storage scheme used by the database system must provide a simple, efficient and stable way of referencing its stored data items.

In this paper, we propose an Integrated Succinct XML (ISX) system, where we store XML in a more succinct structure, which addresses all of the above issues. Theoretically, the ISX system uses an amount of space near the information theoretic minimum on random trees. For a constant $\epsilon$, where $1 \leq \epsilon < 2$, and a document with $n$ nodes, we need $2\epsilon n + O(n)$ bits to represent the topology of the XML document. Node insertions can be handled in constant time over average but worst case $O(\lg^2 n)$ time, and all node navigation operations take worst case $O(\frac{\lg n}{\lg \lg n})$ time but constant time on average.

The rest of this paper is organized as follows: Section 2 summarizes relevant work in the field. Section 3 presents the basics of the ISX storage and topology layer for succinct representation of XML data. The fast node navigation operators, the querying interfaces and the update mechanism are then described in detail in Section 4. The experimental results are then presented in Section 5, and finally Section 6 concludes the paper.

## 2. RELATED WORK

There are two research directions related to the XML representation scheme: i) compression schemes that purely aim for space efficiency and disregard query and update performance; and ii) database schemes that aim for efficient query processing but disregard space consumption and sometimes update efficiency. Recently, some mixed approaches spanning these two directions have emerged. For example, there are XML compression schemes that support some regular path queries, e.g., XGrind [33].

### 2.1 Succinct Data Structure

**Succinct data structure** is a research area that aims to maximize update and search operations whilst setting the constraint of size to be close to the theoretical optimal. Since XML data can be modeled as ordered trees, storing XML succinctly is closely related to succinct tree representations. The earliest space efficient data representations for static unlabeled trees were proposed by Jacobson [17, 18], who showed that the information content of a unlabeled tree of $n$ nodes is $\lg k^n$ or $O(n)$ bits. Hence, any representation of such trees must use at least a linear amount of space. The author then gave a representation which used $2n$ bits, plus an additional $o(n)$ bits, which supported ordered tree operations such as finding the first child, next sibling, and parent of a node in $O(\lg n)$ time. The author also introduced two fundamental operations, rank and select, in terms of which all other operations could be implemented.

Early works on succinct representations all assume a static model, and hence are not easily generalized to support updates. Clark and Munro [11] gave a binary tree representation using $3n$ bits, which was used as a Patricia trie to index large, static, text files whilst minimizing the number of disk accesses. However, their scheme does not support navigation to a node’s parent, and hence it is not clear how to extend the scheme to support updates. Munro and Raman [27] then developed a scheme which essentially solved the succinct representation problem for static unlabeled binary trees, as it allowed $O(1)$ time navigational operations with asymptotically optimal space. However, for rooted ordered trees, finding the $n$-th child of a node took $O(n)$ time. On the other hand, the scheme of Benoit et al [6] can support this operation (and also all other navigational operations) in constant time. We emphasize that all these results hold only when no updates are allowed, which is clearly undesirable in a database system (where being able to update the data efficiently is critical).

The first work giving a succinct representation for dynamic labeled trees was that of Munro et al [28], which supported binary trees with labels of constant size. However, it did not support trees of higher degree. Ramam et al [29] later extended the $2n$ bit representation of Jacobson [18] to a special case of the updatable partial sum problem called the dynamic bit vector problem. It supported rank and select with updates in $O(\lg n / \lg \lg n)$ time using an extra $o(n)$ bits space. Alternatively, the structure supported $O(1)$ time for rank and select with updates in $O(n^\epsilon)$ time, allowing a trade-off between time and space. Raman et al [31] also considered the space and time cost overhead used by the memory manager. They further improved the lower bound for labeled dynamic binary trees, supporting navigational operations in $O(1)$ time with updates in $O((\lg \lg n)^{1+\epsilon})$ and $o(n)$ additional space. One problem with all of the above approaches is that they do not distinguish between the labels of internal and leaf nodes; in practice, XML data has very few unique internal node labels, but many unique leaf node labels. Since these schemes use constant size labels, the large number of unique text nodes in an XML document can cause a dramatic blowup in space usage. Furthermore, Raman et al [31] took little
consideration on minimizing accesses to secondary storage, which is a concern for supporting efficient queries on large dataset.

The work most related to this paper for succinct data structure is by Geary et al [13, 14]. Geary et al [14] first used a static approach that decomposed XML into two tiers of trees. Their structure supports all operations in $O(1)$ time using an asymptotically optimal $2n + o(n)$ bits of space. However, they used a fixed number of bits for every label, and did not address the vastly different size of alphabets for internal node labels (element labels) and leaf node labels (text data) found in practical XML data. Furthermore, they partitioned the tree in such a way that a node can appear multiple times in the representation, which makes it non-trivial to generalize the structure to support update operations.

As all the different succinct representations above recursively divide their structure into too many layers, their constant time overhead is too large to appeal to practical use. Geary et al later propose a totally different approach [13], which is the first practical implementation of succinct representation. Their representation is considered conceptually much simpler than their previous work, which makes implementation actually possible. However, their approach and implementation only represent topology without labels, and they showed that their implementation was several times slower than their reference DOM implementation. The dataset that they used was small, hence scalability is unclear. Therefore, the motivations of this paper include experimenting a practical and efficient implementation of succinct XML representation, and also testing the implementation with large, real datasets with various loading.

### 2.2 XML Compression

To our best knowledge, Lieftek and Suciu [23] proposed the first compressed XML scheme called XMill. They observed that text sharing the same root-to-leaf path often share similar values. Although XMill achieves a good compression ratio, its major drawback is the lack of support for query and update which hinders its broad application in general XML database systems. Various approaches were proposed after XMill and they share similar benefits and drawbacks, e.g., XMP [10].

Related work that share the same motivations with this paper includes Maneth et al [24], Tolani and Haritsa [33], Min et al [26] and Buneman et al [7]. Compared to XMill, XGrind [33] has a lower compression ratio but supports certain types of queries. However, these approaches were proposed after XMill and they share similar benefits and drawbacks, e.g., XMP [10].

Recent approaches proposed after XMill and they share similar benefits and drawbacks, e.g., XMP [10].

Buneman et al [7] separate the tree structure and its data. They then use bi-simulation to compress the documents that share the same sub-tree. Since multiple nodes may be merged into a single node during compression and partial decompression is needed to output the exact results of the merged nodes, they can only support node navigations in linear time (as described in the paper). With a similar idea but different technique, Maneth et al [8, 24] also compress XML by calculating the minimal sharing graph equivalent to the minimal regular tree grammar. In order to provide tree navigations, a DOM proxy that maintains runtime traversal information is needed [8]. Since only the compression efficiency was reported in the paper, both query and navigation performance of their proposed scheme are unclear. Overall, both approaches give very good space performance on XML data with regular structure and support core XPath queries. However, to facilitate efficient query plans, fast set-based query operations and tree navigation operations have to be supported [16]. As a result, these two approaches may not be suitable for efficient processing of a broader class of XML queries.

