A compressed self-indexed representation of XML documents *

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

[1] Database Lab., Univ. da Coruña, Spain. {brisaboa,acerdeira}@udc.es
[2] Dept. of Computer Science, Univ. of Chile. gnavarro@dcc.uchile.cl

Abstract. This paper presents a structure we call XML Wavelet Tree (XWT) to represent any XML document in a compressed and self-indexed form. Therefore, any query or procedure that could be performed over the original document can be performed more efficiently over the XWT representation because it is shorter and has some indexing properties. In fact, XWT permits to answer XPath queries more efficiently than using the uncompressed version of the documents. XWT is also competitive when comparing it with inverted indexes over the XML document (if both structures use the same space).

1 Introduction

XML[1] has long ago become the standard for representing semi-structured documents and W3C has defined the language XPath[2] for querying XML documents allowing constraints on both structure and content. Recently, several works have been devoted to the problem of modelling and querying XML documents and new query languages or XPath extensions have been proposed [10, 9, 3].

On the other hand, the research in text compression has experimented a big advance in the last years. Different compression methods have been proposed, demonstrating beyond doubts that the use of word-based statistical semi-static compressors, such as Plain and Tagged Hufman, ETDC, *(s,c)*-DC or RPBC [12, 6, 8], perfectly fulfil IR requirements because those compressors allow querying the compressed version of the text up to 8 times faster than the uncompressed version. That is, the text is compressed to about 30%-35% of its original size and can be kept in that compressed form all the time, because direct search of words and phrases can be performed over that compressed version. Therefore the text only need to be uncompressed to be shown to a human user, but any process, for IR or any other purpose, can be done over the compressed text. In this way, not only storage space is saved, but also time. Time is the critical factor in efficiency and processing a compressed version of a document saves time when we need to access to disk looking for a document, when it is transmitted through a network, or more importantly, when it is processed.

More recently, compression techniques have become even more sophisticated allowing not only a compressed representation of the text, but also self-indexed

* Funded in part by MEC grant TIN2006-15071-C03-03, for the Spanish group; and for the third author by Fondecyt grant 1-080019 (Chile)

representations using the same compressed space (about 35% of the original size). Those compressed and self-indexed representations of the text successfully compete with the classical inverted indexes, even if they use compression strategies. Among those compressed and self-indexed document representations, the Word Suffix Arrays [11, 7] and the Wavelet Trees [4] are some of the most powerful.

In this paper we present a modified wavelet tree, based on a *(s,c)*-DC compressor, to create a self-indexed and compressed version of XML documents. Our representation, which we call XML Wavelet Tree (XWT), uses only about 30%-40% of the space of a XML document and provides some self-indexing properties that can be successfully used in answering XPath queries.

Notice that any XML document can be represented, using our XWT, in a compressed and self-indexed form, therefore any processing or query that could be performed over the original XML document can also be performed over the XWT representation. Moreover, due to the fact that the XWT representation is smaller and has some indexing properties, any processing will be more efficient over the XWT representation than over the original uncompressed document.

2 Previous work

Among the different word-based byte-oriented semi-static statistical compression methods available in the state of the art, we use *(s,c)*-DC [5, 6] as basis of our representation because it provides flexibility to compress with different models the *tags* of a XML document and the rest of its words. On the other hand, among the self-indexing structures available we chose to work with the WT presented in [4] because it is the only one that could be adapted in order to represent, in a compact way, the structure of the document (that is, the XML *tags*) separated from the rest of the words.

2.1 *(s,c)*-Dense Code

(s,c)-Dense Code is a word-based semi-static statistical prefix-free encoder. In a first pass over the source text the different words and their frequencies are obtained (the *model*). Then, the vocabulary is sorted by frequency and a codeword is assigned to each word (shorter codewords to more frequent words). In a second pass, the compressor replaces each word by its codeword leading to a compressed representation of the text.

