Managing Compressed Structured Text

Nieves R. Brisaboa[1], Ana Cerdeira-Pena[1], Gonzalo Navarro[2]

[1] Database Laboratory, Department of Computer Science, University of A Coruña, A Coruña, Spain

[2] Department of Computer Science, University of Chile, Santiago, Chile

SYNONYMS

Compressing XML; Searching Compressed XML

DEFINITION

Compressing structured text is the problem of creating a reduced-space representation from which the original data can be re-created exactly. Compared to plain text compression, the goal is to take advantage of the structural properties of the data. A more ambitious goal is that of being able of manipulating this text in compressed form, without decompressing it. This entry focuses on compressing, navigating, and searching structured text, as those are the areas where more advances have been made.

HISTORICAL BACKGROUND

Modeling data using structured text has been a topic of interest at least since the 1980s, with a significant burst of activity in the 1990s [3]. Since then, the widespread adoption of XML (appearing in 1998, see the current version at `http://www.w3.org/TR/xml`) as the standard to represent structured text has unified the efforts of the community around this particular format. Very early, however, the same features that made XML particularly appealing for both human and machine processing were pointed out as significant sources of redundancy and wasting of storage space and bandwidth. This was especially relevant for wireless transmission and triggered the proposal of the WAP Binary XML Content Format (WBXML) as early as 1999 (see `http://www.w3.org/TR/wbxml`), where simple techniques to compress XML prior to its transmission were devised.

In parallel, there has been a growing interest in not only compressing the data for storage or transmission, but in manipulating it in compressed form. The reason is the widening gaps in the memory hierarchy. A more compact data representation has the potential of fitting in a faster memory, where manipulating it can be orders of magnitudes faster, even if it requires more operations, than a naive representation fitting only in a slower memory. Moreover, reducing the space usage may be key to meet the requirements of memory-limited devices (such as mobile devices). Regarding distributed scenarios, compact data representations are particularly appealing to minimize the number of machines used, their energy consumption, and the overall communication costs.

SCIENTIFIC FUNDAMENTALS

For concreteness, the entry will focus on the de facto standard XML, where the structure is a tree or a forest marked with beginning and ending tags in the text. There can be free text between every consecutive pair of tags. In fact this encompasses many other structured text proposals, hence most of the material of the entry applies to structured text in general, with minimal changes. In XML, the tags can have attributes and associated values, and there might be available a grammar giving the permissible context-free syntax of the structured document (called the DTD).

Compression of structured text

An obvious approach to compressing structured text is to regard it as plain text and use any of the well-known text compression methods, leading to so-called *XML blind* compressors. Yet, considering the structure might yield improved compression performance ratios. Many *XML conscious compressors* have been proposed to exploit structure in different ways. Rather than fully describing each tool individually, the main principles behind them will be presented, and then illustrated with a few examples.

- The structure can be regarded as a labeled tree, where the labels are the tag and attribute names (which we call collectively tags), and the content as free text. Attribute information can be handled as text as well, or as special data attached to tree nodes.
- The structure and the content can be compressed separately, which has proved to give good results.
  - The structure can be compressed in several ways, which can range from a simple scheme of assigning numbers to the different tag names, to sophisticated tree grammar compression methods. The latter may take advantage of the DTD when it is available.
  - The text content can be compressed using any text compression method. Semi-static compressors permit accessing the content at random without decompressing all from the beginning, whereas adaptive compressors may achieve better compression ratios when the text data is heterogeneous. Splitting the text into blocks that are compressed adaptively yields trade-offs.
  - Structure can be used, in addition, to boost text compression. If the text contents are grouped according to the structural path towards the root, and each group is compressed separately, compression ratios improve noticeably. This can be as simple as grouping texts that are under the same tag (i.e., considering only the deepest tree node containing the text) or as sophisticated as considering the full path towards the root. Texts can also be separated by data type (e.g., dates, numbers, etc.)
- Even if encoding tags and contents separately, they can be stored in the file in their original order, so that the document can be handled as a plain uncompressed document. These are *homomorphic* compressors. Alternatively, *non-homomorphic* compressors store structure and content separately, with some pointer information to reconstruct the tree. In this case, the structure pointers may help to point out relevant content to scan in the querying process.

