Engineering Succinct DOM

O'Neil Delpratt Rajeev Raman Naila Rahman

Department of Computer Science,
University of Leicester, Leicester
LE1 7RH, UK

{ond1,rr29,nyr1}@mcs.le.ac.uk

ABSTRACT

We describe the engineering of *Succinct DOM (SDOM)*, a DOM implementation, written in C++, which is suitable for in-memory representation of large static XML documents. SDOM avoids the use of pointers, and is based upon *succinct* data structures, which use an information-theoretically minimum amount of space to represent an object.

SDOM gives a space-efficient in-memory representation, with stable and predictable memory usage. The space used by SDOM is an order of magnitude less than that used by a standard C++ DOM representation such as Xerces, but SDOM is extremely fast: navigation is in some cases *faster* than for a pointer-based representation such as Xerces (even for moderate-sized documents which can comfortably be loaded into main memory by Xerces).

A variant, SDOM-CT, applies bzip-based compression to textual and attribute data, and its space usage is comparable with "queryable" XML compressors. Some of these compressors support navigation and/or querying (e.g. subpath queries) of the compressed file. SDOM-CT does not support querying directly, but remains extremely fast: it is several orders of magnitude faster for navigation than queryable XML compressors that support navigation (and only a few times slower than say Xerces).

1. INTRODUCTION

XML is increasingly the format of choice for data storage and transmission, particularly when there are complex relationships between data items. However, XML is inherently a verbose representation – for example, the addition of tags to a flat file can easily triple its size. A number of applications require XML documents to be read into main memory; these applications often access the document through the W3C standard Document Object Model (DOM) interface. Unfortunately, typical implementations of DOM produce an in-memory representation that is several times larger than the (already verbose) XML file. This "XML bloat" seriously impedes the performance and scalability of applications that use XML documents.

XML bloat can be addressed – at least for minimizing storage and transmission time – via data compression. The structure inherent in XML files allows regularities that can be exploited for compression purposes to be discovered easily. This has led to the development of specialized XML compressors that achieve excellent compression ratios (see, e.g., [1][4][5][10][14][18]). However, a traditional compression algorithm would require the XML document to be de-compressed in its entirety before it could be processed or queried.

A number of *query-friendly* XML compressors have recently been developed (see, e.g., [1][4][5][10][21][18]). The regularity that makes XML files highly compressible to the right compressor can also be exploited to answer *subpath* or simple XPath queries, for example. The characteristic of a query-friendly compressor is that answering the query involves inspection only of a (usually small) fraction of the XML file, and in principle, only a fraction of the compressed file must be decompressed as well. However, few of these compressors offers support for DOM-like navigation, which can move e.g. from a tag to its sibling in one operation. Indeed, if a node has many descendants, its sibling will be located quite far away in the (compressed or original) file, and query-friendly compressors such as XGRIND or XPRESS may be quite slow when supporting such navigation.

Other compressors such as BPLEX [4] or XBZIPIndex [10] do support navigation using the compressed representation. A detailed experimental evaluation focusing on navigation speeds is not presented in either paper, although [10] claims that navigation operations take a few milliseconds. Also, [4] claims that an individual navigation operation can be performed in time $O(h)$, where h is a parameter that depends upon the XML file being compressed, and is closely related to the size of the compressed representation. Thus, the larger the size of the compressed output, the slower the navigation (BPLEX also does not consider textual data, and instead focuses on compressing the tree structure). However, navigation is not the primary focus of either of these papers: e.g. XBZIPIndex can perform rapid subpath searches; we on the other hand, are focused purely on the (lower-level) DOM operations. From this viewpoint, the excellent compression performance of these query-friendly compressors comes at a significant price in terms of speed (note e.g. that a pointer-based representation takes just a memory access, or tens of nanoseconds, to perform a navigation operation). Finally, we encode XML documents in an implicit manner, eschewing the use of explicit pointers. A number of XML storage schemes for secondary memory use related ideas, for example [2][28]. However, these schemes are not focused on compression and fast navigation, which is our main goal.

1.1 Contributions

We describe *SDOM*, which is a DOM implementation based upon *succinct* data structures. SDOM is particularly suitable for representing large, static XML documents. DOM operations that modify the document are not currently supported, but almost the full DOM Level 3 Core API is currently supported.

EDBT'08, March 25-30, 2008, Nantes, France.

If textual data is kept uncompressed, SDOM uses significantly less space than the original file. A simple variant, *SDOM-CT*, compresses the textual data, and achieves compression ratios competitive with "query-friendly" XML compressors, but worse than the best XML compressors (details in Table 3).

The main advantage of SDOM is that navigation is extremely fast: navigation is, in some common cases such as a document-order traversal of the tree nodes, over three times faster than a standard C++ pointer-based implementation such as Xerces-C (abbreviated to just Xerces in the following). In the rest of the section, the comments we make about Xerces apply to similar C++ DOM implementations. This holds even when both Xerces and SDOM fit comfortably within the main memory of the computer. Parsing an XML file into the representation is also very fast, as is outputting the XML file from its representation.

Even SDOM-CT, tested with more complex navigation, is still only a few times slower than Xerces, and several orders of magnitude faster than other "query-friendly" XML compressors (note however, that SDOM(-CT) only supports DOM operations, and direct support is not provided for more complex operations such as subpath queries).

Obviously, for moderately large documents, whose size is such that the Xerces representation is too large to fit into main memory, SDOM(-CT) can successfully represent a file where Xerces would fail. As Xerces's representation starts to exceed the size of main memory, even SDOM(-CT) begins to approach Xerces.

SDOM is based on *succinctness*, which is related to, but distinct from, data compression. In particular, the size of the representation can be estimated quite accurately using the number of nodes, the number of distinct element and attribute names, and the number characters of textual data in the file. SDOM offers some "compression" even for random files, but misses out on space savings for highly regular files.

XML compressors use insights into the structure of XML files, including regularities in the tree structure [13], the predictive value of the upward path from an element in determining the element [14], and also more generally, the use of *containers* in grouping textual data elements with similar characteristics, and applying specialized compression algorithms to each group [1, 3, 4, 9, 13, 16, 17]. SDOM-CT exploits none of these in any direct way: it effectively concatenates all the textual data in document order into a single string, and represents this string using bzip2 or related libraries [2, 8]. Yet, SDOM-CT compares surprisingly well with regards to compression performance, because:

- If one uses bzip-based algorithms to compress text arranged in document-order, then in most cases, bzip2 does pretty well even relative to specialized compression algorithms applied to containers.
- When using bzip2, the difference between grouping text according to path-order or document-order is limited in most cases.
- In most documents, the tree structure, if represented compactly, as in SDOM(-CT), is already much smaller than the compressed text. Compressing it further using ideas such as [4] yields limited improvements in overall compression ratio.

Succinct data structure building blocks underlying SDOM(-CT) have been studied in isolation in previous works [7][8][11], which already demonstrated their speed. The performance of SDOM(-CT) has been further improved by additional ideas that are specific to the use of these data structures in SDOM.

The remainder of this paper is organized as follows. In Section 2 we discuss existing DOM implementations. In Section 3 we discuss the succinct data structures. In Section 4 we present the architecture of SDOM, together with the application of the succinct data structures. In Section 5, we discuss the outcome of our experimental evaluation, and conclude in Section 6.

2. DOM Implementations

There have been a number of implementations of the DOM API in both C++ and Java. As our focus is on performance, we focus on C++ DOM implementations.

2.1 Xerces

Xerces-C++ [22] is a popular DOM implementation that dopts a pointer-based design of its tree structure. Each node contains pointers to its parent, next-sibling, previous-sibling and first-child node (if it has a child). Other pointers to objects exist, such as the NamedNodeMap object for attribute node handling of elements, in addition basic string arrays for node values and a name pools exist for handling of node names.