As other compressors, *(s,c)*-DC distinguishes between bytes[1] that do not end a codeword, called *continuers*, and bytes that only can appear as the last byte of a codeword, *stoppers*. In this case, where s is the number of *stoppers* and c indicates the number of *continuers* ($s + c = 256$), *stoppers* are the bytes between 0 and $s - 1$ and *continuers*, are those between s and $s + c - 1 = 255$. To minimize compression ratios, optimal values for s and c are computed for the specific word frequency distribution of the text [5]. Then given source symbols sorted by decreasing frequencies, the corresponding *(s,c)*-DC encoding process

[1] For simplicity, we focus on the byte oriented version

gives one-byte codewords to the words in positions from 0 to $s-1$. Words ranked from s to $s+sc-1$ are sequentially assigned two-byte codewords. The first byte of each codeword has a value in the range $[s, s+c-1]$, that is, a *continuer*. The second byte, the *stopper*, has a value in range $[0, s-1]$. Words from $s+sc$ to $s+sc+sc^2-1$ are assigned three byte codewords, and so on.

Example 1. The codes assigned to symbols $i \in 0 \ldots 15$ by a (2,3)-DC are as follows: $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2,0 \rangle$, $\langle 2,1 \rangle$, $\langle 3,0 \rangle$, $\langle 3,1 \rangle$, $\langle 4,0 \rangle$, $\langle 4,1 \rangle$, $\langle 2,2,0 \rangle$, $\langle 2,2,1 \rangle$, $\langle 2,3,0 \rangle$, $\langle 2,3,1 \rangle$, $\langle 2,4,0 \rangle$, $\langle 2,4,1 \rangle$, $\langle 3,2,0 \rangle$ and $\langle 3,2,1 \rangle$.

2.2 Byte-oriented Wavelet Tree (WT)

In [4] we presented a novel reorganization of the codewords bytes of a text compressed with any word-based byte-oriented semi-static statistical prefix-free compression technique. This reorganization, called *Wavelet Tree*, consists basically on placing the different bytes of each codeword at different WT nodes instead of sequentially concatenating them, as in a typical compressed text.

The root of the WT is represented by all the first bytes of the codewords, following the same order as the words they encode in the original text. That is, let assume we have the text words $\langle w_1, w_2 \ldots w_n \rangle$, whose codewords are $cw_1, cw_2 \ldots cw_n$, respectively, and let us denote the bytes of a codeword cw_i as $\langle c_i^1 ... c_i^m \rangle$ where m is the size of the codeword cw_i in bytes. Then the root is formed by the sequence of bytes $\langle c_1^1, c_2^1, c_3^1 ... c_n^1 \rangle$. At position i, we place the first byte of the codeword that encodes the i^{th} word in the source text, so notice that the root node has as many bytes as words has the text. We consider the root of the WT as the first level. Therefore, second bytes of the codewords longer than one byte are placed in nodes of a second level. The root has as many children as different bytes can be the first byte of a codeword of two or more bytes. That is, in a $(190, 66)$-DC encoding scheme, the root will have always 66 children, because there are 66 bytes that are *continuers*. Each node X in this second level contains all the second bytes of the codewords whose first byte is x, following again the same order of the source. That is, the second byte corresponding to the j^{th} occurrence of byte x in the root, is placed at position j in node X. Formally, let suppose there are t words coded by codewords $cw_{i_1} ... cw_{i_t}$ (longer than one byte) whose first byte is x. Then, the second bytes of those codewords, $\langle c_{i_1}^2, c_{i_2}^2, c_{i_3}^2 ... c_{i_t}^2 \rangle$, form the node X. The same idea is used to create the lower levels of the WT. Looking into the example, and supposing that there are d words whose first byte codewords is x and whose second one is y, then node XY is a child of node X and it stores the byte sequence $\langle c_{j_1}^3, c_{j_2}^3, c_{j_3}^3 ... c_{j_d}^3 \rangle$ given by all the third bytes of that codewords. Those bytes are again in the original text order. Therefore, the resulting WT has as many levels as bytes have the longest codewords.

In Fig. 1[2], a WT is built from the text `MAKE EVERYTHING AS SIMPLE AS POSSIBLE BUT NOT SIMPLER`. Once codewords are assigned to all the different

[2] Note that only the shaded byte sequences are stored in tree nodes; the text is shown only for clarity.