Most tools that compress XML run on diverse combinations of these principles. Some of the most prominent examples follow:

- *XMill* was the first compressor separating structure and content. It uses dictionary compression for the tags, while text content and attribute values are grouped based on the rooted data path and their data type, and then compressed independently. The sequences are finally passed to a back-end general text compressor.
- *Millau* is another early XML compressor that generates separate streams for structure and content. The structural one is encoded simply using WBXML, but it is optimized by taking the DTD as the base grammar. The content stream is compressed with a general text compressor.
- *XMLPPM* encodes the tokens and passes them to one of four different PPM models, depending on their syntactic meaning. To exploit correlations between different syntactic classes, *XMLPPM* injects previous symbols into the multiplexed models to be used as context.
- *XGrind* is a homomorphic compressor. It compresses the tags using dictionary encoding, and uses different Huffman coders for the data content associated to each tag and attribute name.
- *XPress* is also homomorphic and applies different compression schemes based on token types, but tags are encoded regarding their full path to the root. Paths are mapped to intervals of real numbers so that the interval of a suffix of a path contains the interval of a path.
- *AXECHOP* produces a context-free grammar of the document structure (after tokenizing it). The grammar is then compressed with an adaptive arithmetic coder. Text containers are separately created according to the tag enclosing the texts, and compressed with *bzip2*.
- *LZCS* is aimed at trees with repetitive topology. It converts the tree into a directed acyclic graph, by factoring common subtrees.

- *TinyT* uses a more powerful tree grammar compression, more specifically TreeRepair [12], which leads to very small representations on repetitive topologies.
- *XBzip* does not explicitly separate structure from content, but its construction automatically leads to a division based on tree paths. The compressor is based on the XBW transform [8], which succinctly represents labeled trees.

Other well-known XML compressors are *XCQ*, *SCM*, *XQueC*, *SCA*, *XComp*, *RNGzip*, *XWRT*, *QXT*, *XQZip*, *XSeq*, and *TREECHOP*. The appropriate references, as well as a more exhaustive coverage, can be found in recent surveys [17, 7].

Navigating and searching in compressed form

The most popular retrieval operations on structured text are related to *navigating* the tree and to *searching* it. Compressors providing some kind of support of these operations are usually referred as *queriable* methods, in contrast with the *non-queriable* ones, that just aim to reduce the amount of space used. Navigating means moving from a node to its children, parent, and siblings. Searching means various *path matching* operations such as, for example, finding all the paths where a node labeled A is the parent of another labeled B and that one is the ancestor of another labeled C, which in turn contains text where word W appears. A popular language combining navigation and searching operations is XPath (see `http://www.w3.org/TR/xpath20`).

Several of the schemes above permit accessing and decompressing any part of the text at random positions. This is because they retain the original order of the components of the document and compress using a semi-static model (e.g., *XGrind* and *XPress*). Those compression methods are transparent, in the sense that the classical techniques to navigate and search XML data, sequentially or using indexes, can be used almost directly over this compressed representation. Other techniques, such as *XCQ*, allow random access under a slightly more complex scheme, because some work is needed in order to start decompression at a specific point. Finally, techniques based on adaptive compression (such as *XMLPPM*) usually achieve better compression ratios, but need to decompress the whole data before they can operate on it. These are considered non-queriable representations. Other non-querieable compressors are *XMill*, *AXECHOP*, *XComp*, and *XWRT*.

Among the *queriable* solutions, some techniques take advantage of the separation between structure and content in order to run queries faster than scanning all the data. This is the case of *XCQ*, where the table that points from each different tree path to all the contents compressed under the corresponding model, is useful to avoid traversing those contents if the path does not match a path matching query. *XQueC* and *QXT* also work on a similar basic idea, but from different points of view. While *XQueC* focuses on query speed and query extent rather than compression efficiency, by creating several auxiliary data structures and indices, *QXT* aims at effective compression and does not keep any index, thus it offers a more limited query support. Another example is *XPress*, which encodes paths in a way that the codes themselves permit checking containment between two paths, and encodes numerical values in a way that allows directly performing range queries in compressed form.