2.2 TinyTree

Saxon [20] is an XSLT/XQuery processor which has an internal data structure to represent XML trees called *TinyTree*. The data structure is composed of several arrays of length n, where n is the number of nodes in the tree. There are arrays to represent the depth of each node, the node type information, element and attribute names as namecode values, and attribute and text node values in character sequence stores, etc. Tinytree interfaces with DOM and only supports the read-only methods in DOM. The memory usage of the TinyTree data structure is more than the original XML file size (often by a factor of 2), but is much better than a pointer-based implementation like Xerces.

2.3 DDOM

DDOM (Dictionary-based Document Object Model) [16] is a DOM implementation that supports read-only access on a document in a Java platform. It uses dictionary compression approach to reduce memory usage when representing the document. The core component is a linear table with an index dictionary of the document structure. Elements are referred by a simple number Id, which is referenced in a table. All textual data instances are stored in a managed indexed dictionary which is referred to by the parent Element tag. They claimed to get 30-80% space saving for real world data-centric or regular documents compared to standard DOM implementations such as Xerces or Crimson. However for document-centric XML they only get a saving of 20% to 30% relative to Xerces. Finally, the representation is usually larger than the file size.

3. Preliminaries

3.1 Succinct Data Structures

Succinctness is based upon a simple information-theoretic lower bound: when representing an object from some set of objects C, one requires at least $\log_2 |C|$ bits to uniquely identify the object in the worst case. A *succinct* representation of an object approaches this lower bound. For example, there are roughly $4^n/(2\pi n)^{1/2}$

ordinal trees[1] on n nodes. Thus, in the worst case, an ordinal tree requires at least $2n - O(\log n)$ bits of storage. Indeed, an ordinal tree can be represented as a sequence of $2n$ parentheses – see Figure 2 **(c)** for an example – giving a succinct representation (if one maps parentheses strings to bit strings appropriately). A naïve way to represent ordinal trees is to store two pointers per node (one each to its first child and next sibling). If a pointer takes 32 bits, a succinct representation is 32 times smaller than the naïve representation. Note the difference between succinctness and data compression: a succinct tree representation takes $2n$ bits regardless of the tree being represented – a random tree and a highly regular tree both take the same amount of space. Equally, since there are 2^n bit-strings of length n, an optimal succinct representation of a bit-string is the bit-string itself, regardless of whether the bit-string is compressible or random. While succinct representations are give up some compressibility, by using appropriate data structures, they support operations very rapidly.

3.2 The Bit-Vector Data Structure

A *bit-vector* is a fundamental data structure used in many succinct data structures. It stores a bit-string x of length n, and supports the following operations on x.

- $\mathsf{SELECT}_1(x, i)$: Given an index i, return the position of the ith **1** bit in x.
- $\mathsf{RANK}_1(x, i)$: Returns the number of **1**s to the left of, and including, position i in x.

For example, if x = **1 0 0 1 1 0 1 0** then $\mathsf{SELECT}_1(x, 4) = 7$ (the fourth **1** is in position 7) and $\mathsf{RANK}_1(x, 4) = 2$ (there are two **1**s in positions 1 to 4). SELECT_0 and RANK_0 are defined analogously for the **0** bits in the bit string. From an asymptotic viewpoint, there are bit-vector data structures that use $n + o(n)$ bits[2] to support $\mathsf{SELECT}_{0/1}$ and $\mathsf{RANK}_{0/1}$ in $O(1)$ time [6]. Fast and practical implementations of bit-vectors were studied in [7] and [13]. We use the implementation from [6], which uses $(1 + \varepsilon)n$ bits for any fixed user-specified parameter $\varepsilon > 0$, and supports operations in $O(1/\varepsilon)$ time[3], thus trading off space for time. We choose a point of the trade-off at the "moderately fast" rather than the "space-efficient" end. With these parameter choices, this data structure uses $2n$ bits to support SELECT_1 and $\mathsf{RANK}_{0/1}$. The $\mathsf{RANK}_{0/1}$ operation is very fast (on the order of a memory access) and SELECT_1 is about 2.5 times slower.

3.3 Balanced Parentheses Data Structure

This data structure stores a balanced sequence s of $2n$ parentheses, and supports the following operations:

- ENCLOSE(s, i): Return the position of the opening parenthesis of the parenthesis pair that most immediately encloses the parenthesis in position i of s.
- FINDOPEN(s, i): Return the position of the opening parenthesis that matches the closing parenthesis in position i of the sequence; and return -1 if the parenthesis in position i of s is an open parenthesis.
- FINDCLOSE(s, i): Return the position of the closing parenthesis that matches the parenthesis in position i of the sequence and return -1 if the parenthesis in position i of s is a closing parenthesis.
- INSPECT(s, i): Return the state of the ith parentheses of s, which is either an opening or closing parenthesis

From an asymptotic viewpoint, there are data structures that take $2n + o(n)$ bits and support these operations in $O(1)$ time (see [16],[11]). In practice, the best implementation [11] uses about $2.86n$ bits in all (varying slightly depending upon the precise parenthesis sequence). Again, there is a trade-off between space and speed; the space usage reported is at the most space-efficient parameter setting. We remark here that although all operations are asymptotically $O(1)$ time, they vary in speed: INSPECT is the fastest, FINDOPEN/FINDCLOSE are next, and ENCLOSE is the slowest, being typically 5-6 times slower than FINDOPEN.

3.4 Prefix Sum Data Structure

Given a (static) sequence of positive integers $\boldsymbol{x} = (x_1, \ldots, x_t)$, such that $\sum_{i=1}^{t} x_i = m$, a prefix sum data structure supports the operation SUM($\boldsymbol{x}$, j), which returns $\sum_{i=1}^{j} x_i$. A naïve approach is to pre-compute all prefix sums, and store each prefix sum using $\log m$ bits. This uses at most $t(\log m + 1)$ bits overall and supports SUM trivially in $O(1)$ time. A succinct representation uses $t \log(m/t) + O(t)$ bits, and also supports SUM in $O(1)$ time[7]. We use the implementation of [7], which uses about $4.33t + t \log(m/t)$ bits and supports SUM in $O(1)$ time.

Note that the succinct representation uses space per prefix sum that depends on m/t, the average of the x_is, plus a fixed overhead of 4.33 bits. In contrast, the naïve representation uses space that depends on m, the sum of the x_is.

4. SDOM Architecture

The process of building the SDOM data structure from an XML document is done using a SAX parser, with event handling methods written by us. The parser creates the tree structure of the document consisting of the elements, their contents, CDataSection, PI instructions, on the 'fly' in temporary structures. These are then converted into their final succinct form. For example, if the tree has n nodes, it is stored as a sequence of $2n$ bits, viewed as a parenthesis sequence, during the parsing phase. Once the document is parsed, the parenthesis sequence is converted into a Balanced Parenthesis data structure. In this instance, the intermediate representation is slightly smaller than the final data structures. For other building blocks, the intermediate representation takes more space than the final representation – for example, in SDOM-CT, the textual data is stored temporarily in uncompressed form before it is compressed. Although we do not go into details here, we point out that the parsing is fast, and the intermediate data structures take, at worst, only somewhat more space than the final SDOM representation.

---

[1] An ordinal tree is a rooted tree with arbitrary fan-out at each node, and where the order of the children of the node is fixed – in other words, XML trees.

[2] In this paper, we say that $f(n) = o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$. Thus, $n + o(n)$ bits means a space bound of $(1 + \varepsilon_n)\, n$ bits, where ε_n goes to zero as n grows.

[3] Note the subtle difference to footnote 2: although ε can be chosen to be arbitrarily small, it does not change with n, and lowering ε increases the running time. A space usage of $(1 + \varepsilon)n$ bits is asymptotically worse than $n + o(n)$ bits; however, for any practical values of n (e.g. $n \le 2^{64}$), the $n + o(n)$-bit data structures can be slower and use more space, as noted in [10].