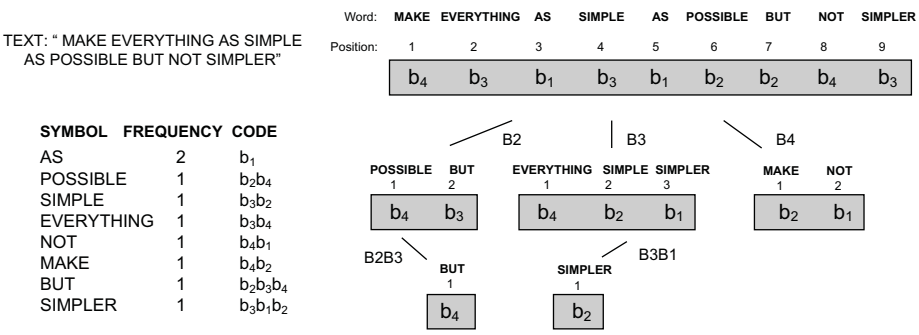

Fig. 1. Example of WT.

words in the text, their bytes are spread in a WT following the reorganization of bytes explained. For example, b_3 is the 9^{th} byte of the root because it is the first byte of the codeword assigned to `'SIMPLER'`, which is the 9^{th} word in the text. In turn, its second byte, b_1, is placed in the third position of the child node $B3$ because `'SIMPLER'` is the third word in the root having b_3 as first byte. Likewise, its third byte, b_2, is placed at the third level in the child node $B3B1$, since the first and second byte of the codeword are b_3 and b_1, respectively.

Original codewords can be rebuilt from the bytes spread along the different WT nodes using *rank* and *select* operations. Let be B a sequence of bytes, b_1, $b_2 \ldots b_n$. Then, *rank* and *select* are defined as:

- $rank_b(B,p) = i$ if the number of occurrences of the byte b from the beginning of B up to position p is i.
- $select_b(B,j) = p$ if the j^{th} occurrence of the byte b in the sequence B is at position p

The two basic procedures using the WT are *locating* a word in the text and *decoding* the word placed at certain position. Both are easily solved using *select* and *rank* operations, respectively.

To find the first occurrence of `'SIMPLER'`, we will start at the bottom of the tree and go up. As we can see in Fig. 1, the codeword of `'SIMPLER'` is $b_3b_1b_2$, therefore, we start at node $B3B1$, in the third level, and search for the first occurrence of the byte b_2 computing $select_{b_2}(B3B1, 1) = 1$. In this way, we obtain that the first position of that node ($B3B1$) corresponds to the first occurrence of `'SIMPLER'`. Now, we need to locate in node $B3$ the position of the first occurrence of byte b_1. Again, this is obtained by $select_{b_1}(B3, 1) = 3$, that newly indicates our codeword is the third one starting by b_3 in the root node. Finally, by calculating $select_{b_3}(root, 3) = 9$, we can answer that the first occurrence of `'SIMPLER'` is at 9^{th} position in the source text.

To decode a word we use rank operations. To know which is the 7^{th} word in the source text we start reading $root[7] = b_2$. According to the encoding scheme we know that the code is not complete, so we will have to read a second

byte in the second level of the WT, more precisely, in the node $B2$. To find out which position of that sequence we have to read, we use $rank_{b_2}(root, 7) = 2$. Therefore, $B2[2] = b_3$, gives us the second byte of the codeword. Again b_3 is not a *stopper*, so we need to continue the procedure. In the child node $B2B3$, that corresponding to the two first bytes of the codeword we have just read (b_2b_3), we have to read the byte which is at position $rank_{b_3}(B2, 2) = 1$. Finally, we obtain $B2B3[1] = b_4$, which marks the end of the searched codeword. As a result, we have the codeword $b_2b_3b_4$, that corresponding to `'BUT'`, which is precisely the 7^{th} word in the source text, as expected.

The performance of the WT depends on the implementation of the *rank* and *select* operations, because they are the base for any procedure over this structure. A detailed description of their implementation can be found in [4]. It is based on a structure of partial counters to avoid counting the number of occurrences of a searched byte from the beginning of a WT node. There is a tradeoff between space and time. If we use more partial counters, we need more space, but *rank* and *select* operations will be more efficient.

3 XML Wavelet Tree (XWT)

Phase I: Parsing the XML document and assigning codewords The first step to obtain the XWT is to parse the input XML document to create the vocabulary and compute the frequencies distribution (the *model*). The *parsing* process distinguishes different kind of words depending on whether a word is[3]: *i*) a name of a *start-tag* or an *end-tag*, *ii*) the name of an attribute, *iii*) an attribute value, *iv*) a word appeared inside a *comment*, *v*) a word appeared inside a *processing instruction*, or *vi*) a word of the XML document content.

