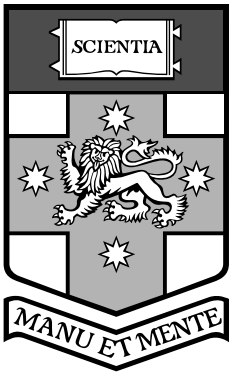
Querying and Maintaining Succinct XML Data

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

Technical Report
UNSW-CSE-TR-0424
July 2004

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES



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 and answers ancestor queries in near constant time. In addition to supporting fast queries, the space requirement 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 efficiently. As an example of the scheme's power, we further demonstrate that the structure can support efficient structural and twig joins. Both formal analysis and experimental evidence demonstrate that the proposed structure is space and time efficient.

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 this information quickly, but also store it compactly. The flexibility of XML makes finding a scheme which satisfies both of these requirements at the same time extremely challenging. 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 costs for all of the desired primitive operations for data processing. There are numerous reasons to maintain such a compact XML representation on secondary storage:

- *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 the XML structures in such databases, whilst also supporting dynamic updates, 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, for example, `libxml`. In the standard computational model, where a pointer takes $O(\lg {}^{1} n)$ bits, using the above approach to represent the topology of n nodes requires $\Theta(n \lg n)$ 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 [10, 12, 13, 18–22], 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. Therefore 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 features on the desiderata:

- *It must support fast navigational operations*: Many XML applications depend upon efficient tree traversal, using a standard interface such as DOM, or make heavy use of queries involving path expressions, for which navigational primitives have been shown to play an important role [11]. 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 [26], 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.

- *It must support efficient insertions and deletions*: Several papers address the space issue by storing XML in compressed form [4, 16, 17, 23]. They also support path expression queries or fast navigational access but do not allow efficient updates, which can be a critical concern in many real database applications. In this paper, we provide a scheme which allows near constant time updates 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 [11], make heavy use of the structural join [1], which relies on a constant time operator to

[1] In this paper, $\lg n$ is the base 2 logarithm of n

determine the ancestor-descendant relationship between two nodes. Thus, any general XML storage scheme should also support such an operator in near constant time.

- *It must be practical*: Most succinct representation techniques require a transdichotomous model (that is, the word size of the machine depends on the size of the data). Pointer and block sizes are usually specified in theoretical terms, whilst neglecting many important practical issues, such as the fixed word size of all real machines. In this paper, we focus on developing a practical storage scheme, using values with 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.

- *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*: As different applications generally need to add specialized indices upon their data, general purpose database systems should use a storage representation which is flexible enough to allow individual users and applications to create extra indices with ease — this means that the scheme must provide simple, efficient, and stable means of referencing items stored using the scheme.

This paper presents a data structure that solves all of the above issues. The proposed structure achieves that by: separating the tree structure from the data, using the concept of *balanced parentheses* to represent the tree structure, and finally adding a compact and yet efficient auxiliary structure to speedup the search. More specifically, the structure uses an amount of space near the information theoretic minimum (for a constant $1 \leq \epsilon \leq 2$ and a document with n nodes, we need $2\epsilon n + O(n)$ bits to represent the topology of the XML document), and handles updates in $O(\lg^2 n)$ time. All navigational operations are supported in $O(\lg n)$ time (in practice, the constant factor is extremely low, so that this is virtually constant). More importantly, the structure is designed to minimize indirections, and hence is secondary storage "friendly". The practical efficiency of the structure is demonstrated through a comprehensive set of experiments.

The rest of this paper is organized as follows: Section 2 summarizes relevant work in the field. Section 3 presents the basics of our succinct representation scheme, without considering the issue of efficient navigation or updates. Efficient updates are discussed in Section 4, and efficient navigation in Section 5. The experimental results are then presented in Section 6, and Section 8 concludes the paper.

2 Related Work

Since XML data can be modeled as trees, storing XML in its succinct form is closely related to succinct tree representations. The earliest space efficient data representations for static unlabeled trees were proposed by Jacobson [12, 13], who showed that the information content of a 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 are all based on static representations, and hence are not easily generalized to support updates. Clark and Munro [5] 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 the navigation to a node's parent, and hence it is not clear how to extend the scheme to support updates. Munro and Raman [18] 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. This was achieved through the use of a balanced parentheses representation, partitioned into three tiers of blocks. 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 [3] 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.

The first work giving a succinct representation for dynamic labeled trees was that of Munro et al [19], which supported binary trees with labels of constant size. However, it did not support updating trees of higher degree. Raman et al [20] extended the $2n$ bit representation of Jacobson [13] 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 and Rao [22] also considered the space and time cost overhead used by the memory manager. This paper also improves 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. However, all the above approaches have the problem of using constant size labels, which is not general enough for XML. They also use a transdichotomous model, where the machine word size fits nicely to the data size, which does not happen on real database. Also, little consideration was placed on minimizing accesses to secondary storage, which is still a concern for large data sets.

A closely related problem for ordered trees is the *order maintenance* problem [2, 9]. Since XML is ordered, the lower bound for order maintenance gives a lower bound for succinct representations of XML.

Recent related work of using succinct representations for XML include [10, 26]. Geary et al [10] used a static approach that decomposed XML into two tiers of trees. It supports all operations in $O(1)$ time using asymptotically optimal space $2n + o(n)$ bits. However, it assumes $\lg|\Sigma|$ bits for every label, and hence does not address the different size of alphabets for internal node labels (element labels) and leaf node labels (text data). More seriously, since it partitions the tree in a way that a node can appear multiple times in the representation, it is not trivial to generalize the structure to support updates.

The approach of Zhang et al [26] 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 can 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.