### 2.3 XML Database

Most XML storage schemes, such as [15, 16, 19, 22], make use of interval and preorder/postorder labeling schemes to support constant time order lookup, but fail to address the issue of maintenance of these labels during updates. Since XML documents are represented as ordered trees, there is a close relation between this problem and the order maintenance problem addressed by Dietz and Sleator [12] and later by Bender [5]. Bender’s algorithm [5] is designed for efficient order maintenance using gaps effectively, i.e., through the traversal of a virtual trie. This paper uses a bit representation of balanced parenthesis for XML topology and maintains the order of the bits during updates. Additionally, we extend Bender’s approach such that it can determine the density threshold for bits’ redistribution without traversing the virtual trie.

Recently, Silberstein et al [32] proposed a data structure to handle ordered XML which guarantees both update and lookup costs. Similarly, the L-Tree labeling scheme proposed by Chen et al [9] addressed the same problem and has the same time and space complexity as [32], however, they do not support persistent identifiers. The major difference between our proposal and these two works is that we try to minimize space usage (and in fact keep the space requirement near the information theoretic minimum) while allowing efficient access, query and update of the database. In this paper, we show that our proposed topology representation will costs linear space while [32] costs $n \log n$ space. Although the scheme proposed by [32] was shown to be I/O efficient, extensions of the scheme for full query and navigation support were not mentioned.

The work most related to this paper regarding databases with efficient storage is from Zhang et al [36]. The succinct approach proposed by Zhang et al [36] targeted secondary storage, and used a balanced parentheses encoding for each block of data. Unfortunately, their summary and partition schemes support rank and select operations in linear time only. Their approach also uses the Dewey encoding (which is a variable length, root-to-leaf path identifier) for node identifiers in their indexes. The drawbacks of the Dewey encoding are significant: updates to the labels require linear time, and the size of the labels is also linear to the size of the database in the worst case. Thus, the storage of the topology can require quadratic space in the worst case.

Finally, there are several recent proposals, such as Grust et al [15], which show that all XPath axes can be handled using a pre-order/postorder labeling. Instead of maintaining these two labels (i.e., two integers), our proposed scheme requires less than 3 bits per node to process all XPath axes, which is an attractive alternative for applications that are both space and performance conscious.

### 3. ISX STORAGE AND TOPOLOGY LAYER

In this section, we describe the succinct storage layer of the ISX system. It consists of three layers, namely, topology layer, inter-
nal node layer, and leaf node layer. In Figure 2(a), we show the topology layer stores the tree structure of the XML document, and facilitates fast navigational accesses, structural joins and updates. The internal node layer stores the XML elements, attributes, and signatures of the text data for fast text queries. Finally the leaf node layer stores the text data of the document. Text data can be compressed by various common compression techniques and referenced by the topology layer. A similar idea has been employed by XMill and was shown to be flexible and efficient. As a result, we will not discuss this layer in detail for the rest of the paper.

3.1 ISX Topology Layer

Jacobson [17] showed that the lower bound space requirement for representing a binary tree is \( \lg(C_n) = \lg(4^n \cdot \Theta(n^{-2})) = 2n + o(n) \) bits, where the Catalan number \( C_n \) is the number of possible binary trees over \( n \) nodes. There exists a one-to-one mapping from binary trees to unranked ordinal trees. As XML documents can be modeled as unranked ordinal trees, it is possible to store the topology portion of a node XML tree using less than \( 2n \) bits. Based on this, if we exclude tag name and text data from an XML document, the tree structure of the document can be represented using one of the many asymptotically optimal encodings described in Katajainen [21] that use exactly \( 2n \) bits.

Our storage scheme is based on the balanced parentheses encoding from [21], representing the topology of XML. Different from [21], our topology layer (Figure 2(a)) actually supports efficient node navigation and updates.

The balanced parentheses encoding used in tier 0 reflects the nesting of element nodes within any XML document and can be obtained by a preorder traversal of the tree: we output an open parenthesis when we encounter an opening tag and a close parenthesis when we encounter a closing tag. In Figure 2(b), the topology of a DBLP XML fragment shown in Figure 1 is represented in tier 0 using the balanced parentheses encoding. In our implementation, we use a single bit 0 to represent an open parenthesis and a single bit 1 to represent a close parenthesis. For clarity, we will omit the bitwise operation implementation details and treat a single bit (parenthesis) like an object.

**Definition:** An excess is the difference between the number of open and close parentheses occurring in a given section of the topology. For instance in Figure 2(b), the excess between the open parenthesis of \( \text{dpub} \) and the close parenthesis of \( \text{@year} \) is 3. The excess between the close parenthesis of the text node ”2003” and \( \text{booktitle} = -1. \) The depth of a node \( x \) in the XML document tree can be calculated by finding the excess between the open parenthesis of \( x \) and the beginning of the document.

There are several benefits of this encoding. First, topological properties (depth, start/end position, preorder/postorder number) topological relations (ancestor/descendant, document order), document traversal, DOM navigation and XPath axes can all be determined using the above notion. Second, we simplify the database by only having a small set of physical operators.

3.2 Representation of Elements and Attributes

We avoid any pointer based approach to link a parenthesis to its label, as it would increase the space usage from \( 2n \) to a less desirable \( \Theta(n \lg n) \). As our representation of the topology also does not include a \( O(\lg n) \) bit persistent object identifier for each node in the document, we must find a way to link the open parenthesis of \( x \) in tier 0 to the actual label itself. To address this, we adopt from Munro’s work [28] although they do not use balanced parentheses encoding. Instead, they control the topology size by using multiple layers of variable-sized pointers, and may require many levels of indirection.

In addition, we make the element structure an exact mirror of the topology structure instead of mirroring to the pointers. This allows us to find the appropriate label for a node by simply finding the entry in the corresponding position at the element structure. As mentioned earlier, a pointer based approach would require space usage of \( \Theta(n \lg n) \), which is undesirable.

The next issue is to handle the variable length of XML element labels. We adopt the approach taken in previous work [33, 36], and maintain a symbol table, using a hash table to map the labels into a domain of fixed size. In the worst case, this does not reduce the space usage, as every node can have its own unique label. In practice, however, XML documents tend to have a very small number of unique labels. Therefore, we can assume that the number of unique labels used in the internal nodes \( E \) is very small, and essentially constant. This approach allows us to have fixed size records in the internal node layer.

We handle other XML constructs, such as processing instruction and comments, in the same way by using the same hash table. But as we want \( E \) to be small, we must not insert character data into the same symbol table, as that would rapidly increase the space used. Thus, we map all character data to an additional label, and handle the actual character data separately.

By limiting the maximum allowed number of unique element and attribute names per XML document to \( E \), we need an extra \( \lg E \) bits of space for each label and \( O(E) \) space for the symbol table.

Note that each element in the XML document actually has two available entries in the array, corresponding to the opening and closing tags. We could thus make the size of each entry \( \frac{1}{2} \lg E \) bits, and split the identifier for each elements over its two entries. However, the two entries are not in general adjacent to each other, and hence splitting the identifier could slow down lookups as we would need to find the closing tag corresponding to the opening tag and decrease cache locality. Hence, we prefer to use entries of \( \lg E \) bits and leave the second entry set to zero; this also provides us with some slack in the event that new element labels are used in updates.