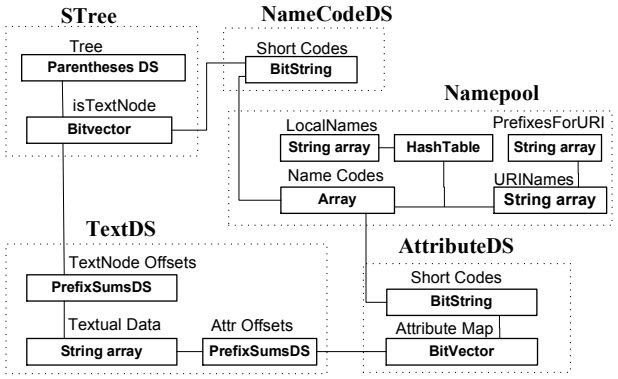


Figure 1. SDOM Architecture

The architecture of SDOM(-CT) consists of 5 core components: the succinct tree data structure (DS), henceforth called *Stree*, the *Namecode* DS, the *Text* DS, the *Attribute* DS, the *Namepool*, and the *hash table* (see Figure 1). We now describe the individual components and their functions.

4.1 Stree

The Stree component consists of an instance of the Balanced Parenthesis data structure (Section 3.3), representing the document tree, together with a bit-vector, which we come back to later. The parenthesis bit-string corresponding to a DOM tree with n nodes is the one created in the obvious way (as already suggested by the examples): traverse the tree in document order, and output an opening parenthesis when a node is first encountered and a closing parenthesis once all its descendants have been visited. In what follows, we use *TP* to denote the tree parenthesis bit-string, and assume that *TP* is stored in a Balanced Parenthesis data structure, which supports the operations in Section 3.3. We now describe how the parenthesis data structure is integrated into SDOM.

We number the nodes 1 to n in document order, and let $\varphi(i)$ denote the position of the ith opening parenthesis in *TP*. In effect, we consider the opening parenthesis at position $\varphi(i)$ as being the representation of node i in *TP*[4]. E.g., in Figure 2 (c), the 8th open parenthesis, which represents node 8, is at position 14 in the parenthesis bit-string, so $\varphi(8) = 14$. Maintaining the association between i and $\varphi(i)$ is critical for fast navigation, as tree navigation operations are implemented by operations on *TP*. For example, to find the parent of a node i, we first find its representation $\varphi(i)$. We then observe that the parent of i is represented by the pair of parentheses that most closely enclose $\varphi(i)$ and $\varphi(i)$'s matching closing parenthesis.

To go from the representation of the parent in *TP* to its document-order number, we invert the mapping φ. In summary:

```
PARENT(i):= φ^-1(ENCLOSE(TP, φ(i))
```

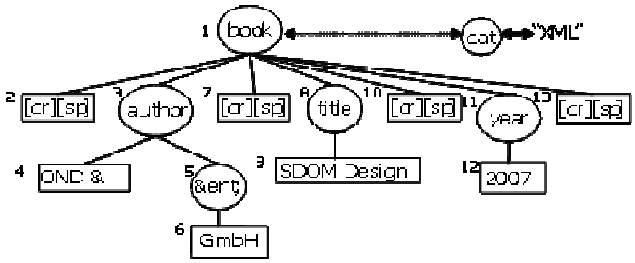
The other operations can similarly be verified:

```
FIRST-CHILD(i):= if INSPECT(TP, φ(i)+1)="("
                 then i+1 else nil
```

[4] Alternatively a node may be associated to a closing parenthesis, or pair of parenthesis, but these are slightly worse alternatives.

```
<book catalogue="XML">
  <author>OND &amp;&plc;</author>
  <title>SDOM Design</title>
  <year>2007</year>
</book>
```

(a)



(b)

φ(i)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
i	1	2		3	4		5	6				7		8	9			10		11	12			13		
	(	(	)	(	(	)	(	(	)	)	)	(	)	(	(	)	)	(	)	(	(	)	)	(	)	)

(c)

Figure 2 - (a): Simple XML fragment. (b): Corresponding DOM tree representation. (c) Parentheses rep of the tree structure with double numbering of nodes. e.g. the 11th node (the element 'year') is at the 20th position in the bit-string.

```
NEXT-SIBLING(i):=
  if INSPECT(FINDCLOSE(TP, φ(i))+1) = "("
     then φ^-1(FINDCLOSE(TP, φ(i))+1) else nil
```

Suppose that in *TP*, an opening (closing) parenthesis is denoted by **1** (**0**). Computing φ and φ^{-1} can be done by augmenting *TP* with data structures to support RANK_1 and SELECT_1, and noting that $\varphi(i) = \mathsf{SELECT}_1(TP, i)$ and $\varphi^{-1}(i) = \mathsf{RANK}_1(TP, i)$. E.g. in Figure 2(c), to get the parent of the 8th node, we first call $\mathsf{SELECT}_1(TP, 8)$, which returns 14. Then, we call ENCLOSE(14), returning 1. Finally, $\mathsf{RANK}_1(TP, 1)$ returns the answer, 1. What we have just described is the standard way to use the balanced parenthesis data structure to represent trees.

The difficulty is that RANK and (particularly) SELECT slow down the navigation operations. In addition, these additional data structures raise the space usage for the tree from $2.86n$ bits to nearly $5n$ bits, and result in a significant time penalty. A better alternative, used in [10], is effectively to number node i using the integer $\varphi(i)$; however, since accessing the information associated with a node requires a document-order numbering, this approach requires a φ^{-1} computation (via RANK) each time any of the (many) DOM methods that access information associated with a node is called. Our approach is superior to both of these, and applies *double-numbering* [7] for the first time to parentheses-based trees. We store with each Node object the pair $(i, \varphi(i))$ and update both components simultaneously (at low cost) during the navigation process. E.g., the `parent` operation becomes:

```
PARENT((i, p)) {
  p' := ENCLOSE(TP, p);          (1)
  i' := i - (p - p' + 1)/2;      (2)
  return (i', p');
}
```

Lemma 1. The pseudo-code for the parent operation, when given the pair $(i, \varphi(i))$ for node i, correctly returns $(i', \varphi(i'))$ where node i' is the parent of node i.

Proof. By our earlier reasoning, line (1) correctly computes $\varphi(i')$, where i' is the parent of i. In line (2), the key observation is that the parentheses that lie in TP between the open parentheses at $\varphi(i')$ and $\varphi(i)$ comprise the representations of the previous siblings of node i and their descendants. This means that there are an equal number of open and close parentheses between positions $\varphi(i')$ and $\varphi(i)$. Furthermore, the open parentheses that lie in between $\varphi(i')$ and $\varphi(i)$ correspond precisely to the nodes that lie in between i' and i in document order. Thus, line (2) subtracts from i the number of nodes that lie between i and its parent in document order, and correctly computes i', the document order number of i''s parent. ■

We modify all navigation operations to work with this "double numbering" in an analogous manner (we omit details). Observe that the root is node 1, and $\varphi(1) = 1$. Thus, we obtain the double numbering of the root directly, and the double numbering of any node reached from the root via navigation operations is correctly computed by induction. The use of double numbering allows the mapping $i \rightarrow \varphi(i)$, which is crucial to navigation, to be maintained incrementally at minimal cost during navigation. Although we do not show experiments here (being a little bit out of the main thrust of this paper) this idea improves the running time for navigational operations by 20% over that reported in [11].

Further speedups can be obtained if one adds as a primitive the operation of going from a node to the next/previous node in document order. This primitive is available in the DOM tree-walker class, and is also required to iterate along the XPATH axes FOLLOWING_NODE or PRECEDING_NODE. We define two new operations on the parenthesis representation:

- NEXTOPEN (TP, x): To return the position and rank of the next opening parenthesis given that we are at the opening parenthesis at position x in the bit-string. Formally, NEXTOPEN(TP, x) returns $(i+1, \varphi(i+1))$ if $i < n$ and NIL otherwise, where $i = \varphi^{-1}(x)$
- PREVIOUSOPEN (TP, x): Analogous.