3 Data Storage

In this section, we give a general overview of our succinct storage scheme for XML data. Sections 4 and 5 will then discuss update handling and optimization in more detail. Our storage structure consists of three main components, as shown in Figure 2:

Topology layer This layer stores the tree structure of the XML document, and layer facilitates fast

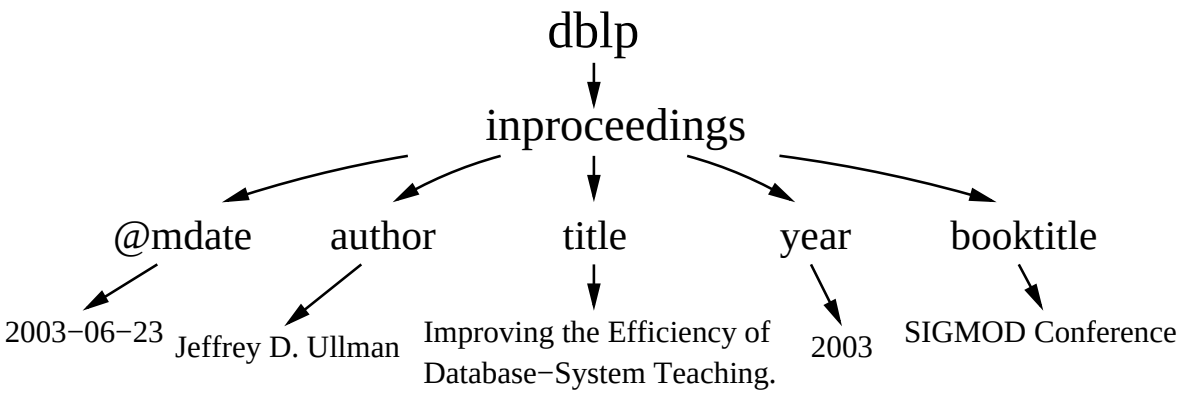


Figure 1: A DBLP XML document fragment

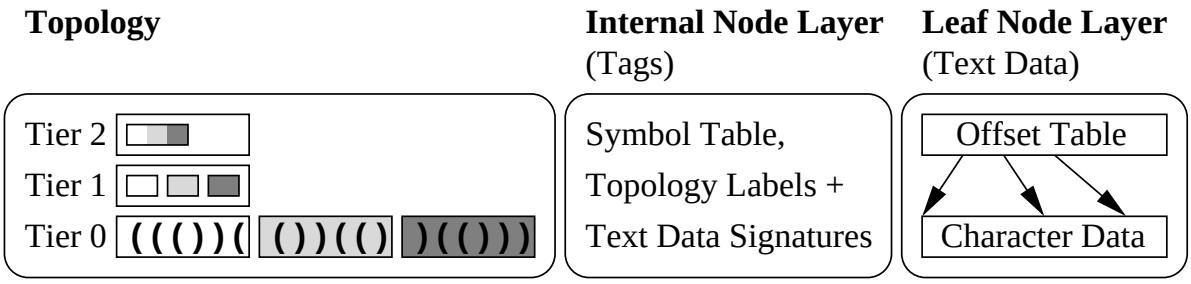


Figure 2: Overview of the data structure

navigational accesses, structural joins and updates.

Internal node layer This layer stores the XML elements, attributes, and signatures of the text data for fast text queries.

Leaf node layer This layer stores the text data in the document.

3.1 Representation of Topology

Jacobson [12] showed that the lower bound space requirement for representing a binary tree is $\lg(C_n) = \lg(4^n \cdot \Theta(n^{-\frac{3}{2}})) = 2n + o(n)$ bits, where the Catalan number C_n is the number of possible binary trees for n number of nodes. As XML documents can be modeled as unranked ordinal trees, we can use the mapping scheme proposed by Jacobson to map XML documents to binary trees. 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 [14] that use exactly $2n$ bits.

For our storage scheme, we use the *balanced parentheses* encoding from Katajainen [14] to represent the topology of XML. This encoding reflects the nesting of element nodes within any XML document. This encoding can be obtained by a preorder traversal of the tree: we output a left parenthesis when we first visit a node and a right parenthesis when we return from the traversal of its descendant nodes. Figure 3 shows the balanced parentheses encoding of the XML document from Figure 1. Herein, we will interchangeably use 0 and $\langle$ to represent left parentheses, and 1 and $\rangle$ to represent right parentheses. We also define:

- $x_{\langle}$: The position of the left parenthesis of node x in the encoding. We will simply write x instead of $x_{\langle}$ when the context is clear. For example in Figure 3, $\texttt{author}_{\langle} = 6$.

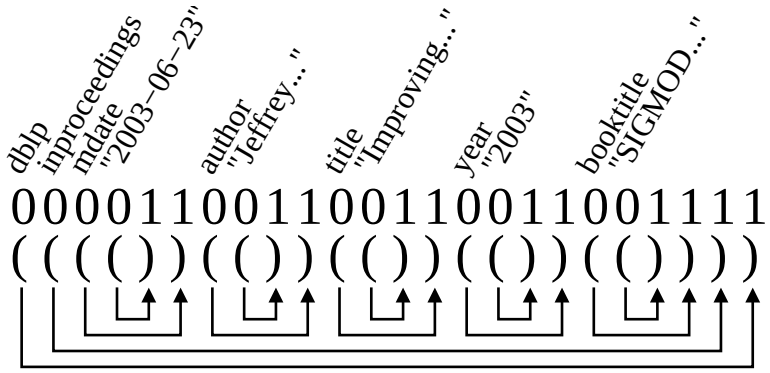


Figure 3: Balanced parentheses encoding of Figure 1

- $x_\rangle$: The position of the right parenthesis of node x in the encoding. For example, $\texttt{title}_\rangle = 13$ in Figure 3.
- *excess*: The excess is the difference between the number of 0s and 1s occurring in a given section of the topology. For instance, in Figure 3, the excess of between $\texttt{dblp}_\langle$ and $\texttt{@mdate}_\rangle$ is 3 and the excess between $\texttt{"2003"}_\rangle$ and $\texttt{booktitle}_\langle$ is -1. Note that measuring excess from the beginning of the document gives the depth of the corresponding node in the tree.

There are two benefits of this encoding:

1. Each node is encoded using a fixed number of bits, which can help to simplify the indexing mechanisms and provides a better fit with secondary storage.
2. The position of the parentheses gives an implicit region algebra representation of the XML document. This allows us to answer ancestor-descendant queries on any two nodes: x is an ancestor of y if and only if $x_\langle < y_\langle < x_\rangle$.

3.2 Representation of Elements and Attributes

As our representation of the topology does not include a $O(\lg n)$ bit persistent object identifier for each node in the document, we must use an approach like that described in Munro [19], in which we make the element structure an exact mirror of the topology structure. This allows us to find the appropriate label for a node by simply finding the entry in the same position of the element structure. 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 [23,26], 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. Figure 4 shows an example of the storage of the element array that mirrors the parentheses array.