Since text nodes are also leaf nodes, they are represented as pairs of adjacent unused spaces in the internal node layer. We thus choose to make use of this “wasted” space by storing a hash value of the text node of size \( 2 \lg E \) bits. This can be used in queries which make use of equality of text nodes such as \( /*[\text{year}="2003"]*/ \), by scanning the hash value before scanning the actual data to significantly reduce the lookup time.

4. QUERYING AND UPDATE MAINTENANCE

In addition to efficiently storing large volumes of data, an XML database system should also have the following features: 1) direct node navigation operators; 2) XPath query processing interface; and 3) efficient node insertion/deletion mechanism. For the rest of this section, we present algorithms and other auxiliary data structures satisfying the above features, utilising the ISX topology layer. Furthermore, we provide a detailed cost analysis of our proposed approach for the database operators.

4.1 Node Navigation with Topology Layer Primitives

Given an arbitrary node \( x \) of a large XML document, a navigation operator should be able to traverse back and forth the entire document via various step axes of node \( x \). Some frequently used...
step axes for an XML document tree are parent, first-child, next-sibling, previous-sibling, next-following and next-preceding. These step axes can then be used to provide programming interfaces, such as the DOM API, for external access to the XML database.

Algorithm 1 Node Navigation Operators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARENT(node)</td>
<td>Returns the parent node of node.</td>
</tr>
<tr>
<td>FIRSTCHILD(node)</td>
<td>Returns the first child of node.</td>
</tr>
<tr>
<td>NEXTSIBLING(node)</td>
<td>Returns the next sibling of node.</td>
</tr>
<tr>
<td>PREVIOUSSIBLING(node)</td>
<td>Returns the previous sibling of node.</td>
</tr>
<tr>
<td>NEXTPRECEDING(node)</td>
<td>Returns the next preceding node.</td>
</tr>
<tr>
<td>NEXTFOLLOWING(node)</td>
<td>Returns the next following node.</td>
</tr>
</tbody>
</table>

Node navigation operators can be visualised in Figure 3(a), which illustrates how each operator corresponds to the parenthesis position of a particular node in the ISX topology layer. This is further described by the pseudo-code in Algorithm 1, which shows a tight coupling between the ISX topology layer primitives and the navigation operators.

The relationship of the ISX topology layer primitives and the node navigation operators can be visualised in Figure 3(a), which shows a tight coupling between the ISX topology layer primitives and the navigation operators.

Algorithm 2 Primitive Operators for Topology Layer Access

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORWARDEXCESS(start, end, k)</td>
<td>Returns the start, end, and k for the current tag</td>
</tr>
<tr>
<td>BACKWARDEXCESS(start, end, k)</td>
<td>Returns the start, end, and k for the previous tag</td>
</tr>
<tr>
<td>PREV(node)</td>
<td>Returns the previous node of node.</td>
</tr>
<tr>
<td>NEXT(node)</td>
<td>Returns the next node of node.</td>
</tr>
</tbody>
</table>

First-Child. In Figures 3(a) and 3(b), we show that FIRSTCHILD operator maps to a call to the primitive NEXT(node). The NEXT(node) primitive itself is a trivial function, which simply returns the next parenthesis of position node. If the NEXT(node) primitive returns a close parenthesis, then the current node does not have a first child.

Next-Sibling. NEXTSIBLING operator is implemented by first calling the FINDCLOSE primitive, which is then followed by a call to NEXT primitive. FINDCLOSE finds the close parenthesis of the current node (i.e., the close tag of the current element) by invoking FORWARDEXCESS with the position of the current node as start and the size of topology layer as end and 0 as the set excess k. The purpose of FORWARDEXCESS(start, end, k) in Algorithm 2 is to scan forward along the topology layer from start position to end position. Within this range, FORWARDEXCESS returns the position of the parenthesis (open or close) that satisfies the excess k from start. Since excess is the difference of the number of open and close parentheses in a given topology range, it is possible for the parenthesis returned by FORWARDEXCESS to be either an open or close parenthesis.

Next-Following. The NEXTFOLLOWING operator is mapped to a sequence of topology layer primitives, starting with finding the first close parenthesis by calling FINDCLOSE and followed by repeated calls to NEXT, until an open parenthesis is encountered.

Parent. The implementation of this operator is presented in Figure 3(b), where the navigation operator is mapped to a call to the primitive BACKWARDEXCESS(start, end, k). Since BACKWARDEXCESS primitive scans along the topology layer of ISX in the opposite direction to FORWARDEXCESS, it is intuitive for the start position to always be larger than end position. In the case of PARENT operator, the k parameter for BACKWARDEXCESS is set to 2, which ensures that there is a difference of 1 in the depth of current node and the parent node.

Previous-Sibling. This operator maps to primitive PREV in Figure 3(b), which returns the position of the previous open or close parenthesis. If the previous parenthesis is an open parenthesis, it indicates the current node is the first child of the previous node. However, if the previous parenthesis is a close parenthesis, a call to FINDOPEN will locate the position of the open tag of the previous sibling. The primitive FINDOPEN is simply the opposite of FINDCLOSE - it finds the open parenthesis of the current node (i.e., the open tag of the element) by calling BACKWARDEXCESS.

Next-Preceding. NEXTPRECEDING(node) is implemented by repeatedly calling PREV, which traverses backward to find the closing tag of the previous-sibling of node. We then keep on calling PREV to find the first opening tag, which is also the position of the last-descendant of the previous-sibling of node.

4.2 Auxiliary Tiers

Node navigation operators are highly dependent on topology layer primitives such as FORWARDEXCESS and BACKWARDEXCESS. In the worst case, node navigation operators could take linear time. However, we can significantly improve the performance of the topology layer primitives by adding auxiliary data structures (tier 1 and tier 2 blocks) on top of the tier 0 layer described in Section 3.1.

In Figure 4, we present the auxiliary tiers 1 ($T^1$) and 2 ($T^2$),
9(a) Overview of the data structure

9(b) Balanced parentheses encoding of Figure 1

Figure 2: ISX Storage Layer Illustrated

9(a) Node Navigation Operators

9(b) Topology Layer Primitives

Figure 3: Navigation of Storage Layer Illustrated
Algorithm 3 Calculate Local Excess in a Tier 2 Block

\begin{align*}
\text{TIER2LOCALEXCESS}(t2) \quad &\{ \text{for each } t1 \text{ from } t1_{\text{start}} \text{ to } t1_{\text{end}} \} \\
1 &\{ \text{tie}\{t2[2], \text{tier}[2][2], M\} = \{\text{tie}\{t1_{\text{start}}, m, \text{tier}[1][t1_{\text{start}}], M\}\} \\
2 &\text{excess} = \text{tie}\{t1_{\text{start}}, L - \text{tie}\{t1_{\text{start}}, R\} \} \\
3 &\text{for each } t1 \text{ from } t1_{\text{start}} + 1 \text{ to } t1_{\text{end}} \text{ do} \\
4 &\text{if } \text{excess} + \text{tie}\{t1[1], m < \text{tie}\{t2[2], M\} \} \text{ then} \\
5 &\text{tie}\{t1[1], m \leftarrow \text{excess} + \text{tie}\{t1[1], m\} \} \\
6 &\text{if } \text{excess} + \text{tie}\{t1[1], M > \text{tie}\{t2[2], M\} \} \text{ then} \\
7 &\text{tie}\{t1[1], M \leftarrow \text{excess} + \text{tie}\{t1[1], M\} \} \\
8 &\text{excess} = \text{excess} + \text{tie}\{t1[1], L - \text{tie}\{t1[1], R\} \}
\end{align*}

where each tier contains contiguous arrays of tuples, with each tuple holding summary information of one block in the lower tier. The tier 0 in the figure corresponds to the balanced parentheses encoding of the topology of the XML document, which was described in Section 3.