These are implemented straightforwardly by inspecting bits in the parenthesis sequence. An individual call to NEXTOPEN (PREVIOUSOPEN) skips over at most d closing (opening) parenthesis, where d is the depth of the tree; thus its time complexity is $O(d)$, but with a small constant. In Section 0 we show that using NEXTOPEN is much the fastest option for document-order traversals.

To understand why, we need to understand how going to the next node using the standard navigational operations varies with the location of the current node (we consider document-order traversal – a reverse document order traversal is symmetric). For a non-leaf node, the next node is its first child. The pseudo-code for FIRST-CHILD shows that this only requires the inspection of a bit in TP, and is consequently very fast. For a leaf node, the next node is its following sibling – and locating it is almost as fast as finding the first child of a non-leaf node – except when the leaf node is the last child of its parent. Note that:

Proposition 2. The number of nodes that are last children of their parents equals the number of non-leaf nodes in a tree.

The proportion of non-leaf nodes in XML documents is relatively high – it varies between 33% and 50% of all nodes in the documents in our corpus. Thus, for at least one-third of the nodes, moving to the next node in document order requires significant computation – a series of alternating PARENT and NEXT-SIBLING calls is made, both of which are relatively expensive (generally similar to a few memory accesses). Using NEXT/PREVIOUSOPEN is much faster in this case.

The Stree component also contains a bit-vector we call *isTextNode*, whose ith bit is **1** if the ith node in document order is a text node. Let t be the number of text nodes, and $e = n - t$ be the number of non-text tree nodes. We handle these two kinds of nodes in a different manner. By augmenting the *isTextNode* bit-vector with the RANK_1 operation, we provide a consecutive numbering of text nodes from 1 to t and of non-text tree nodes from 1 to e. For example, if node i is a text node, then $\mathsf{RANK}_1(isTextNode, i)$ gives the ordinal position of node i among the text nodes, considered in document order.

4.2 Textual data offset compression

We now consider the storage of textual data, including:

- Text node values
- Attribute values
- Comment nodes
- CDATASection nodes

For a first approximation, we assume that each of the above categories is treated independently, and we consider only the first category for now. The content of all text nodes is concatenated into a single (virtual) character array C, in the order that text nodes occur in the document.

To access the string associated with the ith text node, we store the sequence $\boldsymbol{l} = (l_1, \ldots, l_t)$, where l_i is the length of the ith text node, in a prefix sum data structure. The ith string then starts at position $\mathsf{SUM}(\boldsymbol{l}, i - 1) + 1$ and ends at position $\mathsf{SUM}(\boldsymbol{l}, i)$.

Let m denote the sum of the lengths of the text nodes and recall that the space usage of the prefix sum data structure is about $4.33\, t + t \log(m / t)$ bits. Since, as m / t, which is the average length of a text node, is quite small (typically 10-11) the space usage works out to about 8 bits per offset, significantly less than storing offsets into C naively, say as 32-bit integers.

We distinguish between three alternate representations of C: In SDOM, C is stored as an uncompressed character array. When concatenating individual text strings to form C, we include the null terminating character of each string. While not necessary for correctness (the offsets demarcate strings already), it allows string values to be returned as pointers into C.

In SDOM-CT, there are two alternate representations of C. The first is in the FM-index [8], which stores C in a compressed form (using a BZIP-related algorithm). It supports the following operations:

- Allows an arbitrary sub-array of C to be extracted (without decompressing C).

- Given a pattern P, counts the number of occurrences of P in C, or locates one occurrence of P in C, in time dependent only on the size of P (the null terminating character for each individual string must be left in C if the search functionality is required).

The other representation is to divide C into blocks of B characters, and to compress each block using bzip2. When a text node needs to be accessed, the block(s) containing it are decompressed. Once a block is decompressed, it is stored in a cache that contains K uncompressed blocks. Subsequent accesses to a cached block do not require decompression, so long as a block is not evicted from the cache because the cache is full (we use a FIFO replacement mechanism). We use K= 4 and B = 16KB; and have a separate cache for the attribute data and the data in the text nodes.

The compression performances of the two representations are roughly similar. The FM-index is recommended if text nodes are not accessed very often, or the access is highly non-local, or the search functionality is desired, but if text nodes are accessed frequently with a degree of locality, the blocked bzip2 is recommended.

The remaining kinds of textual data: attribute values, comment, processing instruction target data and CDataSection nodes are concatenated into a separate virtual array C'. The reason for doing this (rather than concatenating all textual data into a single virtual array) is that the other kinds of textual nodes are typically far less numerous than text nodes, and appear to have different distributions of the lengths. If t' is the number of such nodes, and m' their total length, then by the convexity of the log function,

$$(t + t') \log ((m + m')/(t + t')) \geq t \log m/t + t' \log m'/t',$$

so the space consumption of the offsets is always reduced by separately considering offsets into C and C'. For example, this avoids the risk that one very large comment node raises the average length of all textual nodes in the tree, and thus the space usage of all offsets, were the offsets into C and C' combined.

4.3 Namepool and short-code data structures

The XML names for elements, attributes and other node types are first converted into 32-bit *name-codes*. The data structure for mapping string names to name-codes and back is adapted from Saxon [20] and works as follows: all unique <localname, URI> pairs are stored in a chained hash table, called the *NamePool*, with 2^{10} buckets, where each bucket is (effectively) limited to hold lists of length 2^{10}. A <localname, URI> pair is specified by a 10-bit hash code (specifying the bucket) and a 10-bit offset into the list in that bucket. A further 10 bits are used to encode the namespace prefix.

However, the use of a 32-bit name-code is wasteful: there tend to be very few distinct name-codes in an XML document. E.g., one of our documents, `SwissProt.xml`, has 5166890 elements and attribute nodes in the document, but only 99 distinct name-codes. To save space, we use an additional level of indirection.

We create an array of size e, where e is the number of non-text tree nodes. The ith entry of this array is a *short-code* for the ith non-text tree node in document order. A short-code is a positive integer, interpreted as follows:

- If the ith short-code is 12 or less, then the ith node is not an element node, and the short-code value gives its type.
- If the ith short-code j is 13 or greater, then the ith node is an element node, and $j - 12$ is an index into a table containing all unique namecodes in the XML document, pointing to the entry in this table corresponding to the ith element's name.

The short-codes thus take $\lceil \log(p + 12) \rceil$ bits, where p is the number of distinct name-codes in the document. The array of short-codes is tightly packed, i.e. if short-codes are 7 bits long, then we would render the short code array as an integer array, where each (32-bit) integer would contain 4 complete short-codes and the remaining 32 – 4 * 7 = 4 bits would contain bits from the preceding or succeeding short-codes. The code for extracting an individual short-code was carefully optimized.

As an example, we show how to determine the type of the ith node in document order (this task is the basis of the NODETYPE operation). First, we access the *isTextNode* bitvector: if the ith bit is 1, then the ith node is a text node. If not, then we compute j = $\mathsf{RANK}_0(isTextNode, i)$, to obtain an index from 1..e. Finally, we extract the j-th entry from the short-code array. If this value is 13 or greater, then i is an element node. Otherwise, its type is given by the value of its short-code.

4.4 Attribute Data Structure

Attributes themselves are not apart of the DOM tree, but are associated with their parent elements, and are accessed through a `NameNodeMap` DOM interface. We propose a mapping strategy which maps elements to their attributes and attribute names to their values. Our technique is fairly space-efficient and accessing attributes is fast; the solution is better than, say, including the attribute nodes as "special" children of their parents in the Balanced Parenthesis data structure, particularly if the number of attributes is large.