That is, when compressing, a same word will be assigned different codewords depending on the category it belongs to. For example, if the word *book* appears as content (e.g. ... *the great book* ...), but also as an attribute value (e.g. *category= "book"*) and inside a comment (e.g. $\langle !-- \ldots$ *this book is* $\ldots --\rangle$) it will be stored as three different entries in the vocabulary, one for each different category.

Keeping this difference between same words according to its function in the XML document structure increases the vocabulary size, however it is translated into efficiency and flexibility when querying.

It is also in the *parsing* that some *normalization* operations take place (all according to [1]). For instance, *empty-element tags* are translated into its corresponding pair of *start-end tag* (e.g. $\langle tag\_name/\rangle \rightarrow \langle tag\_name\rangle\ \langle /tag\_name\rangle$) and redundant spaces and spaces inside tags are eliminated (e.g. $\langle tag\_name \quad \rangle \rightarrow \langle tag\_name\rangle$), etc.

While parsing the XML document, two vocabularies are created. One stores the different *start-* and *end-tags* and therefore the structure of the document. The other stores the rest of the words. We call them, *tag words* and *text words* vocabulary, respectively (see Fig. 2).

[3] Division implicitly given by the different kind of XPath queries[2].

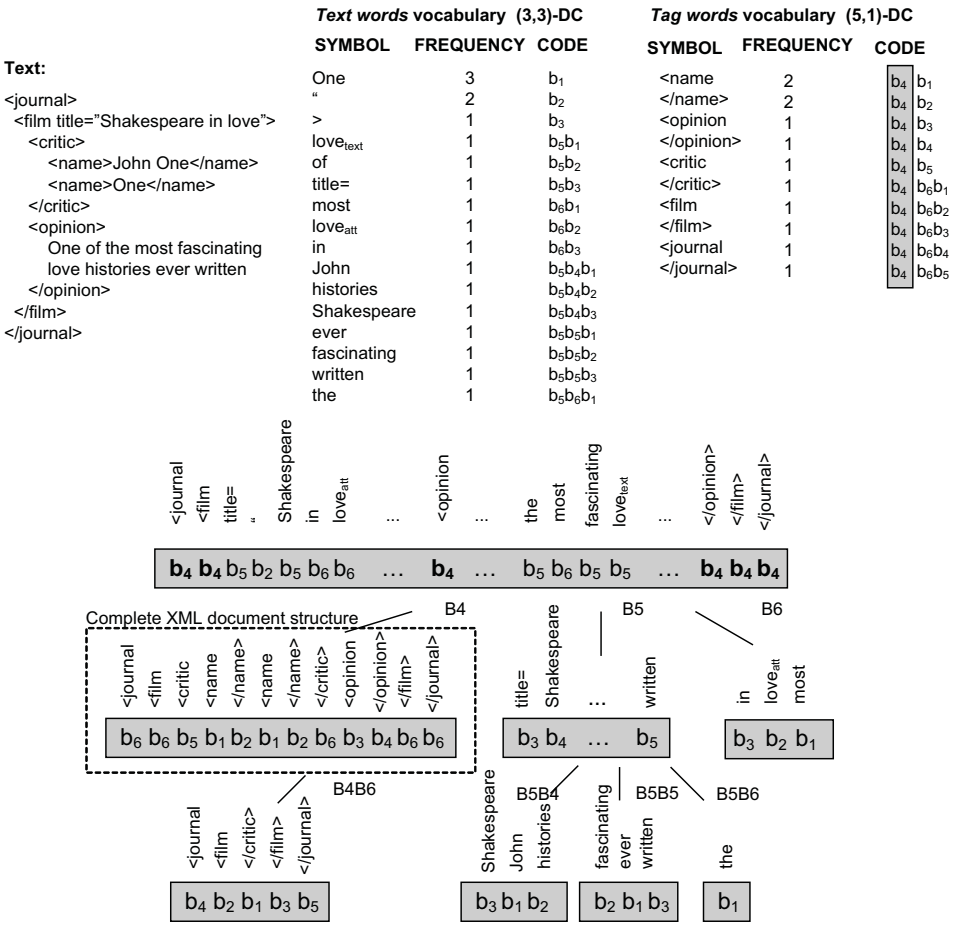

Fig. 2. Example of XWT.