For tiers 1 and 2, each tier 1 block stores an array of tier 0 tuples \(T1_1, T1_2, \ldots, T1_n\), where \(n\) is the maximum number of tuples allowed per tier 1 block. Each tuple \(T1_i\) for \(0 < i \leq n\) is defined as \((L^i, R^i, m^i, M^i, b^i, B^i, D^i)\), where: \(L^i\) is the sum of all \(L^0\) for all tier 1 tuples \(T1^i\) \(= \sum_{t0=0}^{t0} L^0_t\); \(R^i\) is the sum of all \(R^0\) for all tier 1 tuples \(T1^i\) \(= \sum_{t0=0}^{t0} R^0_t\); \(m^i\) is the local forward minimum excess across all of its tier 1 tuples; \(M^i\) is the local forward maximum excess across all of its tier 1 tuples; \(b^i\) is the local backward minimum excess across all of its tier 1 tuples; \(B^i\) is the local backward maximum excess across all of its tier 1 tuples; and \(D^i\) is the total number of character data nodes for all tier 1 tuples \(= \sum_{t0=0}^{t0} D^0_t\).

Although both tier 1 and tier 2 tuples look similar, the values of \(m^1\), \(M^1\), \(b^1\) and \(B^1\) in tier 2 are calculated differently to that of in tier 1. For tier 2, the function TIER2LOCALEXCESS in Algorithm 3 is used to calculate the local minimum/excess and it is not as trivial as the calculation for tier 1 blocks.

Let \(X = (L, R, m, M, b, B, D)\) be a tier 2 tuple, which holds the summary information for the tier 1 tuples \(Y_1, \ldots, Y_n\). To calculate the local forward minimum excess \(X, m\), we know the local minimum excess from the beginning of the first parentheses of \(Y_1\) until the end of \(Y_1\) is the same as \(Y_1.m\), we then assign this value to \(X.m\).

Example 4 In Figure 4, if we need to calculate the minimum forward excess for the tier 2 tuple \(T2_1\), we first assign it to \(T2_2, m = T1_3.m = -1\). Now the excess at the end of \(T1_8\) is \(T1_3.R = 1\) and \(1 + T1_4, m = 1 + (-4) = -3\). As \(-3\) is smaller than \(-1\), \(T2_1.m = -3\).

In the ISX system, the fixed block size for each tier is 4 kilobytes in size. Therefore, each tier 0 block can hold up to 32768 bits and each tier 1 block can hold \(\frac{4 \times 32768}{256}\) tier 0 blocks. Similarly, each tier 2 block can hold up to \(\frac{4 \times 256}{2^3}\) tier 1 blocks, which is equivalent to \(\frac{4 \times 256}{2^3}\) tier 2 blocks. For a 32-bit word machine, there are only 2 tier 2 blocks and in theory, there are \(\Theta(n/\log n)\) tier 2 blocks. Therefore, the worst case for navigational accesses is \(O(n/\log^2 n)\), which is not much of an improvement on \(O(n)\). Fortunately, it is relatively simple to fix this limitation: instead of having 3 tiers, we generalize the above structure in a straightforward fashion to use \(O(\log n/\log \log n)\) tiers. This means that the top-most tier has \(O(n/\log^k n)\) tier 2 blocks, reducing the worst case navigational access time to \(O(\log n/\log \log n)\).

4.3 Improved Topology Layer Primitives

Algorithm 4 Topology Primitives using Auxiliary Structures

\begin{align*}
\text{NEXT}(\text{node}) \quad &\{ \text{if } (Z_{\text{node}} \leq L^0_{\text{node}} + R^0_{\text{node}}) \text{ then} \\
1 &\text{return } Z^0_{\text{node}} + 1 \\
2 &\text{else} \\
3 &\text{return NOT-FOUND} \\
4 &\text{end if} \\
5 &\text{return } E^0_{\text{node}} + [8] \\
\text{FASTFORWARDEXCESS}(\text{start, end, } k) \quad &\{ \text{current} = \text{FastForwardExcess}((\text{start}, B^0 + [8] - 1, k) \\
6 &\text{if } \text{current} \neq \text{NOT-FOUND} \text{ then} \\
7 &\text{return } \text{current} \\
8 &\text{else} \\
9 &\text{for each } T1^i \in B^0_{\text{current}}, \text{ where } T1^i > T0^0_{\text{current}} \text{ do} \\
10 &\text{if } (\text{current} + m^i) \leq k \text{ current } + M^i \text{ then} \\
11 &\text{return } \text{ForwardExcess}(T1^i, B^0_{\text{current}} + [8] - 1, k) \\
12 &\text{else} \\
13 &\text{current } + L^0_{\text{current}} \leftarrow R^0_{\text{current}} \\
14 &\text{for each } T1^i \in B^0_{\text{current}}, \text{ where } T1^i > T0^0_{\text{current}} \text{ do} \\
15 &\text{if } (\text{current} + m^i) \leq k \text{ current } + M^i \text{ then} \\
16 &\text{return } \text{ForwardExcess}(T1^i, B^0_{\text{current}} + [8] - 1, k) \\
17 &\text{else} \\
18 &\text{current } + L^0_{\text{current}} \leftarrow R^0_{\text{current}} \\
19 &\text{end if} \\
20 &\text{end for} \\
21 &\text{end for} \\
22 &\text{return } \text{current} \quad \text{// implemented in the same way as FORWARDEXCESS, but in back}
\end{align*}

FORWARDEXCESS and BACKWARDEXCESS return the position of the first parenthesis matching the given excess \(k\) within a given range \([\text{start, end}]\) (in forward and backward direction respectively).
Using the auxiliary structures (tiers 1 and 2), instead of just a linear scan of tier 0 layer, we can use tier 1 to test whether the position of the parenthesis, matching $k$ excess, lies within the $i$-th tier 0 block, i.e., checking whether $(n_i + e_i) \leq k \leq (M_i + e_i)$, where $e_i$ is the excess between start and the beginning of the $i$-th tier 0 block (excluding the first bit). However, as $|B| = \Theta(|G|)$, there are potentially $n/|B|$ tier 1 tuples to scan. Hence, we use tier 2 to find the appropriate tier 1 block within which excess lies, thus reducing the cost to a near constant in practice.