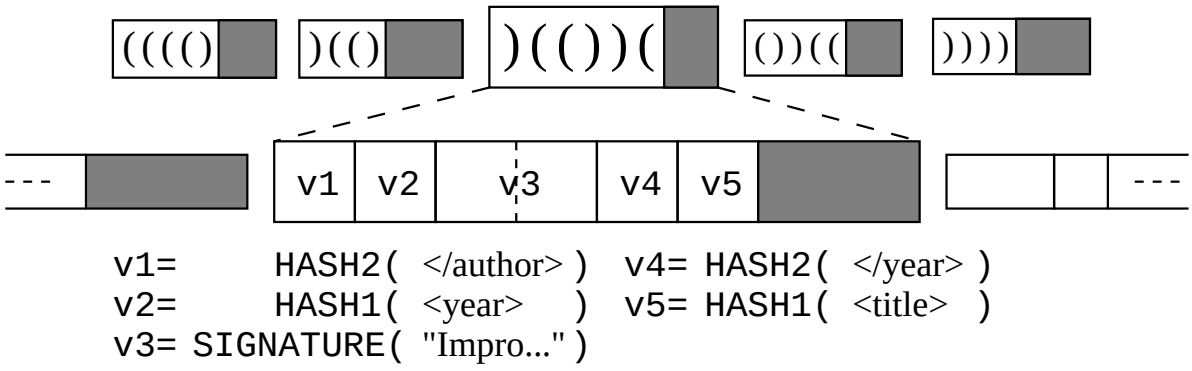


Figure 4: The relationship between the topology and element label structures

Algorithm 1 Unoptimized, linear time, basic topological operations

```
FORWARDEXCESS(start, end, excess)
1   for each current from start to end do
2       if tier0[current] = ⟨ then
3           excess ← excess − 1
4       else
5           excess ← excess + 1
6       if excess = 0 then
7           return current
8   return NOT-FOUND
BACKWARDEXCESS(start, excess)
1   Similar to FORWARDEXCESS but going backward
PREV(node)
1   if node > 0 then return node − 1 else return NOT-FOUND
NEXT(node)
1   if node < |tier0| then return node + 1 else return NOT-FOUND
```

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 `//*[year="2003"]`, by scanning the hash value before scanning the actual data to significantly reduce the lookup time.

3.3 Representation of Text Data

The final layer of our data structure deals with text data storage. We maintain an array of $\lg n$ bit pointers, one for each text node, pointing to its character data. The actual storage of the character data then reduces to the traditional problem of storing variable length records. We have two choices for indexing the array:

- The most concise representation is to pack the array tightly, so that the i-th entry corresponds to the i-th text node. However, this then means that it takes $O(i)$ time to find the i-th value, since we do not explicitly store the text node's position in our structure.

Algorithm 2 Navigation operations

```
FINDCLOSE(node)
1   return FORWARDEXCESS(node, |tier0|, 0)
FINDOPEN(node)
1   return BACKWARDEXCESS(node, |tier0|, 0)
PARENT(node)
1   return BACKWARDEXCESS(node, |tier0|, 2)
FIRSTCHILD(node)
1   if tier0[NEXT(node)] = ⟨ then
2       return NEXT(node)
3   else
4       return NOT-FOUND
NEXTSIBLING(node)
1   if tier0[NEXT(FINDCLOSE(node))] = ⟨ then
2       return NEXT(FINDCLOSE(node))
3   else
4       return NOT-FOUND
```

- A less concise representation would be to make the array's structure mirror that of the element label layer. Then, given a position in the element label array, we could find the corresponding entry quickly. However, this would waste space for the entries corresponding to non-text nodes.

In our scheme, we choose the first method. The reason for this is that the space savings can be significant, and we will see in Section 4 a way of substantially reducing the lookup time. In the worst case, the storage requirement of this method is $\frac{n}{2} \lg n$ bits, because potentially half of the nodes can be character data. In practice, the number of text nodes in XML is within a constant factor of the number of element nodes, so this layer generally uses $\Theta(n \lg n)$ bits space. However, the space requirement is much reduced by treating elements and text data separately. For instance, if we assumed that the number of elements is c times the number of text nodes, and that S was the amount of space taken by considering element nodes and text nodes together, then our scheme would use approximately $S/(c+1)$, a significant space saving for large c.

3.4 Navigational Operations

We now give a brief description of how one may implement navigational operations on this storage scheme. The functions in Algorithm 1 are the basic access operations. If x is the position of a parentheses in an array of balanced parentheses, then the function NEXT(x) returns the position of the next parentheses in the array (for this simple data structure, this is a trivial function). The function PREV(x) is defined analogously. The function FORWARDEXCESS(*start*, *excess*) will scan forward from *start* along the array and return the position *end* of the first parenthesis satisfying the given excess from *start*. Function BACKWARDEXCESS$(start, excess)$ scans backward along the array and returns the first position *end* such that the excess between *end* and *start* is equal to *excess*.

Apart from the basic access operations mentioned in Algorithm 1, other essential navigational operations are shown in Algorithm 2. From observation, we know that the navigational operations are closely tied to the basic access operations. Therefore, the speed of the basic access operations is the determining factor for the performance of our navigational operations. However, both forward and backward excess operations in Algorithm 1 take linear time, which is unsatisfactory. This is addressed in Section 5.

4 Handling Updates

So far, we have treated the balanced parentheses encoding as a contiguous array. This scheme is not suitable for frequent updates as any insertion or deletion of data would require shifting of the entire

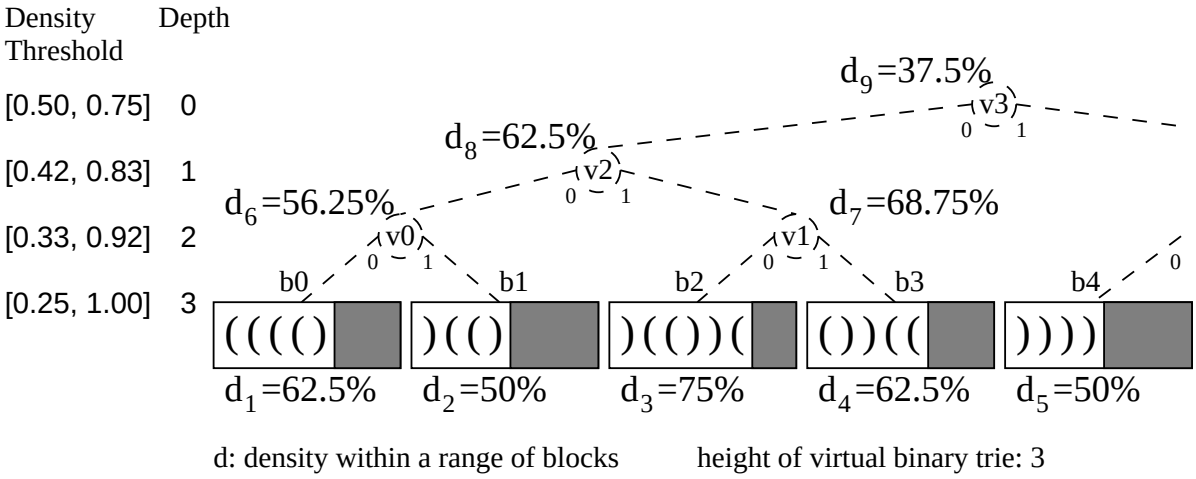


Figure 5: Densities of the parentheses array and the corresponding virtual balanced trie with block size $|\mathcal{B}| = 8$ and $\text{height} = 3$.

bit array. In this section, we present a small modification to our storage scheme, that changes the space usage from $2n$ to $2\epsilon n$, where $\epsilon \geq 1$, so that we can efficiently accommodate frequent updates.