As it was explained in Section 2, *(s,c)*-Dense Code uses different bytes for *continuers* and for *stoppers*. So it is easy to see how reserving a *continuer* to be the first byte of the codewords assigned to *tag words* (in Fig. 2, see the bytes shaded in the `CODE` column of the *tag words* vocabulary) it is possible to keep them all located in the same branch of the XWT (see the branch $B4$ in Fig. 2). Remember that they follow the document order and hence maintain their relationships like in the original XML document. But what is even more striking is that this feature implicitly provides an efficient way to solve structural queries. To do this, we only need to deal with those nodes of the XWT storing the structure of the document, and omit the rest of the compressed text.

Therefore, all the words of the *text words* vocabulary are assigned a codeword following a *(s,c)*-Dense Code encoding scheme, keeping aside one *continuer*. In

Fig. 2, where a $(3,3)$-DC encoding scheme is used to encode *text words*, the first of the *continuers*, b_4, has been selected as the one reserved. Notice that it is not used as a first byte of any of the codewords assigned to the *text words*.

Because of this, compression could be affected, so to minimize this loss, *tag words* are also coded according to another optimal values of s and c. That is, on the one hand we keep the selected *continuer* as the first byte of the *tag* codewords to store the XML structure isolated. On the other hand, the remaining bytes of the *tag* codewords will be given following their own *(s,c)*-DC scheme. In the example of the Fig. 2, codewords assigned to *tags* follow a $(5,1)$-DC encoding scheme, after the first byte, that is always the *continuer* b_4 (shaded column).

Phase II: Compressing and creating the XWT Once codewords are assigned to words, we do a second pass over the text replacing each word by its codeword and storing these codeword bytes along the different nodes of the XWT (it is possible to precalculate the number of nodes as well as their sizes in advance). So, by keeping an array of markers indicating the next writing position for each node, they are filled sequentially following the order of the words in the text.

4 Using XWT

4.1 Decompression

To decompress from a random text word j (*random decompression*), we follow the procedure explained in Section 2.2. But now, we take into account the use of two vocabularies with different encoding schemes. That is, we first access to the j^{th} byte of the root node of the XWT to get the first byte of the codeword and then we check if the byte read, b_i, matches or not with the *continuer* used to mark *tag words*. Depending on this, going down in the XWT to obtain the remaining bytes is done by using the corresponding s and c values.

If we want to decompress the whole text from the beginning (*full decompression*), we can follow a more efficient procedure. Given that the sequences of bytes of all the XWT nodes follow the original order of the words in the source text, *full decompression* can be efficiently implemented using pointers to the next positions to be read in each node. That is, when going to a child node to read the following byte of an uncomplete codeword, we do not need to compute any *rank* operation to find out what byte of this child node sequence we have to read. It always will be the next one to process in that child node.

4.2 Answering XPath queries

Since the XWT structure is an exact representation of the XML document, any operation over the original text can be done over such representation. Therefore, all XPath queries can be answered using our representation. Indeed, some of them take benefit of the implicit indexing properties provided by the own XWT structure and are efficiently answered.

Counting To *count()* the number of occurrences of a word (e.g. *tag*, name of an attribute, attribute value, word inside a comment or node content, etc.) we just compute how many times the last byte of the codeword assigned to that word appears in its corresponding XWT node. Therefore, if a word is encoded with a codeword xyz (being x and y, *continuers* and z, a *stopper*), it is only necessary to count the number of bytes z in the node XY. That is, we only do $rank_z(node_{XY}, i)$, where i is the size of the node XY. In turn, if the codeword has just one byte, z, we will do $rank_z(root,n)$, where n is the number of words in the text, that is, the number of bytes in the root.

Locating We can locate the position in the text of any occurrence of a word (typical XPath queries as *//book*, *//@title*, etc.) searching its last byte in the corresponding XWT node and performing consecutive *select* operations up to the root. If we want to locate all the occurrences of a word, this process is repeated for each one. Since the traversed XWT nodes are the same for each occurrence and these will be processed consecutively, select operations and thus the whole process, can be sped up by using pointers to the already found positions in the WT nodes.