Using the above approach, we can replace primitives NEXT, FORWARDEXCESS and BACKWARDEXCESS in Algorithm 2 with improved primitives in Algorithm 4.

Furthermore, since the depths of real-world XML documents are generally less than $|B|$ (even the depth of the highly nested Tree Bank dataset [25] is much less than 100). Most matching parentheses lie within the same block, and occasionally are found in neighboring blocks. Therefore, when FAST-FORWARDEXCESS is called from navigation operations, we rarely need to access additional blocks in either the auxiliary data structure or the topology bit array. In the worst case, when the matching parentheses lie within different blocks, we only need to read two tier 1 blocks and two tier 2 blocks for a 32-bit word size machine, which is very small in size.

### 4.4 Querying the Succinct Data Store

**Binary Structural Joins.** We mentioned earlier in Section 3.1, a single scan of the internal node layer automatically provides a region encoding [16, 34, 35] of each node. This allows the system to create candidate node lists for each query node. For example, if we have a XPath query \(/a/b/c\). A single scan of the topology layer creates 3 element lists, each corresponding to a different query node and all candidate element nodes in each list are encoded with region encoding (start, end, depth) information. Once the candidate lists are created, it is a straightforward structural join of any pair of element lists using Stack-Tree family of join algorithms described in [2]. Finally, all of the partial solutions are joined together to form the final answer to the query \(/a/b/c\).

**Stream Style XPath Processing.** Recently, a new approach has been proposed to process XPaths for streaming XML. One of which is TurboXPath [20] and it is based on the query processing model from their previous work [4] on processing XPath for both forward and backward axes. Their main contribution was to process and each handle is then attached to the query node and all candidate element nodes in each list are encoded with region encoding \((\text{start}, \text{end}, \text{depth})\) information. After the initial loading of an XML document, the empty space allocated to leaf nodes will eventually be used up as more data is inserted into the database. Therefore, we need to guarantee an even distribution of empty bits across the entire parentheses array, so that we can still maintain the $O(|B|)$ bound for the number of shifts needed for each data insertion. This can be achieved by deciding exactly when to redistribute empty space among the blocks and which blocks are to be involved in the redistribution process.

To better understand our approach, we first visualize these blocks as leaf nodes of a virtual balanced binary trie, with the position of the block in the array corresponding to the path to that block through the virtual binary trie. Figure 5 shows such a trie, where block 0 corresponds to the leaf node under the path $0 \rightarrow 0 \rightarrow 0$, and similarly block 3 corresponds to the path $0 \rightarrow 1 \rightarrow 1$. For each block, we define:

- $L$: the total number of left parentheses within a block.
- $R$: the total number of right parentheses within a block.
- $\text{DENSITY}(b)$: the density of a block $b$, defined as $\frac{L+R}{|B|}$.

Given the above definition of density for leaf nodes, the density of a virtual node is the average density of its descendant leaf nodes. We then control the empty space within all nodes in the virtual binary trie by setting a density threshold $[\text{min}, \text{max}]$, within which the block densities must lie. For a virtual node at height $h$ and depth $d$ in the trie, we enforce a density threshold of $\left[\frac{1}{2} - \frac{d}{h}, \frac{1}{2} + \frac{d}{h}\right]$. For example, the density threshold range for virtual node $v_0$ in Figure 5 is $\left[\frac{1}{2} - \frac{1}{3}, \frac{1}{2} + \frac{1}{3}\right] = \left[0.33, 0.67\right]$, since the depth for $v_0$ is 2 and the height of the trie is 3.

Each insertion of a node into the XML document adds exactly two consecutive parentheses into a block (occasionally, the insertion will span two adjacent blocks). We maintain the empty space
Algorithm 5 Node Insertion and Order Maintenance Operations

\textbf{INSERT}(x)
1 Rightshift tier0\((x, L_0^x + R_0^x)\) to \((x + 2, L_0^{x+2} + R_0^{x+2})\)
2 tier\((x, x + 1)\) → \{open parenthesis, close parenthesis\}
3 Increment \(L_0^x, R_0^x, L_1^x\) and \(R_1^x\)
4 if \((L_0^x + R_0^x) > |B| - 2\) then
5 \textbf{MAINTAIN}(x)

\textbf{MAINTAIN}(x)
1 \{height, weight, δ\} ← \{lg n, height, 1\}
2 \{min, max\} ← \{B_0, B_0 + |B|\}
3 while \(\frac{\text{min} + \text{max}}{2} \geq \frac{x}{2} + \frac{\text{max} - \text{min}}{2}\) do
4 \text{depth} ← \text{depth} - 1
5 δ ← 2δ
6 min ← \text{Max}(0, \text{min} - δ)
7 max ← max + δ
8 Evenly distribute bits in blocks \([\text{min}, \text{max}]\) and update
the corresponding tier 1 and tier 2 tuples.

after each insertion as follows: if the density of the leaf node ex-ceeds its maximum threshold, then we redistribute occupied bits among a range of leaf nodes by calling the function \textbf{MAINTAIN} in Algorithm 5. This function traverses up the virtual binary trie and stops at the first ancestor node \(v\) which does not have its maximum density threshold violated. We then evenly redistribute all the occupied bits (parentheses) amongst all the descendant leaf nodes of the \(v\). It should be stressed that the trie is a pure visualization of the concept, and that in reality we are simply traversing a sequence of consecutive blocks in the bit array. Thus, each time we traverse up the binary trie, we are merely doubling the range of blocks consid-ered for redistribution. Deletions are handled in a similar manner.

Why do we use the formula above for controlling the density threshold? This is due to two factors: first, in order to guarantee good space utilization, the maximum density of a leaf node should be 1, and the minimum density threshold of root node should be 1/2. Secondly, the density threshold should satisfy the following invariant: the density threshold range of an ancestor node should be tighter than the range for its descendant nodes. This is so that space redistribution for an ancestor node \(v\), the density threshold of all its descendants are also immediately satisfied.

In the worst case, we use 4 bits per node, since the root node can be only half full. Thus, on a 32-bit word machine, we can store at most \(2^{32}/4 = 2^{30}\) nodes. However, by adjusting the minimum root node density threshold, from \(1/2\) to \(\delta\) is possible to store more than \(2^{30}\) nodes by choosing a smaller \(\delta\). In practice, \(\delta\) should be 2 and therefore \(2\times n\) bits in is in effect \(4n\). The factor \(\delta\) should only be less than 2 when the document is relatively static.

Notice that although we shift the parentheses within tier 0 during update, we never need to shift the tuples in tier 1 because the same \(2^{30}\) tuple always corresponds to the same tier 0 block, regardless of its density. Therefore unlike tier 0, we do not need to redistribute tuples within tier 1 (similarly for tier 2) during the update operation.

4.5.2 Updating Auxiliary Tiers

From Section 4.2, the auxiliary tiers may first appear to increase the update costs to \(O(\lg^3 n / \lg \lg n)\), since moving a node requires updating \(O(\lg n / \lg \lg n)\) tiers. However, this overhead can be eliminated by updating the upper tiers once per redistribution, instead of once per node. A simple proof then demonstrates that the overall update cost is unaffected, and remains \(O(\lg^4 n)\).