4.1 Empty Space and Density Thresholds

It is obvious that in order to efficiently handle frequent updates, we need to have some empty space within the array to minimize the chance of shifting the entire array. In our approach, we first divide the array into blocks of $|\mathcal{B}|$ bits each, and store the blocks contiguously. Within each block, we leave some empty space by storing them at the rightmost portion of each block. Now, we only need to shift $O(|\mathcal{B}|)$ entries per insertion or deletion. We can control the cost of shifting by adjusting the block size. Section 5 will discuss in detail how we can use auxiliary structures to keep track of the total number of parentheses per block.

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 $O(|\mathcal{B}|)$ 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.
- DENSITY(b): the density of a block b, defined as $\frac{L+R}{|\mathcal{B}|}$.

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 $[min, max]$, within which the block densities must lie. For a virtual node at height h and depth d in the virtual trie, we enforce a density threshold of $[\frac{1}{2} - \frac{d}{4h}, \frac{3}{4} + \frac{d}{4h}]$.

Algorithm 3 Insertion and maintenance operations

INSERT(x)
1 Rightshift $tier0[x, L_x^0 + R_x^0]$ to $[x + 2, L_x^0 + R_x^0 + 2]$
2 $tier0[x, x + 1] \leftarrow \{\langle, \rangle\}$
3 Increment L_x^0, R_x^0, L_x^1 and R_x^1
4 **if** $L_x^0 + R_x^0 > |\mathcal{B}| - 2$ **then**
5 MAINTAIN(x)

MAINTAIN(x)
1 $\{height, weight, \delta\} \leftarrow \{\lg n, height, 1\}$
2 $\{min, max\} \leftarrow \{\mathcal{B}_x^0, \mathcal{B}_x^0 + |\mathcal{B}|\}$
3 **while** $\frac{\sum_{\mathcal{B}_{min}^1}^{\mathcal{B}_{max}^1} L^0+R^0}{(max-min)|\mathcal{B}|} \geq \frac{3}{4} + \frac{d}{4h}$ **do**
4 $depth \leftarrow depth - 1$
5 $\delta \leftarrow 2\delta$
6 $min \leftarrow$ MAX($0, min - \delta$)
7 $max \leftarrow max + \delta$
8 Evenly distribute bits in blocks $[min, max]$ and update the corresponding tier 1 and tier 2 tuples.

For example, the density threshold range for virtual node v_0 in Figure 5 is $[\frac{1}{2} - \frac{2}{4\times3}, \frac{3}{4} + \frac{2}{4\times3}] = [0.33, 0.92]$, since the depth for v_0 is 2 and 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 after each insertion as follows: if the density of the leaf node exceeds its maximum threshold, then we redistribute occupied bits among a range of leaf nodes by calling the function MAINTAIN in Algorithm 3. 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 considered for redistribution. Deletions are handled in a similar manner.

The reader may wonder why 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.

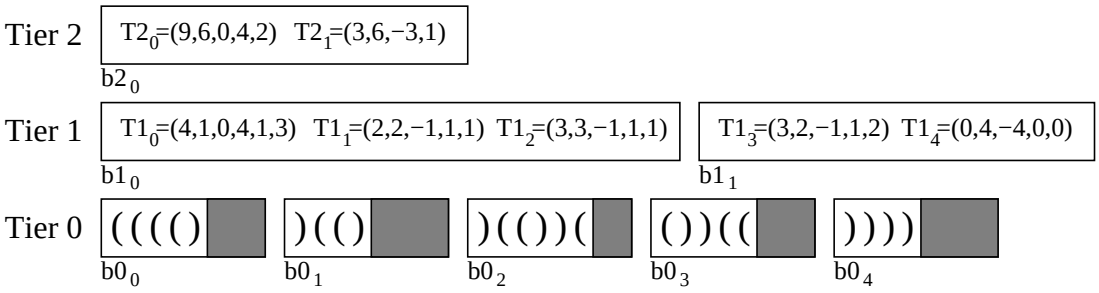
4.2 Space and Time Cost

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 $\frac{1}{2}$ to $\frac{1}{\epsilon}$ it is possible to store more than 2^{30} nodes by choosing a smaller ϵ. In practice, ϵ should be 2 and therefore $2\epsilon n$ bits is in effect $4n$. The factor ϵ should only be less than 2 when the document is relatively static.

The correctness of the above scheme, and its running time, are summarized in the following lemma (with proof omitted, see Bender et al [2]):

Lemma 1 *Given an n node unlabeled higher degree ordinal tree with $2\epsilon n$ bits where $\epsilon \geq 1$, we can obtain update in amortized $O(\lg^2 n)$ time with block size $|\mathcal{B}| = \Theta(\lg n)$.*

In practice, we try to leave approximately 20% of each block empty during insertions. Even when there are bulk insertions in the middle of the document, the lemma above should still guarantee a good worst-case performance.



(L,R,m,M,D): (total left parentheses, total right parentheses, min excess, max excess, total character node)

Figure 6: Example of Tiers of Topology Part

5 Optimizations

This section optimizes navigational operations from linear time (as presented previously) to near constant time. It also analyzes the total space cost and finally, outlines how containment queries can be built on top of the proposed succinct storage.

5.1 Auxiliary Data Structure

In order to speedup the navigational accesses, auxiliary data structures (tier 1 and tier 2 blocks) are added on top of the tier 0 structure we presented in Section 3.1. Both tier 1 and tier 2 contain contiguous arrays of tuples, with each tuple holding summary information of one block in the lower tier.

Each tier 1 block stores an array of tuples $T_1^0, T_2^0, \ldots, T_n^0$, where n is the maximum number of tuples allowed per tier 1 block. Each T_i^0 for $0 < i \leq n$ is defined as $(L^0, R^0, m^0, M^0, D^0)$, where:

L^0**:** the total number of left parentheses of a block.

R^0**:** the total number of right parentheses of a block.

m^0**:** the minimum excess within a single block by traversing the parentheses array from the beginning of a block.