Tree grammars may also support tree traversals without decompression. For instance, *LZCS* is aimed at compressing highly structured data, by replacing identical subtrees by a pointer to their first occurrence. The structure can be navigated almost transparently, and path matching operations can be sped up by factoring out the work done on repeated substructures. *TinyT* uses a more general tree grammar compressor, and it efficiently handles navigation and some structural queries. Many such queries can be solved by running a *tree automaton* on the tree [11].

The de facto XPath query language, however, requires much more than handling just traversals and a few path queries. In the rest of this entry, we will describe two recent approaches to support XPath functionality over succinct representations of XML data.

Succinct encodings for labeled trees

Succinct representations of labeled trees are an algorithmic development that finds applications in navigating structured text in compressed form. In its simplest form, a general labeled tree of n nodes can be represented using a sequence P of $2n$ balanced parentheses and a sequence L of n labels (which correspond to tag names and will be regarded as atomic for simplicity). This is obtained by traversing the tree in preorder (i.e., first the current node and then recursively each of its children). As the tree is traversed, an opening parenthesis is added to P each time one goes down to a child, and a closing parenthesis when going up back to the parent. That is

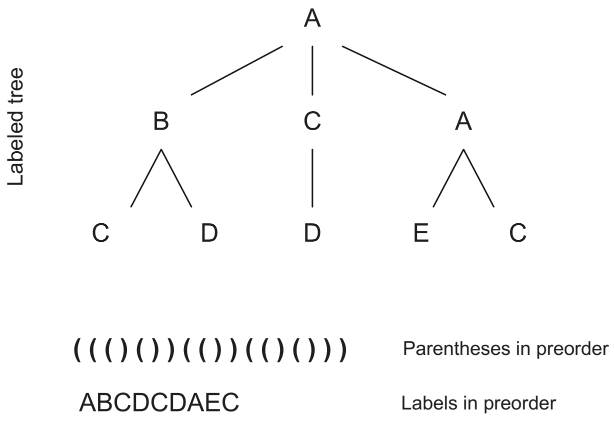


Figure 1: Succinct representation of a labeled tree, based on parentheses and labels sequences.

called the *balanced parentheses* representation of a tree. In L, the labels are added in preorder.

Figure 1 shows an example representation of a labeled tree as a sequence of parentheses and labels in preorder. It is not hard to rebuild the tree from this representation. However, what is really challenging is to navigate the tree directly in this representation (where a node is represented by the position of its opening parenthesis).

An essential operation to achieve efficient navigation in compressed form is the *rank* operation on bitmaps: $\mathit{rank}(P, i)$ is the number of 1's (here representing opening parentheses) in $P[1, i]$. One immediate application of *rank* is to obtain the label of a given node i, as $L[\mathit{rank}(P, i)]$. For example, consider the second child of the root in Figure 1. It is represented by the opening parenthesis at position 8 in the sequence. Its label is therefore $L[\mathit{rank}(\text{“((()())(())(()()))”}, 8)] = L[5] = \text{“}C\text{”}$. Another application of *rank* is to compute the depth of a node i. This is the number of opening minus closing parentheses in $P[1, i]$, that is, $\mathit{rank}(P, i) - (i - \mathit{rank}(P, i)) = 2 \cdot \mathit{rank}(P, i) - i$. For example, the depth of the second child of the root is $2 \cdot \mathit{rank}(\text{“((()())(())(()()))”}, 8) - 8 = 2 \cdot 5 - 8 = 2$. The dual of operation *rank* is $\mathit{select}(P, i)$, which gives the position of the i-th 1 in P. This yields the tree node with preorder i, or the tree node corresponding to label $L[i]$.

A large number of tree traversal and query operations can be supported with *rank*, *select*, and a few extra primitives: Operation $\mathit{close}(i)$ gives the position of the parenthesis that closes i (i.e., the next parenthesis with the same depth of i). Operation $\mathit{enclose}(i)$ gives the lowest parenthesis that contains i (i.e., the preceding parenthesis with depth smaller than that of i). All those operations can be solved efficiently with little extra space on top of P [1].