Updating auxiliary tiers is simple. During the insertions and deletions in a tier 0 block, we simply update the appropriate tuples in the corresponding blocks in the higher tiers. Since the re-distribution process we described in Section 4.5.1 can be seen as a sequence of insertions and deletions, the corresponding updates to the auxiliary tiers do not affect the worst case complexity for updates.

4.5.3 Persistent Identifiers and Indexes

One of the purposes of this paper is to minimize update costs while using theoretically optimal space. By their very definition, an index is an alternative access path for the data, which is a form of redundancy that we are trying to avoid here. While a surprising amount can be done without persistent node identifiers, in some circumstances they are a useful feature that allows for the creation of additional indexes upon the data. It is possible to extend our data structure to support \(O(\lg n)\) bit persistent identifiers using an additional \(2\lg n + c\) bits space, without affecting the asymptotic time and space complexity. This allows any traditional index to be built upon our structure if desired.

As this is not the main focus of the paper, in here we will show one of the simplest approaches that can achieve our claim in the last paragraph. We can support persistent identifiers easily by mirroring the topology layer with an additional linear array of blocks, using \(\frac{30}{32}\) bits per parentheses, that are used to maintain a map from the parentheses to persistent identifiers. We create a second linear array indexed by the persistent identifiers, the entries of which give the absolute excess of the node (including empty bits). Obviously this additional data structure does not affect the original lookup time, and requires one redirection if we use the persistent identifier. For updates, when we need to shift \(n\) parentheses, we also need to update \(n\) records in the array to reflect the new absolute excesses. This asymptotically does not affect the update cost, and hence this simple augmented structure serves our purpose of supporting persistent identifiers with only a constant time difference in space and time.

Although a single scan of the element layer can give us the start position and end position of all element within a query which can be used as twig join. It is still desirable to create index (which itself is a form of redundancy) to elements that access frequently. If a query purely consists of rare elements, it is more efficient to find the nodes from the index.

With the tier 1 and tier 2 structures, it is relatively less expensive to update an index than without such an auxiliary structure. As each node has no persistent label, it is identified by the implicit physical offset of the parenthesis in tier 0. Thus when the physical offset changes, such as insertion or deletion of a nearby node that is located within (but before) the same tier 0, its corresponding index also needs to be updated. However, with tier 1 and tier 2, we can efficiently and easily convert between the implicit address, the rank (absolute preorder number), depth, and its Dewey coding, therefore indexes also have the option of using the different type of addressing instead. Indexes are free to choose either approach that fits the document update pattern.

4.6 Space Cost

Having \(2n\) bits used per node including update, using 32-bits word, we can store as much as \(2^{30}\) nodes. In our implementation we also chose to use four kilobytes sized block. Based on these values, we now discuss the space cost of each component of our storage scheme. Of course, if larger documents need to be stored, we can increase the word size that we use in the data structure and adjust the bit length used on tier 1 and tier 2.

Tier 0: From above, Tier 0 can take up at most \(2^{32/2c} = 2^{17}\) bits space (or \(\frac{256}{\text{bit}} = 2^{17}\) blocks).
TIER 1: We need \( \lg |B| = 15 \) bits for each variable \((L^m, R^m, M^m, B^m, D^m)\) within a \(T^0\) tuple. Each \(T^0\) tuple requires a total of \(7 \lg |B| = 112\) bits including bit alignments and based on this calculation, each tier 1 block can then store up to \(\left\lfloor \frac{2^{17}}{112} \right\rfloor = 292\) \(T^0\) tuples. Since the maximum number of nodes can be stored in tier 0 is \(2^{30}\), then we only need \(\left\lfloor \frac{2^{30}}{292} \right\rfloor = 217\) \(T^0\) tuples to represent all tier 0 blocks and they can be stored in \(\left\lfloor \frac{2^{30} \times 292}{2^{17}} \right\rfloor = 449\) tier 1 blocks.

TIER 2: We need a total of 24 bits for each variable \((L^1, R^1, M^1, B^1, D^1)\) within a \(T^1\) tuple. This is derived from \(\lg |B| + \lg (\frac{30}{|B|}) = \lg (\frac{2^{17}}{|B|})\), where each variable holds the size of a tier 1 tuple and total number of bits required to represent the total number of tuples per tier 1 block. So each \(T^1\) tuple requires a total of \(\left\lfloor \frac{T^1}{T^0} \right\rfloor = 7 \lg (\frac{2^{17}}{292}) = 168\) bits and each tier 2 block holds up to \(\left\lfloor \frac{|B|}{168} \right\rfloor = 195\) \(T^1\) tuples. Thus, we will only need a total of \(\left\lfloor \frac{2^{17} \times 168}{2^{19}} \right\rfloor = 98\) \(T^1\) tuples. Since the maximum number of nodes in tier 2 blocks to store the 449 tier 1 tuples.

Since we only need a maximum of two tier 0 blocks, even for \(2^{30}\) nodes document, we can just keep them in main memory. In fact, the entire tier 1 can also be kept in main memory, since it requires at most \(449 \times 4\) KB < 2MB. In summary, the space required by the topology layer (in bits) is:

\[
\begin{align*}
2en + 14 &\lg |B| + \frac{98 \lg |B| \lg (\frac{2^{17}}{|B|})}{|B|^2} = 2en + o(en)
\end{align*}
\]

and the space required by the internal node layer (in bits) plus the symbol table is: \(en \lg E + O(E)\).

We can use the above equations to estimate the space used by an XML file, using as our example a 100 MB copy of DBLP, which was roughly 5 million nodes. If we assume there are no updates after the initial loading, we can set \(e = 1\). According to the equation, we will have used roughly \(2en = 3MB\) for the topology layer, and \(en \lg E + O(E) = 8MB\), which is consistent to our experimental results (Figure 6) in Section 5. This, of course, disregards the space needed for the text data in the document.

Based on the block size \(|B|\), we know the exact size of tuples and tiers in our topology layer. Therefore, given a bit position \(x\), we can calculate which tier 0 block this bit belongs to and which tier 1 block contains summary information for the tier 0 block. For a given \(x\), Algorithm 6 lists all the calculations needed to find its resident tier 0 to tier 2 blocks and the index within the blocks to get the summary.

4.7 Summary

Different to the succinct data structures mentioned earlier, such as Munro et al. [28], our structure for ISX carries out node navigation without the need to store any reference tables, pointers nor any offsets in any topology layer. All required information is presented implicitly in our data structure. The absence of any offset and pointer between tier 0 blocks is crucial to the efficiency of our storage layer. As a result, summary information can be calculated locally and update operations only affect the residing tier 0 block and its summary blocks.

With ISX storage layer, we can provide for each node in the database, encoding information (such as the start, end, depth), labelling (used in the region encoding scheme) and preorder, postorder numbering (used in [15]) to answer forward and backward axes queries.