M^0**:** the maximum excess within a single block by traversing the parentheses array from the beginning of a block.

D^0**:** total number of character data nodes.

Using the summary information in T^0 tuples, we can then easily calculate the density of each tier 0 block by using the formula $density = \frac{L^0+R^0}{|\mathcal{B}|}$.

Similar to tier 1 blocks, each tier 2 block stores an array of tuples $T_1^1, T_2^1, \ldots, T_n^1$, where n is the maximum number of tuples allowed per tier 2 block, Each tuple T_i^1 for $0 < i \leq n$ is then defined as $(L^1, R^1, m^1, M^1, D^1)$, where:

L^1**:** the sum of all L^0 for all tier 1 tuples T^0 ($\sum_{i=0}^{|\mathcal{B}|/|T^0|} L_i^0$).

R^1**:** the sum of all R^0 for all tier 1 tuples T^0 ($\sum_{i=0}^{|\mathcal{B}|/|T^0|} R_i^0$).

m^1**:** the local minimum excess across all of its tier 1 tuples.

M^1**:** the local maximum excess across all of its tier 1 tuples.

Algorithm 4 calculate local excess in Tier 2 block

```
TIER2LOCALEXCESS(t2)
1  {t1_start, t1_end} ← {(t2*|T^2|)/|T^1|, ((t2+1)*|T^2|)/|T^1| − 1}
2  {tier2[t2].m, tier2[t2].M} ← {tier1[t1_start].m, tier1[t1_start].M}
3  excess ← tier1[t1start].L − tier1[t1start].R
4  for each t1 from t1_start + 1 to t1_end do
5      if excess + tier1[t1].m < tier2[t2].M then
6          tier1[t1].m ← excess + tier1[t1].m
7      if excess + tier1[t1].M > tier2[t2].M then
8          tier1[t1].M ← excess + tier1[t1].M
9      excess ← excess + tier1[t1].L − tier1[t1].R
```

D^1**:** the total number of character data nodes for all tier 1 tuples ($\sum_{i=0}^{|\mathcal{B}|/|T^0|} D_i^0$).

The three tiers are clearly illustrated in Figure 6, where each tier consists of contiguous fixed size blocks, which in our implementation, are four kilobytes in size. Therefore, each tier 0 block can hold up to 32768 bits and each tier 1 block can hold $\frac{4\text{KB}}{|T^0|}$ tier 0 blocks. Similarly, each tier 2 block can hold up to $\frac{4\text{KB}}{|T^0|}$ tier 1 blocks, which is equivalent to $(\frac{4\text{KB}}{|T^0|})^2$ tier 0 blocks.

Even though both tier 1 and tier 2 tuples look similar, the values of m^1 and M^1 are calculated in a different way to m^0 and M^0. The algorithm to calculate the local minimum/maximum excess in tier 2 is given in Algorithm 4.

Updating both of the auxiliary tiers is fairly easy. 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 redistribution process we described in Section 4 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.

5.2 Using Auxiliary Structures

Recall the function FORWARDEXCESS($start$, end, $excess$) in Algorithm 1 returns the position of the first parenthesis with the given excess within the range $[start, end]$. If we only have tier 0 available, then this scan is linear. However, we can use tier 1 to test whether this value lies within the i-th tier 0 block by checking whether $(m_i^0 + e_i) \leq excess \leq (M_i^0 + e_i)$, where e_i is the excess between $start$ and the beginning of the i-th tier 0 block (excluding the first bit). However, as $|\mathcal{B}| = \Theta(\lg n)$, there are potentially $n/|\mathcal{B}|$ tier 1 tuples to scan. Hence, we use tier 2 find the appropriate tier 1 block within which $excess$ lies, thus reducing the cost to a near constant in practice. This is essentially how we implement this function, with the pseudo-code given as function FASTFORWARDEXCESS in Algorithm 5.

Other operations, such as accessing text nodes, can be implemented in a similar fashion to FORWARDEXCESS, and hence we omit the details.

In practice, most matching parentheses lie within the same block, and occasionally are found in neighboring blocks. This is because the depth of an XML document is generally much less than $|\mathcal{B}|$ (even the depth of the highly nested Tree Bank dataset [7] is much less than 100). Therefore, when FASTFORWARDEXCESS is called from navigation operations, we rarely need to do access additional blocks in either the auxiliary data structure or the topology bit array. In worst case, when the matching parentheses lies within a different block, we only need to read two tier 1 blocks and two tier 2 blocks.

5.3 Space Cost

As we mentioned in Section 4.2, using 32-bit words, we can store 2^{30} nodes; now in our implementation we also chose to use four kilobyte sized blocks. Based on these values, we now discuss the

Algorithm 5 Optimized basic topology operations

```
NEXT(node)
 1 if T^0_node < L^0_node + R^0_node then
 2     return T^0_node + 1
 3 else
 4     if B^0_node is the last tier 0 block
 5         return NOT-FOUND
 6     else
 7         return B^0_node + |B|
FASTFORWARDEXCESS(start, excess)
 1 current ← FORWARDEXCESS(start, B^0_x + |B| − 1, excess)
 2 if current ≠ NOT-FOUND then
 3     return current
 4 for each T^0_i ∈ B^1_current where T^0_i > T^0_current
 5     if current + m^0_i ≤ excess ≤ current + M^0_i then
 6         return FORWARDEXCESS(T^0_i, B^0_{T^0_i} + |B| − 1, excess)
 7     current ← current + L^0_i − R^0_i
 8 for each T^1_j ∈ B^2_current where T^1_j > T^1_current
 9     if current + m^1_j ≤ excess ≤ current + M^0_j then
10         for each T^0_i ∈ B^1_j where T^0_i > T^0_j
11             if current + m^0_i ≤ excess ≤ current + M^0_i then
12                 return FORWARDEXCESS(T^0_i, B^0_{T^0_i} + |B| − 1, excess)
13             current ← current + L^0_i − R^0_i
14     current ← current + L^1_j − R^1_j
```

space cost of each component of our storage scheme. Of course, if larger documents need to be stored, we can simply increase the word size that we use in the data structure.

Tier 0: From Lemma 1 of Section 4.2, tier 0 can take up at most 2^{32} bits space (or $\lceil \frac{2\epsilon n}{|\mathcal{B}|} \rceil = 2^{17}$ blocks).