With these two operations one can navigate the tree as follows. The next sibling of i is $\mathit{close}(i)+1$ (unless it is a ')', in which case i is the last child of its parent). The first child of i is $i+1$ unless $P[i+1]$ is a ')', in which case i is a leaf and hence has no children. The parent of i is $\mathit{enclose}(i)$. The size of the subtree rooted at i is $(\mathit{close}(i) - i + 1)/2$. For example, consider the first child of the root in Figure 1, such that $i = 2$. It finishes at $\mathit{close}(i) = 7$. Its next sibling is $\mathit{close}(i) + 1 = 8$, the node of the previous examples. Its first child is $i + 1 = 3$, the leftmost tree leaf. Its parent is $\mathit{enclose}(i) = 1$, the root. The size of its subtree is $(\mathit{close}(i) - i + 1)/2 = (7 - 2 + 1)/2 = 3$.

In order to enrich the navigation using the labels, sequence $L[1, n]$ may also be processed for symbol *rank* and *select* operations, where $\mathit{rank}_c(L, i)$ is the number of occurrences of c in $L[1, i]$ and $\mathit{select}_c(L, j)$ is the position of the j-th occurrence of c in L. Different sequence representations supporting this functionality exist. We refer the reader to a recent practical development [4].

For example, the following procedure finds all the descendants of node i which are labeled c: (i) Find the position $j = \mathit{rank}(P, i)$ of node i in the sequence of labels. (ii) Compute $k = \mathit{rank}_c(L, j - 1)$, the number of occurrences of c prior to j. (iii) Find the positions $p_r = \mathit{select}_c(L, k + r)$ of c from j onwards, for successive r values until $\mathit{select}(P, p_r) > \mathit{close}(i)$, that is, until the answers are not anymore descendants of i. For example, consider again the first child of the root in Figure 1, where $i = 2$ and $\mathit{close}(i) = 7$, and find its descendants labeled “D”. The first step is to compute $j = \mathit{rank}(P, 2) = 2$, the position of its label in L. Now, $k = \mathit{rank}_{\text{“}D\text{”}}(L, 1) = 0$ tells that there are zero occurrences of “D” before $L[2]$. Now the next occurrences of “D” in L are found as $\mathit{select}_{\text{“}D\text{”}}(L, 1) = 4$,

$select_{\text{"}D\text{"}}(L,2) = 6$, ... The first such occurrence is mapped to the tree node $select(P,4) = 5$ (the second tree leaf), which is within the subtree of i because $i \leq 5 \leq close(i)$. The second occurrence of "D" is already outside the tree because $select(P,6) = 9$ exceeds $close(i) = 7$.

The XPath language integrates path matching and content queries. *SXSI* [2] is a recent system that integrates the encoding of labeled trees just described with a compressed *self-indexed* representation of the collection of text nodes in the XML document. A self-index [14] is a representation of a text that not only allows accessing arbitrary text passages, but also supports efficient substring searches on it. SXSI addresses a relevant subset of XPath by combining queries on the parentheses sequence, *rank/select* operations on the sequence of labels, and pattern matching queries on the text. Queries are solved by using a tree automaton that traverses the XML structure using the described representations to direct the navigation towards just the relevant parts of the structure.

Different realizations of SXSI fit different scenarios. In the original proposal [2] they aim at speed, thus they use the so-called "fully-functional" parentheses representation [1], an FM-index [9] for the text contents, and a plain representation of the tag sequence L for fast access, plus one bitmap B_c for each tag c, with $B_c[i] = 1$ iff $L[i] = c$, thus $rank_c(L,i) = rank(B_c,i)$ and $select_c(L,i) = select(B_c,i)$ are solved efficiently. The resulting index use nearly the same space of the original XML data and its query efficiency is competitive with non-compressed representations, such as that of *MonetDB*.

Another realization [16] aims at greatly reducing space on repetitive XML collections (e.g., versioned collections of XML data), where grammar compression stands out. Tree-grammar-based representations for the structure are not powerful enough to support the XPath operations. Instead, they use a grammar-compressed representation of the parentheses sequence, enriched with data that supports the described navigation operations [15]. This is complemented with a new grammar-compressed representation for L that supports *rank/select* operations. Finally, a self-index aimed at repetitive text collections, called *run-length compressed suffix array* [13], is used. On real-life highly repetitive collections (e.g., versioned software repositories), the final representation is much smaller than the original data (e.g., 25%, dominated by the text component). In exchange, XPath queries are noticeably slower.