Also different to Bender [5]’s algorithm, our approach is single bit based and the empty gaps are not distributed between the stored bits but gather and redistribute as a group. Secondly, our topology summary layer provides density region information for update operations without the need to perform a full scale linear traversal, thus saving one pass of block reads (calculate, gather and redistribute).

5. EXPERIMENTS

The ISX system is implemented in C++ using Expat XML parser\(^9\). In this section, we compare the performance of ISX with other related implementations, namely, XMill [23], XGrind [33], NoK [36] and TIMBER [19]. Experiments were setup to measure various performances according to the feature matrix of these implementations as shown in Table 1.

Due to disk space and some implementation constraints, the experiments were performed using two machines. For experiments involving small datasets (i.e., using the original DBLP, PSD, TreeBank), we used a 3.2GHz AMD Athlon machine with 1GB RAM, 80GB of 7,200 RPM IDE hard drive. For experiments with large datasets (≥ 1GB), we used an Apple G5 2.0 GHz machine with 2.5GB RAM and 160GB of 7,200 RPM IDE hard drive. Binaries for our ISX, XGrind and XMill were obtained by recompiling their sources on OS X using the same compiler and flags. Note that the memory buffer pool of ISX has been fixed to 64MB for all the experiments on either machine. Three XML datasets were used, namely, DBLP [1], Protein Sequence Database (PSD) [3], TreeBank [25]. We found that the experiment results from PSD are very similar to those from DBLP due to their regular, shallow tree structure. Therefore, PSD results are skipped from some plots below for clarity. Large datasets (i.e., ≥ 1GB) were generated by repeatedly duplicating and merging the source dataset, e.g., the 16GB DBLP document contains more than 770 million nodes.

5.1 Storage Size Comparison

Table 2 provides some idea of the compression ratio of ISX when compared to storage size using an XML database system. Figures 6(a) and 6(b) show the growth of the different sections of our data structure using DBLP and TreeBank, respectively. They show

\(^9\)http://www.sourceforge.net/projects/expat/
that the total size consumed by all the tiers is insignificant compared to the text data itself. It also shows that the compression ratio is insensitive to the structure of the data (as the structure of DBLP is much more regular than TreeBank, i.e., TreeBank has much more distinctive root-to-leaf paths than DBLP does).

Table 3: Storage size of ISX (with and without text compression), XMill and XGrind on DBLP

<table>
<thead>
<tr>
<th>Document Size (GB)</th>
<th>ISX (MB)</th>
<th>ISX Compressed (MB)</th>
<th>XMill (MB)</th>
<th>XGrind (MB)</th>
<th>Source Data (MB)</th>
<th>ISX (MB)</th>
<th>ISX Compressed (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.51</td>
<td>0.41</td>
<td>0.30</td>
<td>0.30</td>
<td>256</td>
<td>131.08</td>
<td>104</td>
</tr>
<tr>
<td>2</td>
<td>1.02</td>
<td>0.81</td>
<td>0.58</td>
<td>0.58</td>
<td>500</td>
<td>243.72</td>
<td>192</td>
</tr>
<tr>
<td>4</td>
<td>2.04</td>
<td>1.63</td>
<td>1.16</td>
<td>1.16</td>
<td>750</td>
<td>365.50</td>
<td>289</td>
</tr>
<tr>
<td>8</td>
<td>4.09</td>
<td>3.26</td>
<td>2.30</td>
<td>2.30</td>
<td>1000</td>
<td>487.43</td>
<td>385</td>
</tr>
<tr>
<td>16</td>
<td>8.19</td>
<td>6.53</td>
<td>4.60</td>
<td>4.60</td>
<td>2000</td>
<td>974.69</td>
<td>770</td>
</tr>
<tr>
<td>64</td>
<td>32.77</td>
<td>44.49</td>
<td>18.35</td>
<td>18.35</td>
<td>8000</td>
<td>4052.58</td>
<td>3203</td>
</tr>
<tr>
<td>128</td>
<td>65.54</td>
<td>52.26</td>
<td>36.69</td>
<td>36.69</td>
<td>16000</td>
<td>7797.56</td>
<td>6165</td>
</tr>
</tbody>
</table>

Table 4: Storage size of ISX (with and without text compression), XMill on TreeBank

4 show that XMill has the best compression ratio for both DBLP and TreeBank datasets. Compared to XMill that does not support any direct data navigation and queries, XGrind does allow simple path expressions. Therefore, it has a relatively less attractive compression ratio. In fact, XGrind failed to run on large datasets in our experiments. Both XMill and XGrind have better space consumption as they are primarily designed for read-only data and do not support efficient updates. Furthermore, they only support access to the compressed data in linear time.

Table 3 and 4 show again that ISX is relatively less sensitive to the structure of the data. Although the compression ratio of ISX for TreeBank is not as good as for DBLP, the reason can be found from Figures 6(a) and 6(b) that TreeBank has the text content that is harder to compress (TreeBank text are more random than the DBLP’s). XMill compression ratio on TreeBank is relatively much worse than that on DBLP is due to both the random text content as well as the more complex tree structure of the data.

5.2 Bulk Loading Performance

The performance comparison of bulk loading using ISX, NoK, XGrind and XMill are shown in Figure 7. For the smaller datasets (up to 500MB DBLP), Figure 7(a) shows our ISX system significantly outperforms NoK and TIMBER in loading. It also highlights the scalability of ISX in loading large datasets.

To further test the scalability of loading even larger XML documents, we compared the loading time of ISX and the other well known systems such as XMill and XGrind on 1 to 16 GB of DBLP documents. During the loading process, XGrind failed to load XML documents greater than 100MB. Although Figure 7(b) shows that the loading time for ISX is slower than XMill’s, it still exhibits a similar trend (similar scalability). The gap between the two curves is contributed by the fact that ISX does not compress the XML data as much as XMill does. This results in a larger storage layer than XMill, which will then uses higher number of disk writes.

5.3 Query Performance
When considering the proposed structure as a storage scheme of a full-fledged database system, one must consider its query performance. Figure 8 (with details listed in Tables 5 and 6) shows the query performance of ISX against other schemes. From this experiment, we found that ISX outperforms other systems in either the ISX or ISX Stream (using the TurboXPath approach [20]) modes. The performance of ISX is measured by using binary structural join to perform XPath queries; while ISX Stream execute the same query by scanning ISX topology layer linearly.

Finally, according to our experiments, none of the chosen XML database systems can load the large datasets with the experiment setup. As a result, the experiment shown in Figure 9 is to show the query performance and scalability of ISX for large data. Note that apart from the Stack-Tree structural join algorithm itself, no other query optimization techniques or indexes have been implemented and employed for this experiment. In particular, Figure 9 suggests that our storage scheme can support query processing in very efficient and scalable manner. In fact, the path evaluation for Q1 - Q6 using binary structural join (Stack-Tree family of join-algorithm [2]) and our storage scheme yields a linear performance curve. This is because the binary structural join requires the ISX system to first perform a single scan of the topology layer, which produces lists of solution nodes for each query nodes. Based on the element lists, Stack-Tree structural join algorithm is then performed. As a result of the overhead incurred by the initial construction of element lists, larger database with large number of elements per tag may be slower to process using the binary structural join approach. Overall, Figure 9 shows that the stack-based based join algorithms are scalable when used on the proposed storage scheme.