Tier 1: We need $\lg |\mathcal{B}| = 15$ bits for each variable $(L^0, R^0, m^0, M^0, D^0)$ within a T^0 tuple. Each T^0 tuple requires a total of $5 \lg |\mathcal{B}| = 80$ bits including bit alignments and based on this calculation, each tier 1 block can then store up to $\lfloor \frac{|\mathcal{B}|}{|T^0|} \rfloor = 409$ T^0 tuples, Since the maximum number of nodes can be stored in tier 0 is 2^{30}, then we only need $\frac{2\epsilon n}{|\mathcal{B}|} = 2^{17}$ T^0 tuples to represent all tier 0 blocks and they can be stored in $\lceil \frac{2\epsilon n}{|\mathcal{B}|} / \lfloor \frac{|\mathcal{B}|}{|T^0|} \rfloor \rceil = \lceil \frac{10 \lg |\mathcal{B}| \epsilon n}{|\mathcal{B}|^2} \rceil = 321$ tier 1 blocks.

Tier 2: We need a total of 24 bits for each variable $(L^1, R^1, m^1, M^1, D^1)$ within a T^1 tuple. This is derived from $\lg |\mathcal{B}| + \lg(\frac{|\mathcal{B}|}{|T^0|}) = \lg(\frac{|\mathcal{B}|^2}{5 \lg |\mathcal{B}|})$, 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 $|T^1| = 5 \lg(\frac{|\mathcal{B}|^2}{5 \lg |\mathcal{B}|}) = 120$ bits and each tier 2 block holds up to $\lfloor \frac{|\mathcal{B}|}{|T^1|} \rfloor = 273$ T^1 tuples. Thus, we will only need a total of $\lceil \frac{10 \lg |B| \epsilon n}{|\mathcal{B}|^2} / \frac{|\mathcal{B}|}{|T^1|} \rceil = \frac{50 \lg |\mathcal{B}| \lg(\frac{|\mathcal{B}|^2}{5 \lg |\mathcal{B}|}) \epsilon n}{|\mathcal{B}|^3} = 2$ tier 2 blocks to store the 321 tier 1 tuples.

Since we only need a maximum of two tier 2 blocks, 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 $321 * 4\text{KB} = 1\text{MB}$. In summary, the space required by the topology layer (in bits) is:

$$2\epsilon n + \frac{10 \lg |\mathcal{B}| \epsilon n}{|\mathcal{B}|} + \frac{50 \lg |\mathcal{B}| \lg(\frac{|\mathcal{B}|^2}{5 \lg |\mathcal{B}|}) \epsilon n}{|\mathcal{B}|^2} = 2\epsilon n + o(\epsilon n)$$

and the space required by the internal node layer (in bits) is:

Algorithm 6 Offset calculation for block and indexes within the block in all tiers

$\mathcal{B}_x^0 = \lfloor \frac{x}{|\mathcal{B}|} \rfloor$, $\mathcal{I}_x^0 = \lfloor x \bmod |\mathcal{B}| \rfloor$
$\mathcal{B}_x^1 = \lfloor \frac{\mathcal{B}_x^0 5 \lg |\mathcal{B}|}{|\mathcal{B}|} \rfloor$, $\mathcal{I}_x^1 = \lfloor (\mathcal{B}_x^0 5 \lg |\mathcal{B}|) \bmod |\mathcal{B}| \rfloor$
$\mathcal{B}_x^2 = \lfloor \frac{\mathcal{B}_x^1 5 \lg(\frac{|\mathcal{B}|^2)}{5 \lg |\mathcal{B}|})}{|\mathcal{B}|} \rfloor$, $\mathcal{I}_x^2 = \lfloor (\mathcal{B}_x^1 5 \lg(\frac{|\mathcal{B}|^2)}{5 \lg |\mathcal{B}|})) \bmod |\mathcal{B}| \rfloor$
$T_x^0 = (L_x^0, R_x^0, m_x^0, M_x^0, D_x^0) = (\mathcal{I}_x^0, \ldots, \mathcal{I}_x^0 + 4 \lg |\mathcal{B}|)$
$T_x^1 = (L_x^1, R_x^1, m_x^1, M_x^1, D_x^1) = (\mathcal{I}_x^1, \ldots, \mathcal{I}_x^1 + 4 \lg |\mathcal{B}|)$

$$\epsilon n \lg E + O(E)$$

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 $\epsilon = 1$. According to the equation, we will use roughly $2\epsilon n = 1\text{MB}$ for the topology layer, and $\epsilon n \lg E + O(E) = 8\text{MB}$, which is consistent with the storage size in Table 1. This, of course, disregards the space needed for the text data in the document.

Based on the block size $|\mathcal{B}|$, we know the exact size of tuples and tiers in our topology layer. Therefore, given a bit position $x_\langle$, 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 $x_\langle$, 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.

5.4 Theoretically Fast Navigation

Our experiments will demonstrate that the above scheme has impressive speed in practice, because there are only two tier 2 blocks for a 32-bit word machine. However, in theory, there are $\Theta(n/\lg^2 n)$ tier 2 blocks, and hence the worst case for navigational accesses is also $O(n/\lg^2 n)$, 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 $O(\lg n)$ tiers. This means that the top-most tier has $\Theta(n/\lg^{\lg n} n) = \Theta(1)$ blocks, reducing the worst case navigational access time to $O(\lg n)$, without affecting the overall update cost of $O(\lg^2 n)$.

5.5 Join Queries

Joins (including structural joins and twig joins) are the primitive operations that form complex queries, e.g., ancestor-descendant queries or branched queries. Using our storage scheme, we first scan through the internal node layer to select all of the candidate node lists. As we have mentioned in Section 3.1, a single scan of internal node layer automatically provides a *region encoding* labeling [11, 24, 25] for each node. We can then employ any region encoding based structural join and twig join algorithms to perform the operations. For our experiments in Section 6, we implemented such join algorithms by extending the *skip join* proposed in our previous paper [15] for structural joins and twig joins.

6 Performance Evaluation

This section presents our experimental results, which demonstrate the superior performance of our succinct storage scheme in a variety of ways, such as physical storage size, update cost, and navigational and query performance. All experiments were performed on a PC with a 1.1GHz AMD Athlon

Data Size (MB)	# of Text Nodes	# of Non-Text Nodes	T^0 (Bytes)	T^1 (Bytes)	T^2 (Bytes)	H (Bytes)	E (KB)	D (KB)
1	19,950	27,387	10,752	6,148	5,028	255	79	555
5	107,402	144,859	56,832	6,148	5,028	262	417	2,744
10	209,967	312,205	111,104	6,148	5,028	262	814	5,480
50	1,038,758	1,406,980	548,352	18,436	5,028	284	4,067	27,480
100	2,065,320	2,832,060	1,089,536	30,724	5,028	284	7,990	55,003
500	10,613,430	14,280,334	5,588,992	135,172	5,028	316	41,435	275,513

Table 1: Statistical information of the physical storage of different size XML documents

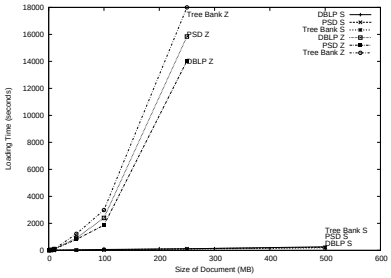
processor, 768MB of main memory, 1GB of swap partition and 40GB of 10,000 RPM SCSI hard disk. The PC was running Debian Linux 3.0, kernel build 2.4.25.