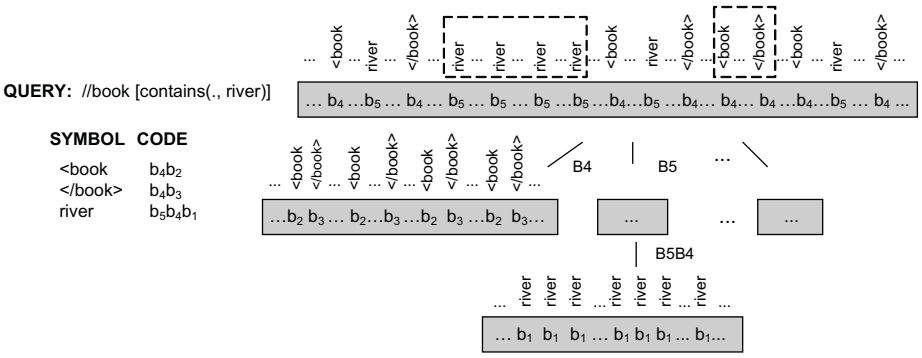

Fig. 3. Example of searching pairs of start-end tags containing a word.

Locating phrases To locate a *phrase* pattern we start locating the first occurrence of the least frequent word of the pattern in the root node. Then we check if all the first bytes of the codewords of each word of the phrase pattern match with the previous and next bytes of the root node. If those matches happen, we follow validating the rest of the bytes of the corresponding codewords. But if it is not the case, we save going down into the XWT and we simply locate the next occurrence of the least frequent word to be processed in a same way.

Searching pairs of start-end tags containing a word In XPath, a *predicate* is a filter applied to a set of XML nodes. For simplicity, here we have chosen

predicates over text: *//tag_name* [$contains(., word_{text})$]. That is, we are interested in reporting the pairs of start-end tags that fulfill $\langle tag\_name\rangle \ldots word_{text} \ldots \langle /tag\_name\rangle$. For example: *//book* [$contains(., river)$] (see Fig. 3).

We begin locating the first occurrence of the desired word (*river*, in the example) in the root node. Then, by counting the number of occurrences of the desired *start-tag* ($\langle book$ in Fig. 3) placed before that position and that of the desired *end-tag* ($\langle /book\rangle$) we will know how many of the element nodes we are looking for contain that occurrence of the word. We can easily figure out those number of occurrences dealing only with the branches of the XWT storing the *tags*. In the example, we locate the first occurrence of *river* which is surrounded by the first occurrences of $\langle book$ and $\langle /book\rangle$. Therefore they are reported as a hit.

Now, instead of performing the same process with the next occurrence of the word (in the example, the 2^{nd} occurrence of *river*), we can skip some text looking for the first occurrence of the desired *start-tag* placed after the position of the just located occurrence of the word. That is, in the example, we locate the second occurrence of $\langle book$ and its corresponding *end-tag*, and then we look for an occurrence of *river* between their positions. Given that there is one occurrence (the 6^{th} occurrence of *river*), the 2^{nd} occurrence of the element node *book* is also reported as a hit. By doing this we skip the occurrences of *river* that could be before the second occurrence of $\langle book$, and which are not interesting for the search (those occurrences of *river* surrounded by a striped rectangle in Fig. 3). After that, we proceed in one of the two ways. If the XML element node we are searching allows *self nesting*, we take the first occurrence of the word placed after the position of the desired *start-tag* just located (the 2^{nd} occurrence of $\langle book$). If not, we take the next occurrence of the word after its corresponding *end-tag* (it is the case of the example, so we take the 7^{th} occurrence of *river*). In both cases, we repeat the whole procedure. Again, this allows skipping those occurrences of the element node that could not contain any occurrence of the word searched (in Fig. 3 we skip the 3^{th} occurrence of *book*).

Although we have explained the algorithm for the particular case of XML element nodes and words being part of their content, it can be generalized to predicates over other element nodes. That is, queries like *//tag_name*$_1$[*//tag_name*$_2$] but also *//tag_name*$_1$*//tag_name*$_2$.

Searching attributes values Another important query in XPath is to find all the occurrences of an attribute having a given value, being it a simple word or a phrase. That is, queries like @*att_name* = "*att_value*".

Whatever the case of the value, the algorithm to find out those attributes is that aimed at searching *phrase patterns*. That is, we will find all the phrase patterns given by the phrase built from the name of the attribute and its value: `att_name="att_value"`.