We now describe the attribute data structure. Recall from Section 4.1 that the *isTextNode* bit-vector numbers non-text tree nodes from 1 to e, where e is the number of non-text tree nodes. We create a sequence of non-negative integers $X = (x_1, \ldots, x_e)$ of length e as follows. If the ith non-text tree node is an element node, then x_i is the count of attributes it has. Some non-text tree nodes have the corresponding $x_i = 0$; others, such as processing instruction, CDATA or comment nodes, have the corresponding $x_i = 1$, as they have associated data that will be treated as a dummy attribute. Letting a be sum of the x_is (i.e. a is the total number of attributes, including dummy attributes), we now show how to represent X to satisfy the following goals:

a) All attributes should be numbered from 1 to a, and the attributes associated with a given non-text tree node should be numbered consecutively.

b) Given a non-text tree node, it should be possible to determine quickly the range of integers that number its (dummy) attributes, if any.

c) Given an integer i, $1 \leq i \leq a$, it should be possible to determine quickly the integer j such that the ith (dummy) attribute belongs to the jth non-text tree node. (This is needed because DOM defines the parent of an attribute node to be the element node with which it is associated.)

These requirements are met as follows. We consider each non-text tree node in document order, and number all its (dummy)

```
<root>
  <U a="val" b="val" c="val" />
  <V /> <-- comment -->
  <W d="val" e ="val">
  <X f="val" g ="val" h="val" i="val">
  <Y j="val">
  <Z />
</root>
```

(a)

r				U	V		//			W					X		Y	Z
	a	b	c			com		e	f		g	h	i	j		k		
	1	2	3			4		5	6		7	8	9	10		11		
0	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**

(b)

Figure 3 - (a) Example XML document with elements and arrangement of attributes. (b) Bit-string of the attribute representation.

attributes consecutively. The attributes (if any) of the first non-text tree node are numbered starting from 1; for any other node, its attributes (if any) are numbered starting from the next available integer. Clearly, all attributes of a node are numbered consecutively, and (a) is satisfied.

For (b) and (c), we represent X as a bit-string as follows. Each value x_i is written in unary (e.g. if $x_i = 4$, then x_i is written as **11110**) and concatenated in order. Note that this bit-string has e **0**s and a **1**s (see Figure 3 (b) for an example), and it is stored as a bit-vector that supports SELECT:

- The attributes of the ith non-text node are numbered from $\mathsf{SELECT}_0(i-1) - i + 2$ to $\mathsf{SELECT}_0(i) - i$. (Observe that $\mathsf{SELECT}_0(i) - i$ gives the number of **1**s before the ith **0** in the bit-string.)
- The parent of the ith attribute is given by $\mathsf{SELECT}_1(i) - i + 1$.

Finally, an array of size a stores the short-codes and node types, analogously to the array of element short codes. Attribute values, as well as textual data associated with some dummy attributes (e.g. text associated with CDATASection or comment nodes) are concatenated and stored in an array C', as described at the end of Section 4.2. Using (a)-(c), the attribute data structure provides the following functionality:

- Get an attribute, given the element number and the index of the attribute;
- get the attribute owner node;
- iterate forwards and backwards through the attribute nodes;
- get a count of attribute nodes belonging to a particular element node and get attribute names and their values. Internally these functions apply to the other node types.

This is a highly space-efficient solution. Existing implementations achieve the functions above using pointers: e.g. in Xerces, element node objects have a pointer to a vector of attribute node objects belonging to that element. Attribute nodes also have pointers to their parent. However, we require only about 2 bits for each non-text tree node or attribute node to maintain the mapping between attribute and element nodes.

4.5 SDOM Interface

A `Node` object in SDOM is fairly lightweight. It comprises a reference to the containing `Document`, and the integers i and $\varphi(i)$. It is important to remember that, unlike a pointer-based DOM implementation, SDOM does not create all `Node` objects in a document when the XML document is parsed. However, SDOM creates a `Node` object whenever a navigational operation is invoked on an existing `Node` object (the implementation currently does not check if an object has previously been created for the same node). Since C++ does not have garbage-collection facilities, even transient `Node` objects stay allocated for the duration of an application unless explicitly freed. Traversing a document via navigation performed through the `Node` interface will therefore result in at least one `Node` object being created for each node in the tree; this collection of `Node` objects will, in many cases, occupy more space than the SDOM representation of the document. To avoid this problem, we recommend the use of the `TreeWalker` class [27] for navigation; this has an iterator-like behaviour, so new `Node` objects are not created by a navigation operation, but it supports all the navigational operations supported by `Node` (our `TreeWalker` implementation does not yet support node filters).

5. Experimental Evaluation

SDOM currently supports the static methods of the W3C DOM Level 3. In this section we draw comparisons of the space usage and running times between SDOM(-CT), Xerces-C and Saxon's TinyTree (as TinyTree is implemented in Java we did not compare running times with TinyTree). We also compare our space usage against XML-specific compressors such as XMILL, XBZipIndex, XPRESS, XQZip and XGRIND. We do not make a detailed comparison with their running times: some are not efficiently queriable (e.g. XMILL), and those that are focus on various kinds of queries rather than navigation, and do not generally report times for navigation. (An exception is [10], where they report navigation operations as taking milliseconds; however, we are several orders of magnitude faster.)

5.1 Setup

We used the Xerces-C v2.5 C++ DOM implementation. The test machine was a Pentium 4 machine with 2GB RAM, 3.4 GHz CPU and a 2MB L2 cache, running Ubuntu 6.06 Linux. The compiler was g++ 3.3.5 with optimization level 2.

For RANK and SELECT we used an optimised version of the Clark-Jacobson bit-vector [10], with parameters B=64 and s=32. We used the parenthesis implementation of [10], with parameter B=128. We do not report the construction times of SDOM in this paper, however it was faster than the Xerces parser.

We tested our algorithms on XML files taken mainly from a standard corpus [23]. `Orders.xml`, `Lineitem.xml`: TPC-H relational database benchmark converted into XML. `Treebank_e.xml`: English sentences tagged with parts of a speech, including encrypted text from the WSJ. `SwissProt.xml`: data from the Swiss-Prot protein database. `DBLP.xml`: DBLP bibliographic data. `XPATH.xml`: is not in [24], but uses the LocusXML schema to represent geospatial information in an XML format, it stores annotated human genomic data.

Table 2 contains the basic statistics of these files such as sizes, number of nodes and the break-down of the node types.

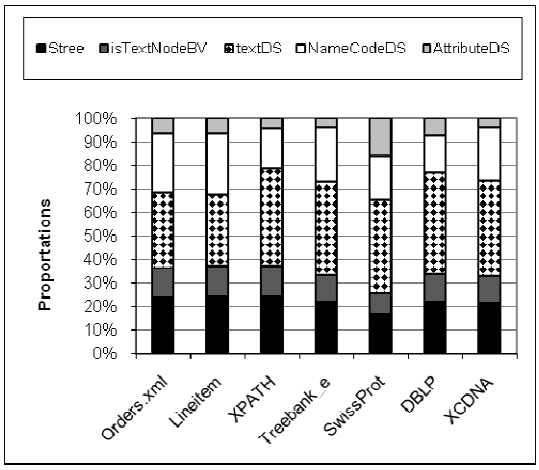


Figure 4 - Space usage distribution of SDOM components excluding text.

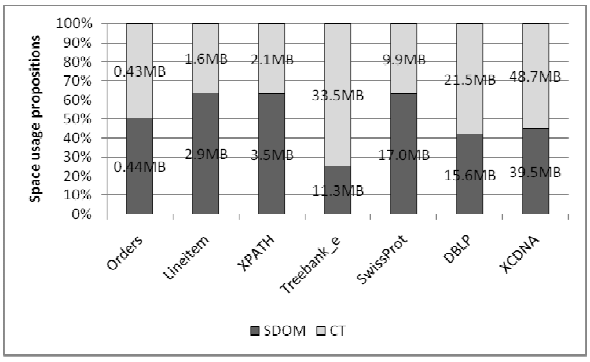
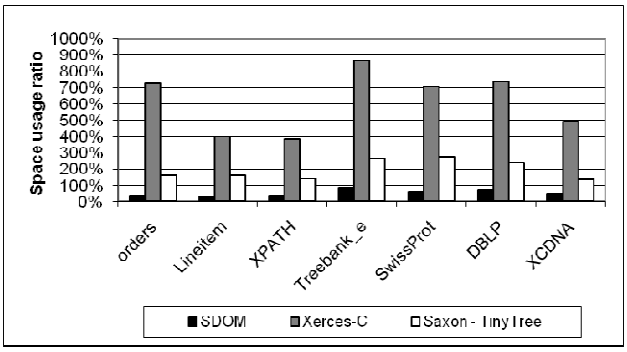


Figure 5 - Space usage of SDOM components from figure 4 (shaded in grey) with compressed text (shaded in black).