For all experiments, we compared the performance of our storage scheme with the implementation presented by Zhang et al [26], since they demonstrated experimentally that their system outperformed other related systems in almost all cases. We used several data sets covering a wide range of XML applications: the Protein Sequence Database (PSD) [6], DBLP [8] and Tree Bank [7] database. Both PSD and DBLP are extremely regular data sets, whereas Tree Bank's deep recursive tree structure and its over 300,000 unique paths make it an interesting and challenging dataset to handle. We prepared samples of each data set of varying sizes: 5MB, 10MB, 50MB, 100MB and 500MB. The larger sized samples were created by repetitively duplicating and merging the same dataset until it reached the desired size.

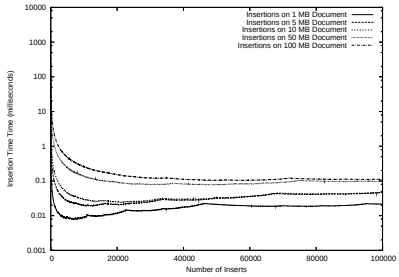
6.1 Physical Storage Size of Data

In our first experiment, we loaded DBLP into our data structure, and measured the sizes of various portions of the structure, which are given in Table 1. Columns T^0, T^1 and T^2 represent the disk usage for tier 0, tier 1 and tier 2. It shows that the size of tier 0 increases the most as document size increases, which is hardly surprising since its size is linearly proportional to the number of elements in the document. Tier 1 grows much more slowly, and for all practical purposes the size of tier 2 remains constant, since gigabytes of XML data would have to be loaded into our database before tier 2 would increase in size, and then only negligibly. The remaining columns give the size of the hash table used to hold the tag names (column H), the internal node layer (column E), and the text data layer (column D). As can be clearly seen, the majority of the space consumed is used up by the internal node label layer (E) and the text data block (D).

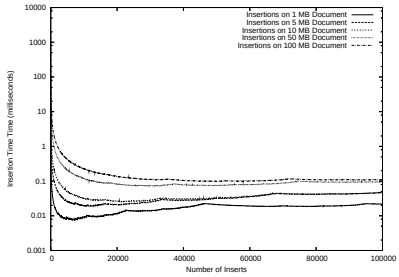
In order to compare our space consumption against previous schemes, we loaded the data sets into both our storage scheme (scheme S) and Zhang's storage scheme (scheme Z). A comparison of the total space used is given in Figure 8(a). Unfortunately, Zhang's implementation [26] was unable to load the 500MB dataset due to insufficient memory, and hence we omitted the storage size for the 500 MB category. Figure 8(a) not only further confirms our expectation of the final disk usage storage size, but it also shows that our storage scheme uses at most 20% of the disk usage of Zhang's storage scheme for all three data sets. Since Zhang's structure is currently the most compact storage system for XML data, this gives a fivefold improvement over the state of the art. This gain can be partially attributed to the fact that we manage to avoid using any indexes for querying or navigating data (apart from the auxiliary tiers defined), whereas Zhang's storage scheme relies on the use of a B-tree to index both element tags and text data.



(a) Loading time for different XML data sets using Scheme (S) vs. Scheme (Z)



(b) Average worst case insertion time using DBLP



(c) Average random insertion time on DBLP

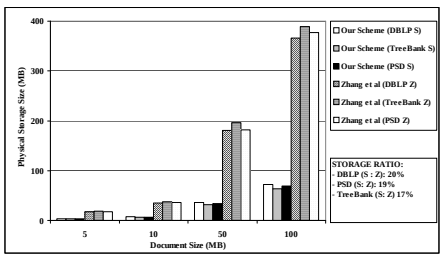
Figure 7: Average run time for updates and loading

6.2 Update Performance

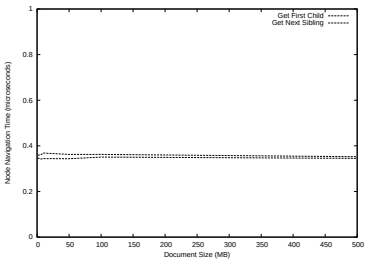
In our second set of experiments, we tested the scalability of our structure under updates by doing frequent insertions in both a worst case manner and in a random manner. The worst case for Algorithm 3 is to insert nodes at the beginning of an already completely packed database, with no gaps between blocks. The random insertion scenario simply inserts a new node as a child of any randomly selected node.

For both worst case and random insertions, we pre-loaded a set of 1, 5, 10, 50 and 100MB of XML documents into our databases and packed each one of them, leaving *no gaps*. For each experiment, we did multiple runs (resetting the database after each run). The average insertion times per node for both worst case and random are shown in Figures 7(b) and 7(c). In Figure 7(b), we see initial spike in the execution time for the worst case insertion. This corresponds to the initial packed state of the database, so that 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 $O(\lg^2 n)$, which conforms to Lemma 1 proposed in Section 4. In fact, all subsequent insertions up to 100,000 took no more than 0.5 milliseconds.

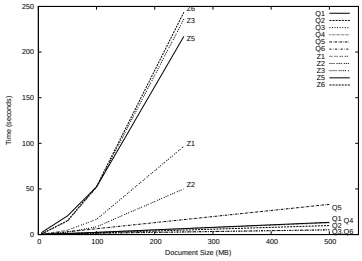
The average random node insertion times are plotted in Figure 7(c). It is interesting to notice how similar Figure 7(c) is to the worst case insertions of Figure 7(b). The initial jump in time for random insertion is also due to the redistribution of the whole leaf node layer, since the database was packed at the beginning. However, after the redistribution process, we have enough gaps between blocks such



(a) Physical storage size



(b) Performance time for accessing next child and next sibling of random nodes using Scheme (S)



(c) Performance times for path evaluation using Scheme (S Q1-6) vs. Scheme (Z Z1-6)

Figure 8: Storage size, navigation time and query evaluation of Scheme (S)

that any random insertion of nodes will at most require redistribution of a few blocks, not the entire leaf node layer. In fact, when a database is fully packed, the initial redistribution will render the random and worst case insertion into the same category. Eventually, when the number of gaps gets smaller, more redistribution is required.