Other queries We have just explained a common subset of the XPath queries. However, any other one can be answered using the representation we have presented. Some other queries like, for example, *//tag_name* [*position() = i*] or

//tag_name [*position()* <= n] can be solved by simple locating the i^{th} occurrence or the n-first occurrences of the *tag*, respectively, instead of locating all as we have seen. If we want to cope with queries involving *parent* XPath axis, /, it is not hard to imagine how to incorporate it from the discussion about $//tag\_name_1[//tag\_name_2]$.

5 Experimental results

An isolated Intel®Pentium®Core 2 Duo 2.13 GHz system, with 2 GB dual-channel DDR-667Mhz RAM was used in our tests. It ran Ubuntu 8.04 GNU/Linux (kernel version 2.6.24.23). The compiler used was gcc version 4.2.4 and `-O9` compiler optimizations were set. Time results measure CPU user time in seconds.

The four different XML documents used to run our experiments are:

- `0.5d` and `9d` : files generated with *xmlgen*, an XML data generator developed inside *XMark Project* (http://monetdb.cwi.nl/xml/).
- `dblp` : file corresponding to the revision of April 16, 2008.
- `psd7003` : file of the public proteins database, *Integrated Protein Informatics Resource for Genomic and Proteomic Research* (http://pir.georgetown.edu/).

Table 1. Description of the documents used and compression properties.

XML doc.	size	EN	MD	VT	VNT	#T	#NT	R1	R2	CT	DT
0.5d	55,32	832	12	148	85	1,665	9,468	31.82	29.06	4.16	0.66
dblp	282,42	6,928	6	70	1,750	13,856	61,649	41.50	37.32	28.84	3.68
psd7003	683,64	21,305	7	128	3,142	42,611	106,621	41.29	40.35	60.43	6.84
9d	1007,12	15,040	12	148	743	30,080	171,595	31.28	28.57	69.61	12.24

On the one hand, Table 1 presents the name of the XML documents used, their size in MBytes, their number of XML element nodes (EN)(x10^3), their maximum depth level (MD), the number of different words in *tag words* (VT) and *text words* (VNT)(x10^3) vocabularies, and the number of *tag words* (#T) and *text words* (#NT) that compose each document (x10^3). On the other hand, the last four columns of Table 1 also show, respectively, the compression ratios (in %) obtained by XWT (R1) and the *(s,c)*-DC compressor (R2) over each XML document, as well as the compression (CT) and decompression (DT) times (in seconds) using XWT. Notice that XWT represents each XML file using about 30%-40% of its original size and, which is more striking, XWT only uses 3% more space than the needed to compress the documents with *(s,c)*-DC. That is, the powerful indexing capabilities of XWT only need 3% of extra space over the compressed text.

In Table 2 we can see the times obtained to answer the different common XPath operations explained in Section 4.2. The results presented are obtained using a XWT implementation with a waste of 3% of extra space for the structures of the partial counters used to speed up *rank* and *select* operations. From column 1 through column 12 we present the times obtained for *count* all the

Table 2. Searching operations.

	$1000 < f \leq 10000$				$100 < f \leq 1000$				$1 \leq f \leq 100$					
	count (μs)	first (μs)	all (ms)	snip. (ms)	count (μs)	first (μs)	all (ms)	snip. (ms)	count (μs)	first (μs)	all (ms)	snip. (ms)	TCW (ms)	ATT (ms)
0.5d	3.33	4.44	3.39	65.39	3.59	4.16	3.30	19.77	0.74	8.19	0.04	0.23	8.78	0.04
dblp	3.55	5.42	5.09	68.31	2.92	10.84	2.58	11.04	0.22	20.71	0.03	0.06	21.83	0.13
*psd*7003	3.04	6.64	6.06	55.65	3.40	6.33	2.06	7.27	0.41	15.80	0.04	0.10	62.08	0.02
9d	3.47	4.41	13.84	214.94	3.19	8.26	2.97	11.38	0.52	8.98	0.04	0.12	80.13	0.04

occurrences, locate the *first* position, locate *all* the positions, and extract all the 10-words *snippets* of a word. We distinguish 3 groups of words depending on their frequency f: *i)* $1000 < f \leq 10000$, *ii)* $100 < f \leq 1000$ and *iii)* $1 \leq f \leq 100$ and show the average time of searching for 100 distinct words (skipping *stopwords*) randomly chosen from the two vocabularies in each group.