5.2 Space Usage

The succinct data structures share a static lookup table that is approximately 2MB in size. We have not added this cost in our figures. For relatively large documents this cost is negligible, and for multiple documents loaded in SDOM we only pay the cost once. Figure 4 shows the space usage of the SDOM components in their relative proportions (excluding the text). Note that the textual offset data structure (shaded in black diamonds in Figure 4) makes up the largest proportion of the space usage: recall that the succinct representation is four times smaller than the naïve one. Indeed, Figure 5 shows that the offsets (assuming naïve storage) would take up more space than the compressed text!

Also, the tree structure, despite being very compactly represented, still takes a fourth of the cost of SDOM (excluding the text). The naïve representation, which would require at least 2 * 32 = 64 bits per node, would be prohibitive. There are other data structures whose space usage is so small that they do not show up in Figure 4. For example, the Namepool data structure is very small because there are few distinct tag names in our files. Figure shows the breakdown of space usage within SDOM-CT. We see that the compressed text is often smaller than the SDOM components, but for irregular or document-centric files such as `Treebank_e.xml` text nodes that do not compress well, the compressed text is larger than the SDOM components.



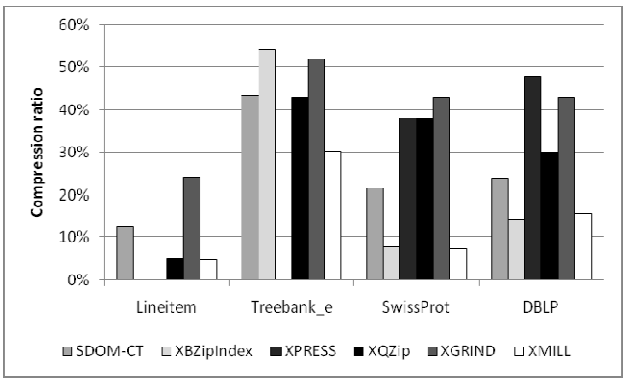


Figure 6 - (Top) Space usage of DOM implementations compared to original file. (Bottom) Compression ratio comparisons with XML compressors

Table 1 - Textual data compression. Uncompressed text data size, compression ratio for bzip2 (text in path-order and document-order) and FMIndex on text in document-order.

		libBzip2-blocks		FMIndex
Files	**Text**	**path-order**	**doc-order**	**doc-order**
Orders	1MB	22%	30%	29%
Lineitem	6MB	21%	31%	26%
XPATH	13MB	9%	13%	13%
Treebank	57MB	42%	45%	56%
SwissProt	49MB	17%	29%	20%
DBLP	71MB	28%	35%	30%

Table 1 compares the compression of text for bzip2 arranged in path-order versus depth-order. Path-order is where the textual data with the same upward path from leaf node to root are arranged together in the concatenated file. Document-order is where we concatenate the textual data as we meet them in a document-order traversal of the tree. We also compare the compression ratio of compressing the textual data with bzip2[5] and FM-Index[6]. We observe that text arranged in document-order compresses comparably well to text in document-order. Also, the compression

[5] Currently using default settings for bzip2 and 8KB blocks.

[6] Using default parameter settings for FMIndex [8, 9]. *b*=2Kb, *B*=32Kb and *f*=0.05.

ratio of bzip2 is roughly similar to FM-Index as mentioned earlier (we exclude the fixed cost of the cache in the bzip2 columns in Table 1, so FM-Index is better than it seems at first sight).

Figure 6 (top) compares the space usage of SDOM with other DOM implementations, and SDOM-CT (bottom) with XML compressors: both query-frendly ones (XPRESS, XQZip, XBZIPIndex and XGrind) and a standard compressor, XMILL. We quote the results for the other compressors from the papers, and have not re-derived them ourselves. We only show files in Figure 6 (bottom) that are reported by the majority of other compressors. The raw data can be found in Table 3. In Figure 6 (top) the space usage ratio is compared with to original file size (in percentage), e.g. SDOM space usage for SwissProt.xml is 59.9% of the file size. We observe in Table 3 that files that would not easily fit into main memory of our test machine under Xerces, such as `XCDNA.xml` (size 607MB, which Xerces increases by a factor of 4) fit comfortably into the main memory using SDOM.

XMILL gives the best compression ratios for all our files (we do not report results from XBZIP, which are similar to XMILL); however XMILL does not support navigation of queries upon the compressed representation. We observe in Figure 6 that SDOM-CT often gives better compression ratios than the other query-friendly XML compressors.

5.3 Running Time

Our tests are based on traversals of XML documents. We always use the SDOM `TreeWalker` interface, and not the SDOM `Node` interface, to avoid creating many transient objects. Even so, there are two different ways of traversing a document in SDOM(-CT):

- using the NextNode method, which is implemented using the optimized NEXTOPEN operation.
- using the standard DOM navigational methods. Over the course of a document-order traversal of an *n*-node tree, this results in a total of *n* calls to each of the methods FirstChild, NextSibling and Parent, thus providing a test that involves a mix of standard navigational operations.

We perform three kinds of traversals: document-order, reverse document order and *upward path enumeration.* The latter works as follows. We perform a document-order traversal using the standard DOM navigational methods. When the main iterator reaches a leaf, an auxiliary iterator traverses the entire upward path from the leaf to the root using the DOM Parent method.

Along with the traversals, we either gather *basic statistics*, which include the count of element and text nodes, or perform a *full test*, which (i) determines the type of each node (ii) checks whether nodes have associated attributes (iii) for nodes with attribute data, or text nodes, we retrieve the node value, and check to see if the value contains a substring that is unlikely to appear (hence forcing the substring search to scan the full node value).

Each test is repeated several times to obtain stable results (50 times for `Orders`, 10 times for `Lineitem`, `XPATH`, `Treebank`, `DBLP` and `SwissProt`, and 2-5 times for `XCDNA`). The file sizes vary widely, and the graphs are geared primarily towards comparing different algorithms on the same file. It is therefore a little hard to see scalability with instance size from the graphs. However, given the theoretical analysis of the running time, no scalability surprises are expected (and the raw data, where shown, confirms this).

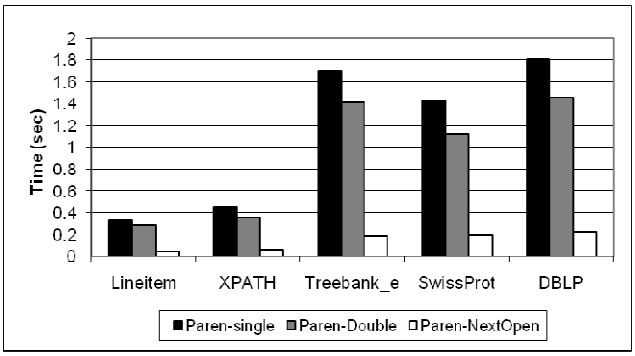


Figure 7 – Average running times of tree traversal using single numbering, double numbering and the nextOpen operation.