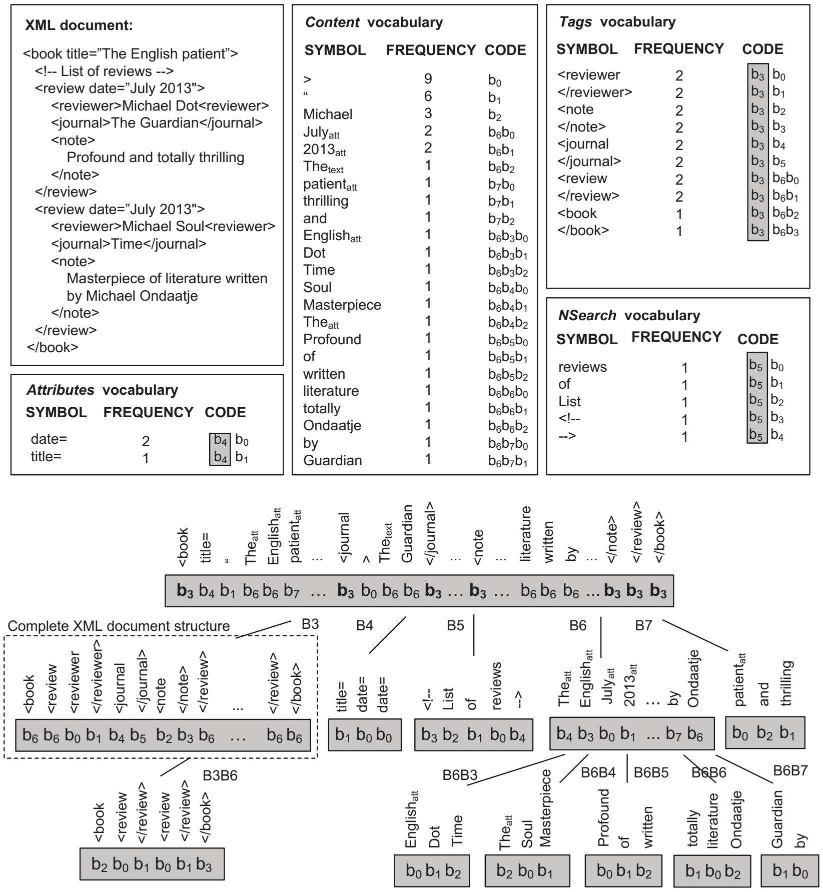
Integrating indexing and compression

We have mentioned self-indexing as the concept of representing text data in compressed form so that the representation itself is querieable, for pattern-matching queries in that case. This concept can be translated into structured text compression, so that the representation of the XML data is itself an index that supports fast XPath queries. The first representation that built on this idea [8] was the basis of *XBzip*, but its query support is limited to very simple path matching queries. In this final section we describe a more recent development, *XXS* [5], which supports a large subset of XPath, yet it is limited to XML collections containing natural language. *XXS* represents the XML collections in 35%–50% of their original space, and is competitive in time efficiency with *SXSI* and *MonetDB*, which use much more space.

XXS represents the XML collection using a data structure called the XML Wavelet Tree (XWT). The XWT representation of a document separates the tokens into four categories: (*i*) start/end tags, (*ii*) attribute names, (*iii*) comments and processing instructions, and (*iv*) text content and attribute values. The words of each vocabulary are statistically encoded using a byte-wise representation called (s,c)-DC [6], which is almost as good as byte-wise Huffman codes but more flexible. This encoding is tailored to make XWT suitable for querying: the codewords of the vocabularies (*i*), (*ii*) and (*iii*), called *special*, are forced to start with specific byte values.

Instead of writing down the codes one after the other, the bytes of all the codewords are reordered and arranged into the XWT nodes. The root of the tree (i.e., the first level) contains the first bytes of the codewords, in the same order as in the XML collection. Then, each node Bx in the second level stores the second bytes of the codewords whose first byte is b_x (preserving again the text order). That is, the second byte corresponding to the j-th occurrence of byte b_x in the root is located at position j in node Bx, and so on. Operations *rank/select* on the byte sequences are sufficient to efficiently access and query the XWT.