In turn, columns 13 and 14 of Table 2 show, respectively, the average times obtained to locate all the occurrences of a certain pair of *start-end tags* containing a word (TCW) and to locate all the occurrences of an *att_name* = "*att_value*" pattern (ATT). In the first case, we have randomly chosen 100 *tags* and 100 *text words* from their respective vocabularies and have performed the algorithm. For the second operation, we used 100 randomly chosen pairs of the different *att_name* = "*att_value*" pairs with frequency between 1 and 100 existing in each XML document. Notice that here the search times of locate all the occurrences of a certain *att_name* = "*att_value*" pattern depend also on the number of words that form the "*att_value*". The greater the number of words, the fewer the number of false positives we find in the root of the XWT that will spend time being processed down in the XWT. Moreover, it also depends on the frequency of the least frequent word of the *att_name* = "*att_value*" pattern. The greater the frequency, the greater the number of possible candidates we will check.

To properly valuate these data we need to take into account that, long ago [12], it has been clearly established beyond doubts that any kind of word or phrase search over the uncompressed text takes up to 8 times more time than to perform the same search over the compressed text, due to the fact that processing the uncompressed text imply to process around three times more bytes. Therefore, it only makes sense to compare our data, about searches to answer the different XPath queries, with those that could be obtained using a compressed version of the text. But in [4] it was experimental tested the performance of WT against compressed text. Different compressors were used to create the compressed text and to obtain the codewords for the WT. In all the cases the WT was dramatically faster to perform any kind of search, thanks to the self-indexing properties.

As consequence, it is more interesting to compare our results against those that can be obtained using a compressed and indexed version of the XML documents. In [4] the performance of WT was compared against inverted indexes to blocks of text (not to individual words occurrences). Different block sizes in the inverted index and different number of partial counters in the WT were used in order to compare the efficiency of both approaches when using different amounts

of space. Results clearly proved the superior efficiency of the WT in searching words and phrases. The WT was superior even in recovering snippets when the amount of used space was inferior to the 40% of the original text size.

In our case, where not only we need to find a word, but also to process the text around it to know if that word is between some specific pair of *start-end tags*, or if it is an attribute value, etc. the use of our XWT will be even more advantageous.

6 Conclusions and future work

In this paper we introduce a strategy for compressing XML documents to about 35% of their size giving them, furthermore, self-indexing properties. This strategy is based on the use of a data structure we called XML Wavelet Tree (XWT). XWT is a new approach to the problem of storing, processing and querying XML documents in a time and space efficient way. Although our results are promising, more research must be done to improve the self-indexing properties. This is especially important for the tags representation, because most XPath queries imply the use of the document structure that the tags provide. On the other hand a systematic experimental evaluation of our XWT must be done comparing its performance with some of the other efficient XML representations.

References

1. Xml 1.0, W3C Recommendation of Extensible Markup Language (XML) Version 1.0 (Fifth Edition). `http://www.w3.org/TR/REC-xml`.
2. Xpath 2.0, W3C Recommendation of XML Path Language (XPath) Version 2.0. `http://www.w3.org/TR/xpath20`.
3. G. Bordogna and G. Pasi. Personalised indexing and retrieval of heterogeneous structured documents. *Inf. Retr.*, 8(2):301–318, 2005.
4. N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *SIGIR'08*, pages 139–146, 2008.
5. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. (s, c)-dense coding: An optimized compression code for natural language text databases. In *SPIRE'03*, LNCS 2857, pages 122–136, 2003.
6. N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Lightweight natural language text compression. *Inf. Retr.*, 10:1–33, 2007.
7. N. R. Brisaboa, A. Fariña, G. Navarro, A. S. Places, and Eduardo R. López. Self-indexing natural language. In *SPIRE'08*, LNCS 5280, pages 121–132, 2008.
8. J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *SPIRE'05*, pages 1–12, 2005.
9. N. Fuhr and K. Grobjohann. Xirql: A query language for information retrieval in xml documents. In *SIGIR'01*, pages 172–180, 2001.
10. H.-G. Li, S. A. Aghili, D. Agrawal, and A. E. Abbadi. Flux: fuzzy content and structure matching of xml range queries. In *WWW'06*, pages 1081–1082, 2006.
11. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
12. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *TOIS*, 18(2):113–139, 2000.