5.4 Navigation and Document Traversal

![Figure 9: 1 - 16 GB DBLP with ISX w/ Stack Join](image)

Table 5: Test Queries and Final Result Sizes

<table>
<thead>
<tr>
<th>Query #</th>
<th>XPath Expression</th>
<th>1 GB</th>
<th>2 GB</th>
<th>4 GB</th>
<th>8 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>//inproceedings</td>
<td>402667</td>
<td>981484</td>
<td>2102761</td>
<td>4160339</td>
</tr>
<tr>
<td>Q2</td>
<td>//mastersthesis</td>
<td>74</td>
<td>156</td>
<td>315</td>
<td>627</td>
</tr>
<tr>
<td>Q3</td>
<td>//dblp/article</td>
<td>442185</td>
<td>717450</td>
<td>1379946</td>
<td>2680712</td>
</tr>
<tr>
<td>Q4</td>
<td>//inproceedings/title</td>
<td>74</td>
<td>156</td>
<td>315</td>
<td>627</td>
</tr>
<tr>
<td>Q5</td>
<td>//article/(//month/text() = &quot;July&quot;)//title</td>
<td>857309</td>
<td>1729814</td>
<td>3459628</td>
<td>6920136</td>
</tr>
<tr>
<td>Q6</td>
<td>//inproceedings/(//col//pages</td>
<td>796742</td>
<td>1607116</td>
<td>3214232</td>
<td>6428464</td>
</tr>
</tbody>
</table>

Table 6: Intermediate Result Size and Query Processing Time (Seconds)

<table>
<thead>
<tr>
<th>Q</th>
<th>1 GB</th>
<th>2 GB</th>
<th>4 GB</th>
<th>8 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[Int.]</td>
<td>Time</td>
<td>[Int.]</td>
<td>Time</td>
</tr>
<tr>
<td>Q1</td>
<td>402667</td>
<td>7.05</td>
<td>981484</td>
<td>15.55</td>
</tr>
<tr>
<td>Q2</td>
<td>74</td>
<td>4.18</td>
<td>156</td>
<td>8.84</td>
</tr>
<tr>
<td>Q3</td>
<td>442185</td>
<td>7.70</td>
<td>717450</td>
<td>14.32</td>
</tr>
<tr>
<td>Q4</td>
<td>1259976</td>
<td>13.66</td>
<td>2710668</td>
<td>29.12</td>
</tr>
<tr>
<td>Q5</td>
<td>1304883</td>
<td>10.11</td>
<td>2454842</td>
<td>20.21</td>
</tr>
<tr>
<td>Q6</td>
<td>1637575</td>
<td>13.28</td>
<td>3485826</td>
<td>28.47</td>
</tr>
</tbody>
</table>

To test the performance and scalability of random node navigation, we pre-loaded our XML datasets, and for each database, we randomly picked a node and called the node traversal functions (e.g., FIRSTCHILD, NEXTBROTHER) multiple times. The average access time for these node traversal operations are plotted in Figure 10(a). The graph shows that as the database size gets bigger, the running time for these functions remains constant. This is not surprising, since in general most nodes are located close to their siblings, and hence are likely to be in the same block. For example, it generally only takes a scan of a few bits on average to access either the first child node or the next sibling node. Some operations are faster than the others, due to their different implementation complexity (listed in Algorithm 1) and the characteristics of the encoding itself. For instance, as Figure 10(a) shows, FIRSTCHILD performed slightly faster than NEXTBROTHER function, because the first child is always adjacent to a node, whereas its next sibling might be several nodes away.

With fast traversal operations, ISX can traverse XML data in the proposed succinct encoding significantly faster than other XML compression techniques such as XMill, as shown in Figure 10(b). We argue that this feature is important to examine the content of large XML databases or archives.

5.5 Update Performance

The worst case for Algorithm 5 happens when nodes are inserted at the beginning of a completely packed database, i.e., with no gaps between blocks. The insertion experiment was set to measure its average worst case performance by inserting nodes at the beginning of the database. For each experiment, we did multiple runs (resetting the database after each run). The average insertion times (per node) are shown in Figures 11. In Figure 11, we see an initial spike in the execution time for the worst case insertion. This corresponds to the initial packed state of the database, in which case the very first node insertion requires the redistribution of the entire leaf node layer. Clearly, in practice this is extremely unlikely to happen, but the remainder of the graph demonstrates that even this contrived situation has little effect on the overall performance. The graph also shows that the cost of all subsequent insertions increases at a rate of approximately $O(\log^2 n)$. In fact, all subsequent insertions up to 100,000 took no more than 0.5 milliseconds.

Upgrading the values of nodes will not cause extra processing time apart from the retrieval time for locating the nodes to be updated. In case of deletion, the reverse sequence of steps for node insertion will be left as gaps to be filled by
Figure 7: Loading Time Comparison

9(a) ISX vs. TIMBER and NoK (up to 500 MB data)  
9(b) ISX vs. XGrind and XMill (upto 16 GB data)

Figure 8: Query Performance of ISX vs. Other Systems

9(a) DBLP Q1  
9(b) DBLP Q2  
9(c) DBLP Q3  
9(d) DBLP Q4  
9(e) DBLP Q5  
9(f) DBLP Q6
Figure 10: Navigation and traversal performance time of ISX

Figure 11: Insertion time of ISX using 128 MB - 16 GB DBLP
tem, a compact storage scheme could be used as an index storage that can be manipulated entirely in memory and hence substantially improve the overall performance. In this paper, we proposed an elegant succinct data structure for storing XML data.

Our data structure is shown to be exceptionally concise, without sacrificing query and update performance. While having the benefits of small data footprint, experiments have shown that the proposed structure still out-performs other XML database systems and scales significantly better for large datasets. In particular, all standard navigational primitives (parent, first child, and sibling navigation) can run in near constant time. We have demonstrated in our experiments that due to the conciseness of the data structure and built-in document ordering information, a single pass to evaluate joins is an effective evaluation technique. XML compression techniques such as XMIll will certainly have better compression ratios than our proposed database structure. However, they usually have limited features to be served as a database storage (e.g., efficient query processing, updates). Furthermore, as shown in the experiments, our proposed structure allows direct document traversal and queries that are significantly faster and more scalable than previous compression techniques.

There are still a few open issues that need considering. First, we plan to extend our system to cope with traditional database issues such as concurrency and transactions. We believe that our structure lends itself to scalable implementations of both of these concepts and the fact that during updates most of the work is done in a few sequential scans of the structure. We also plan to improve storage utilization for text data. Finally, after discussions with Sebastian Maneth, we believe that integrating the approaches from schema based compression techniques such as [24] with our proposed succinct storage scheme can yield a hybrid representation to gain the best compression ratio for XML data with various degrees of regularity.

7. REFERENCES

[22] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In Proceedings of the