Figure 2 shows an example XWT built from an XML document. Note that the combination of the XWT arrangement, combined with the usage of specific first bytes to encode the special vocabularies, isolates those special words below some XWT nodes. This yields several benefits. For instance, attribute isolation provides direct access to this type of tokens during query evaluation, while keeping comments and processing instructions separated allows skipping those fragments in general text searches. Yet, even more important is the implicit structural isolation. Note that the subtree below node $B3$ in Figure 2 stores only start/end-tags, and that they



```
<book title="The English patient">
  <!-- List of reviews -->
  <review date="July 2013">
     <reviewer>Michael Dot<reviewer>
     <journal>The Guardian</journal>
     <note>
        Profound and totally thrilling
     </note>
  </review>
  <review date="July 2013">
     <reviewer>Michael Soul<reviewer>
     <journal>Time</journal>
     <note>
        Masterpiece of literature written
        by Michael Ondaatje
     </note>
  </review>
</book>
```

Attributes vocabulary

SYMBOL	FREQUENCY	CODE
date=	2	b_4 b_0
title=	1	b_4 b_1

Content vocabulary

SYMBOL	FREQUENCY	CODE
>	9	b_0
“	6	b_1
Michael	3	b_2
July$_{att}$	2	b_6b_0
2013$_{att}$	2	b_6b_1
The$_{text}$	1	b_6b_2
patient$_{att}$	1	b_7b_0
thrilling	1	b_7b_1
and	1	b_7b_2
English$_{att}$	1	$b_6b_3b_0$
Dot	1	$b_6b_3b_1$
Time	1	$b_6b_3b_2$
Soul	1	$b_6b_4b_0$
Masterpiece	1	$b_6b_4b_1$
The$_{att}$	1	$b_6b_4b_2$
Profound	1	$b_6b_5b_0$
of	1	$b_6b_5b_1$
written	1	$b_6b_5b_2$
literature	1	$b_6b_6b_0$
totally	1	$b_6b_6b_1$
Ondaatje	1	$b_6b_6b_2$
by	1	$b_6b_7b_0$
Guardian	1	$b_6b_7b_1$

Tags vocabulary

SYMBOL	FREQUENCY	CODE
<reviewer	2	b_3 b_0
</reviewer>	2	b_3 b_1
<note	2	b_3 b_2
</note>	2	b_3 b_3
<journal	2	b_3 b_4
</journal>	2	b_3 b_5
<review	2	b_3 b_6b_0
</review>	2	b_3 b_6b_1
<book	1	b_3 b_6b_2
</book>	1	b_3 b_6b_3

NSearch vocabulary

SYMBOL	FREQUENCY	CODE
reviews	1	b_5 b_0
of	1	b_5 b_1
List	1	b_5 b_2
<!--	1	b_5 b_3
-->	1	b_5 b_4

<book title= “ The$_{att}$ English$_{att}$ patient$_{att}$ … <journal > The$_{text}$ Guardian </journal> … <note … literature written by … </note> </review> </book>
b_3 b_4 b_1 b_6 b_6 b_7 … b_3 b_0 b_6 b_6 b_3 … b_3 … b_6 b_6 b_6 … b_3 b_3 b_3
Complete XML document structure
B3
<book <review <reviewer </reviewer> <journal </journal> <note </note> </review> … </review> </book>
b_6 b_6 b_0 b_1 b_4 b_5 b_2 b_3 b_6 … b_6 b_6
B4
title= date= date=
b_1 b_0 b_0
B5
<!-- List of reviews -->
b_3 b_2 b_1 b_0 b_4
B6
The$_{att}$ English$_{att}$ July$_{att}$ 2013$_{att}$ … by Ondaatje
b_4 b_3 b_0 b_1 … b_7 b_6
B7
patient$_{att}$ and thrilling
b_0 b_2 b_1
B3B6
<book <review </review> <review </review> </book>
b_2 b_0 b_1 b_0 b_1 b_3
B6B3
English$_{att}$ Dot Time
b_0 b_1 b_2
B6B4
The$_{att}$ Soul Masterpiece
b_2 b_0 b_1
B6B5
Profound of written
b_0 b_1 b_2
B6B6
totally literature Ondaatje
b_1 b_0 b_2
B6B7
Guardian by
b_1 b_0