6.3 Node Navigation

To test the performance and scalability of random node navigation, we pre-loaded our XML data sets, and for each database, we randomly picked a node and called NEXTSIBLING and FIRSTCHILD multiple times. The average access time for these two operations are plotted in Figure 8(b). The graph shows that as the database size gets bigger, the running time for FIRSTCHILD and NEXTSIBLING function both remained constant. This is not surprising, since in reality most nodes lie close to their siblings, and hence are likely to lie in the same block. Therefore, it generally only takes a scan of a few bits on average to access either the first child node or the next sibling node. As Figure 8(b) shows, FIRSTCHILD performed slightly faster than NEXTSIBLING function, which again is unsurprising, because the first child is always adjacent to a node, whereas its next sibling might lie some distance away.

6.4 Path Evaluation

One of the most important features of any XML system is its ability to evaluate path expressions quickly. Using both our storage scheme (with the skip-join algorithm [15]) and Zhang's implementation (with their NoK algorithm), we repeated the execution of the queries listed in Table 2 on DBLP, PSD

Table 2: Query Categories

	Data Set	Path Expression
Q1	Tree Bank	`//EMPTY//NP`
Q2	PSD	`//ProteinEntry//refinfo//year`
Q3	DBLP	`//inproceedings//pages`
Q4	Tree Bank	`//EMPTY[.//NP]//VBN`
Q5	PSD	`//ProteinEntry[.//feature-type/ text()="modified site" AND .//status/text()="predicted" AND .//author/text()="Needleman, S.B."]//year`
Q6	DBLP	`//inproceedings[.//i]//ee`

and Tree Bank databases three times. As can be seen, the queries selected test the performance of branch queries and ancestor-descendant queries. As we reported before, our PC ran out of memory when trying to load a 500MB XML document using Zhang's storage scheme. However, our storage scheme was able to process the document without any problem, so we have included the run time for path evaluation for the 500MB data set to show the scalability of both systems.

Figure 8(c) shows the overall run time of each queries on different size databases, using existing skip-join algorithm on Scheme (S) and the NoK algorithm [26] on Scheme (Z). Lines labeled Z_{1-6} are the run-times of the NoK algorithm and the lines labeled Q_{1-6} are the run-times of our skip based join algorithms. The NoK implementation obtained from its author was unable to successfully evaluate Q_4, and hence Z_4 is omitted from the figure.

Figure 8(c) suggests path evaluation is doable on our storage scheme. In fact, the path evaluation for Q_{1-6} using skip-join algorithms and our storage scheme yields a linear performance curve. This is because the skip based join algorithms require the system to first scan through the internal nodes to select sets of candidate nodes before either the structural join or twig join can be performed. However, for most queries, we only need a maximum of one scan of the internal node layer for selecting all necessary candidate nodes. The higher run-time for query Q_5 compared to other queries is mainly due to the testing of text node values, since we have to fetch each text node's value. Overall, Figure 8(c) shows that our proposed skip based join algorithms are significantly more scalable when used on the proposed storage scheme.

7 Acknowledgments

We would like to thank Zhang et al [26] for providing the implementation of their NoK system.

8 Conclusions

A compact and efficient XML repository is critical for a wide range of applications such as those mobile XML repositories running on devices with resources constraints. For a heavily loaded system, a compact storage scheme could be used as an index storage that can be manipulated entirely in memory and hence the overall performance could be substantially improved. In this paper, we proposed an elegant succinct data structure for storing XML data with impressive performance. The performance justifications are supported both theoretically (through formal analysis) and practically (through experiments with real data sets). In particular, all navigational operations can be performed in near constant time, while the storage space is within a constant factor of the information theoretic minimum. Furthermore, insertions and deletions can be performed with a theoretical worst case time of $O(\lg^2 n)$. In practice, these are also close to constant time. Finally, experiments on real data sets proved that the

proposed succinct representation improved all storage, access and update performances over previous results, including the most recent work by Zhang et al. [26], by signifi cant margins.

For future work, we plan to improve the storage space for the text data. Since each data item can be arbitrarily long and the number of different text values within a text document is generally within a constant factor of the number of nodes, it is challenging to store them in a succinct manner while supporting both fast retrievals and updates.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, 2002.

[2] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164, Rome, Italy, Sep 2002.

[3] D. Benoit, E. D. Demaine, J. I. Munro, and V. Raman. Representing trees of higher degree. *Lecture Notes in Computer Science (LNCS)*, 1663:169–180, 1999.

[4] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, 2003.

[5] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.

[6] P. S. Database. http://pir.georgetown.edu/pirwww/search/textpsd.shtml.

[7] T. B. Database. http://www.cs.washington.edu/research/xmldatasets/.

[8] DBLP. http://www.informatik.uni-trier.de/~ley/db/.

[9] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 365–372. ACM Press, 1987.

[10] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[11] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed Mode XML Query Processing. In *VLDB Conference*, pages 225–236, 2003.

[12] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1988.

[13] G. Jacobson. Space-efficient static trees and graphs. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE, 1989.

[14] J. Katajainen and E. Makinen. Tree compression and optimization with applications. In *International Journal of Foundations of Computer Science (FOCS), Vol. 1*, pages 425–447. IEEE, 1990.

[15] F. Lam, W. M. Shui, D. K. Fisher, and R. K. Wong. Skipping Strategies for Efficient Structural Joins. In *DASFAA*, pages 196–207, 2004.

[16] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *SIGMOD Conference*, 2000.

[17] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS: A Queriable Compression for XML Data. In *SIGMOD Conference*, 2003.

[18] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[19] J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succinctly. In *Proc. of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 529–536, 2001.

[20] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. *Lecture Notes in Computer Science (LNCS)*, 2125:426–437, 2001.

[21] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[22] R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. of the 30th International Colloquium on Automata; Languages and Computation (ICALP 2003)*, pages 345–356, 2003.

[23] P. M. Tolani and J. R. Haritsa. Xgrind: A query-friendly xml compressor. In *ICDE*, 2002.

[24] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *TOIT*, 1(1):110–141, 2001.

[25] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.

[26] N. Zhang, V. Kacholia, and M. T. Ozsu. A succinct physical storage scheme for single-pass evaluation of next-of-kin path queries in XML. In *ICDE*, 2004.