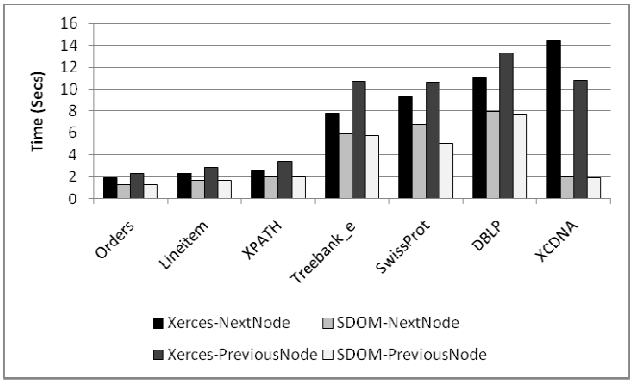


Figure 8 - Running times, document-order and reverse document-order traversals gathering basic statistics, of Xerces and SDOM using NextNode and PreviousNode operations. Average time for XCDNA.

We begin by demonstrating the effectiveness of the optimizations we have made to the parenthesis structure. Figure 7 shows the times taken to traverse (in document order) a tree derived from the tree structure of the XML documents of our set, when this tree is represented as a parenthesis sequence (however, the additional data structures for namecodes, textual data etc. are not created). The trees are traversed in three ways: using the original parenthesis operations, the parenthesis operations using double-numbering and finally, using the using NEXTOPEN operation. We observe double-numbering is about 20% faster than single-numbering. However, using NEXTOPEN is by far the fastest: the improvement over the original parenthesis code is more than 80%. However, NEXTOPEN can only be used for simple traverals.

Figure 8, Figure 9and Figure 10 show various traversals that gather basic statistics (as described above). Our tests are repeated: 50 times for `Orders`, 10 times for `Lineitem`, `XPATH`, `Swissprot`, `Treebank_e` and `DBLP`, with total times reported. For `XCDNA` we report the average time over 5 runs. Figure 8 shows the result of a document-order and reverse document-order traversal, and shows that traversal using SDOM's NextNode/PreviousNode operations was on average 40% faster than Xerces-C. As expected, the gap grows for the largest file, `XCDNA`. Figure 9 shows the result of a document-order and reverse document-order traversal using DOM navigation methods; SDOM is always within a factor of 2 of Xerces, but equals or betters Xerces for XCDNA. Note that SDOM shows very similar performance for document-order and reverse document-order

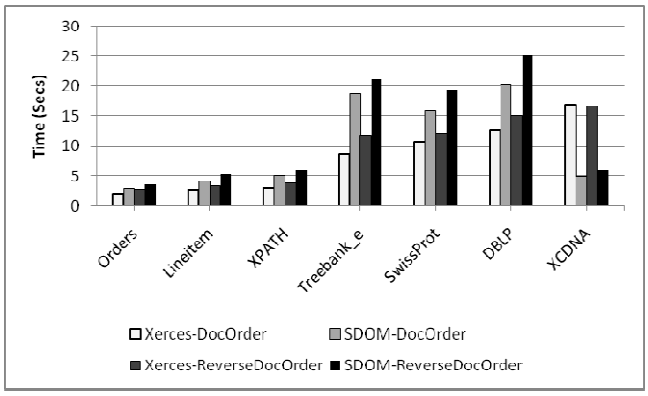


Figure 9 – Running times, for document-order and reverse document-order traversals using DOM navigation, with basic statistics for Xerces and SDOM. Average time for XCDNA.

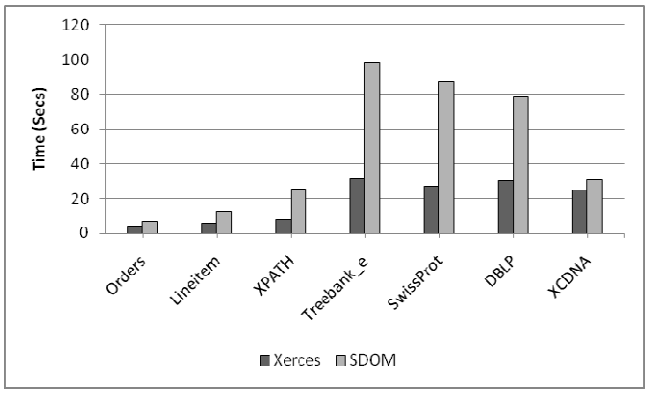


Figure 10 – Running times of Xerces and SDOM for 'upward path enumeration' gathering basic statistics. Average time for XCDNA.

traversals. Figure 10 and Table 4 show the results for an upward path enumeration traversal. This traversal makes a very heavy use of the Parent operation which is (relatively) inefficient in SDOM, and may be considered a "worst case" for SDOM. Even here, SDOM on average was only a factor of 2.5 slower than Xerces.

Figure 11 and Table 5 show the result of a document-order traversal but performing a full test. Observe that even SDOM-CT with the slow DOM navigation is only a few times slower than Xerces, for small files, and for our largest file, the gap starts narrowing rapidly. Particularly noteworthy is the time of SDOM (using the NextNode interface) on XCDNA, which is nearly 3.5 times faster than Xerces.

6. Conclusion

SDOM is a fast in-memory representation of XML documents with a small memory footprint. The current implementation is close to being a plug-in replacement for a standard DOM implementation in any application that does not require dynamic changes to the XML document, with very little penalty in terms of CPU usage. It is therefore not only suitable for handling moderately large (a few GB) size documents on standard PCs, but may also be useful for enabling the use of XML on devices with limited resources, such as smart cards or handheld computers.

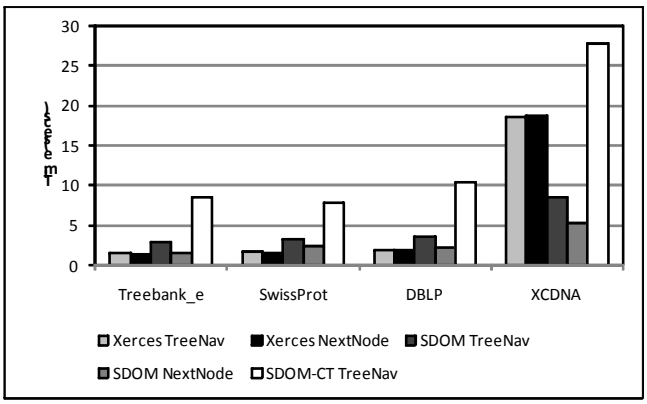


Figure 11 - Running times for DOM full test including examination of attributes and substring test on contents of text and attribute nodes. Average times is reported.

SDOM is built upon succinct data structures. Although there has been a great deal of interest in the algorithms community in the theory of succinct data structures, and implementations of full-text indices that are based upon succinct data structures (see e.g. [9]), these appear to be relatively unknown to the database community. We believe that the data structures we use could also be applied to other XML compressors.

There are a number of tasks and open questions that remain. Firstly, SDOM, as described can only be used for static documents. Dynamizing succinct data structures is an area of active research (see e.g. [19]), but it is far from clear how to implement a full DOM with dynamic operations. Secondly, although loading an XML document is fast (it needs to be – our traversal tests take so little time that reading in the XML file would otherwise be a serious bottleneck in our experiments) and does not take anywhere near the amount of memory required by a standard DOM parser, we have not made a serious attempt at optimizing either the speed or the memory usage of parsing. Finally, in addition to the tests that we have performed, it would be very interesting to wrap SDOM in an application such as Xalan, and investigate its performance therein.

Acknowledgement. We thank Michael Kay for helpful comments. Delpratt was supported by PPARC e-Science studentship PPA/S/E/2003/03749.

7. REFERENCES

[1] Arion, A., Bonifati, A., Manolescu, I., and Pugliese, A. 2007. XQueC: A query-conscious compressed XML database. *ACM Trans. Inter. Tech.* **7**, Article 10. doi:10.1145/1239971.1239974

[2] Boncz, P., Grust, T., van Keulen, M., Manegold, S.,Rittinger, J., and Teubner, J. 2006. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In Proc. 2006 ACM SIGMOD international Conference on Management of Data, pp. 479-490.doi:10.1145/1142473.1142527

[3] BZip2. http://www.bzip.org.