Figure 2: Example of a XWT structure built from an XML document.

follow the document order. Hence, the root of this subtree (i.e., node $B3$) actually matches a balanced parentheses representation of the XML document structure. Both structures are used in conjunction by *XXS* to solve queries. Instead of tree automata, *XXS* uses the so-called *bottom-up* query evaluation [10], where the leaves of the query syntax tree are solved first and they feed the data to the higher nodes. The evaluation process is built on the following principles: (*i*) map subtrees to segments in a line to facilitate structural comparisons, (*ii*) use *lazy* evaluation to produce only the necessary data, and (*iii*) a *skipping* strategy that propagates restrictions from the top query nodes to the bottom nodes, thus avoiding processing unnecessary parts of the tree. The XWT representation is used to implement these tasks efficiently.

KEY APPLICATIONS

Any application managing structured text, particularly if it has to transmit it over slow channels or operate within limited fast memory, even if there is an unlimited supply of slower memory, benefits from these techniques.

FUTURE DIRECTIONS

The future of the area is likely to be in manipulating XML in compressed form, and in this aspect much more than XPath is needed. Note that the compressed representations we have considered are static. Languages like XQuery (`www.w3.org/TR/xquery`), that query but also transform the XML data, are much more demanding than XPath, in particular because manipulating XML requires generating new data as the result of queries.

EXPERIMENTAL RESULTS

Experiments can be found in the cited papers. The most recent ones are those of *SXSI* [2] and *XXS* [5].

URL TO CODE

Several public XML compressors are available, for example *XMill* (`https://sourceforge.net/projects/xmill`), *XMLPPM* (`https://sourceforge.net/projects/xmlppm`), *LZCS* (`http://www.infor.uva.es/~jadiego/download.php`), *XGrind* (`https://sourceforge.net/projects/xgrind`), *XWRT* (`http://xwrt.sourceforge.net/`), *XBzipIndex* (`http://pages.di.unipi.it/ferragina/software.html`), *SXSI* (`http://fclaude.recoded.cl/archives/193`), and *XXS* (`http://vios.dc.fi.udc.es/xxs/`).

CROSS REFERENCE

Compression, Semi-structured Data, XML, XPath

RECOMMENDED READING

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97. SIAM Press, 2010.

[2] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyen, J. Sirén, and N. Välimäki. Fast in-memory XPath search using compressed indexes. In *Proc. 26th IEEE International Conference on Data Engineering (ICDE)*, pages 417–428, 2010.

[3] R. Baeza-Yates and G. Navarro. Integrating contents and structure in text retrieval. *ACM SIGMOD Record*, 25(1):67–79, 1996.

[4] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

[5] N. R. Brisaboa, A. Cerdeira-Pena, and G. Navarro. XXS: Efficient XPath evaluation on compressed XML documents. *ACM Transactions on Information Systems*, 32(3):13, 2014.

[6] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.

[7] A. Cerdeira-Pena. *Compressed Self-indexed XML Representation with Efficient XPath Evaluation.* PhD thesis, Department of Computer Science, University of A Coruña, 2013.

[8] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):4:1–4:33, 2009.

[9] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005.

[11] M. Lohrey, S. Maneth, and R. Mennicke. The complexity of tree automata and XPath on grammar-compressed trees. *Theoretical Computer Sciences*, 363(2):196–210, 2006.

[12] M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using RePair. *Information Systems*, 38(8):1150–1167, 2013.

[13] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[14] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.

[15] G. Navarro and A. Ordóñez. Faster compressed suffix trees for repetitive text collections. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, LNCS 8504, pages 424–435, 2014.

[16] G. Navarro and A. Ordóñez. Grammar compressed sequences with rank/select support. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, 2014. To appear.

[17] S. Sakr. XML compression techniques: A survey and comparison. *Journal of Computer and System Sciences*, 75(5):303–322, 2009.