[4] Busatto, G., Lohrey, M., and Maneth, S. 2005. Efficient memory representation of XML documents. In *Proc. Database Programming Languages, 10th International Symposium*, *DBPL 2005*. Springer LNCS 3774, pp. 199-216. doi:10.1007/11601524_13

[5] Cheng, J., Ng, W., 2004. XQzip: Querying compressed XML using structural indexing. In *Proc. 9th International Conference on Extending Database Technology*, *EDBT '04*. Springer LNCS 2992, pp. 219-236. doi:10.1007/b95855.

[6] Clark, D. R. and Munro, J. I. 1996. Efficient suffix trees on secondary storage. In *Proc. Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '96*, pp. 383-391.

[7] Delpratt, D., Rahman, N., and Raman, R. 2007. Compressed prefix sums. In *Proc. Theory and Practice of Computer Science, SOFSEM '07*. Springer LNCS 4362, pp. 235-247. doi:10.1007/978-3-540-69507-3_19.

[8] Delpratt, D., Rahman, N., and Raman, R. 2006. Engineering the LOUDS succinct tree representation. In *Proc.5th Workshop on Experimental Algorithms, WEA '06*. Springer LNCS 4007, pp. 134-145. doi:10.1007/11764298_12.

[9] Ferragina, P. and Manzini, G. 2001. An experimental study of an opportunistic index. In *Proc. Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms* 269-278.

[10] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S. 2006. Compressing and searching XML data via two zips. In *Proc. 15th International World Wide Web Conference, WWW '06*. ACM Press, pp. 751-760. doi:10.1145/1135777.1135891

[11] Geary, R. F., Rahman, N., Raman, R., and Raman, V. 2006. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, **368**, pp. 231-246. doi:10.1016/j.tcs.2006.09.014

[12] Jacobson, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th Symp. Foundations of Computer Science, FOCS '89*, pp. 549-554. doi:10.1109/SFCS.1989.63533

[13] Kim, D. K., Na, J. C., Kim, J. E., and Park, K. 2005. Efficient implementation of rank and select functions for succinct representation. In *Proc. 4th Workshop on Experimental Algorithms, WEA 2005*. Springer LNCS 3503, pp. 315-327. doi:10.1007/11427186_28.

[14] Liefke, H. and Suciu, D. 2000. XMill: an efficient compressor for XML data. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data*. ACM Press, pp. 153-164. doi:10.1145/342009.335405

[15] Min, J., Park, M., and Chung, C. 2003. XPRESS: A queriable compression for XML data. In *Proc. 2003 ACM SIGMOD Intl. Conference on Management of Data, SIGMOD '03*. ACM Press, pp. 122-133. doi:10.1145/872757.872775

[16] Munro, J. I. and Raman, V. 2002. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM J. Comput*. **31**, pp. 762-776. doi:S0097539799364092

[17] Neumüller, M. and Wilson, J. N. 2003. Improving XML processing using adapted data structures. In *Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, Revised Papers*, Springer LNCS 2593, pp. 206-220.

[18] Ng, W., Lam, W., Wood, P. T., and Levene, M. 2006. XCQ: A queriable XML compression system. *Knowl. Inf. Syst.* **10**, pp. 421-452. doi:S10115-006-0012-z

[19] Raman, R. and Rao, S. S. 2003. Succinct dynamic dictionaries and trees. In *Proc. ICALP 2003*, Springer LNCS 2719, pp. 357—368.

[20] Saxon. http://saxon.source.forge.net/

[21] Tolani, P., Haritsa, J.R. 2002. XGRIND: A query-friendly XML compressor. In *Proc. 18th International Conference on Data Engineering, ICDE 2002,* IEEE Computer Society, pp. 225-234. doi:10.1109/ICDE.2002.994712

[22] Xerces C++ Parser. http://xerces.apache.org/xerces-c/

[23] Xerces Java 2 Parser. http://xerces.apache.org/xerces2-j/

[24] UW XML Repository. http://www.cs.washingon.edu/research/xmldatasets/www/repository.html

[25] VOTable Documentation. http://www.us-vo.org/VOTable/

[26] W3C DOM API documentation. 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/

[27] W3C DOM Traversal, 2000. http://www.w3.org/TR/DOM-Level-2-Traversal-Range/traversal.html

[28] Zhang, N., Kacholia, V., and Özsu, M. T. 2004. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proc. 20th Intl. Conference on Data Engineering*, pp. 54-65. doi:10.1109/ICDE.2004.1319984

Table 2 - Statistics of XML documents used in our experiments (NEG = negligibly small).

File	Nodes	#Elem	#Attr	#text	#Entity Ref	Entity	#Leaf	Max. Depth	Max node degree	#text chars	#attr Chars
Orders	300,004	150001	1	150001	0	0	150001	3	30001	1MB	NEG
Lineitem	2,045,954	1022976	1	1022976	0	0	1022976	3	120351	6MB	0
XPATH	2,522,571	840857	0	1681713	0	0	1681713	5	42075	13MB	NEG
Treebank_e	7,312,613	2437666	1	4874945	0	0	4875331	36	112769	57MB	NEG
SwissProt	10,599,084	2977031	2189859	5432193	0	0	5954060	5	100000	35MB	13MB
DBLP	10,595,379	3332130	404276	6792148	66756	67	6792149	6	657717	64MB	7MB
XCDNA	25,221,153	8407051	0	16814101	0	0	16814101	7	82237	256MB	0

Table 3 - Space usage of XML representations.

		Uncompressed repn's			Queriable compressed representations					
File	**Size**	**SDOM**	**Xerces-C**	**Saxon**	**SDOM-CT**	**XBZipIndex**	**XPRESS**	**XQZip**	**XGRIND**	**XMILL**
Orders	5MB	37%	451%	157%	17%	-	-	-	-	12%
Lineitem	32MB	28%	399%	161%	13%	-	-	5%	24%	5%
XPath	50MB	33%	383%	137%	10%	-	-	-	-	3%
Treebank_e	82MB	84%	866%	266%	43%	54%	-	43%	52%	30%
SwissProt	110MB	60%	704%	272%	22%	8%	38%	38%	43%	7%
DBLP	128MB	68%	737%	240%	24%	14%	48%	30%	43%	15%
XCDNA	594MB	50%	491%	136%	14%	-	-	-	-	8%

Table 4 – Running times for Xerces and SDOM for 'upward path enumeration', plus relevant file parameters.

File	#Nodes	%Non-leaf nodes	Max. Depth	Xerces	SDOM	Slowdown
Orders	300003	50%	3	0.08	0.13	1.64
Lineitem	2045954	50%	3	0.55	1.28	2.33
XPATH	2522571	33%	5	0.80	2.52	3.16
Treebank_e	7312613	33%	36	3.22	9.84	3.05
SwissProt	10599084	55%	5	2.71	8.76	3.23
DBLP	10595379	37%	6	2.97	7.90	2.66
XCDNA	25221153	33%	7	24.50	30.72	1.25

Table 5 - Full test using TreeWalker. Shows running times in seconds for Xerces using tree navigation operations, and using NextNode, versus SDOM using tree navigation and NextNode and SDOM-CT using tree navigation. Times in seconds.

File	#Nodes	Xerces TreeNav	Xerces NextNode	SDOM TreeNav	SDOM NextNode	SDOM-CT TreeNav
Orders	300003	0.06	0.05	0.09	0.06	0.22
Lineitem	2045954	0.37	0.32	0.64	0.39	1.22
XPATH	2522571	0.44	0.40	0.82	0.51	1.68
Treebank_e	7312613	1.40	1.25	2.85	1.57	8.51
SwissProt	10599084	1.70	1.48	3.28	2.30	7.78
DBLP	10595379	1.86	1.67	3.56	2.28	10.45
XCDNA	25221153	17.63	16.88	8.50	5.42	27.90