An Efficient Implementation of a Flexible XPath Extension *

Nieves R. Brisaboa
Fac. de Informática,
Univ. da Coruña,
A Coruña, Spain.
brisaboa@udc.es

Ana Cerdeira-Pena
Fac. de Informática,
Univ. da Coruña,
A Coruña, Spain.
acerdeira@udc.es

Gonzalo Navarro
Dept. Computer Science,
Univ. of Chile,
Santiago, Chile.
gnavarro@dcc.uchile.cl

Gabriella Pasi
DISCO, Univ. degli Studi
di Milano Bicocca,
Milano, Italy.
pasi@disco.unimib.it

ABSTRACT

In this paper we present an efficient implementation of different flexible queries (that constitute an extension of the XPath query language) to be executed on XML documents represented by using a recent structure called XML Wavelet Tree (XWT) [3]. A XWT represents the XML document compressed by using only about 35% of its original size, but it also provides some implicit self-indexing features that help to obtain not only efficient implementations of standard XPath queries, but also of extended ones. This is shown based on the implementation of the flexible structure based constraints, *below* and *near* [7].

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: Data Compaction and Compression; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms

Algorithms

Keywords

Querying XML documents, XML documents representation, XML documents compression, XML documents indexing.

1. INTRODUCTION

XML (Extensible Markup Language) is a World Wide Web Consortium (W3C) standard markup language that was originally defined as a simplified subset of the Standard Generalized Markup Language (SGML) for use on the World Wide Web. Since its first introduction in 1998, the language and its data model have soon proved their suitability to be the basis for the data interchange on the Internet. Today, XML is widely employed as a basic data model for semi-structured information, and is now the basic standard for representing semi-structured documents in Information Retrieval. It is widely recognized that in order to exploit the expressive power of XML, any retrieval system storing XML documents should allow constraint formulation on both document content and structure [2]. Current XML query languages like XPath and XQuery [2], however, assume that the user is fully aware of the target document structure, and allow only the formulation of exact queries [11]: either the desired XML information is retrieved exactly as it is in query formulation or the query result is empty. This assumption is debatable since most XML documents have no pre-set structure; even worse, it requires the user to write a different query for each variation of the document structure itself. In other words, current XML query languages do not support any diversity in either data structure or content within a document base. In order to tackle this problem, both Information Retrieval and Database communities have proposed several approaches. In Information Retrieval, the defined proposals are classified as content-only search (*CO*) and content and structure search (*CAS*) [1]. Both proposals aim at introducing some degree of flexibility in the retrieval of information stored in XML documents by either focusing on keyword-based queries (as used by unstructured Information Retrieval) or by taking into account the *structure* of the information without being so strict as traditional XML querying.

In [7, 9, 10] an extension of the XPath query language has been proposed, aimed at allowing the specification of flexible constraints on both textual content (in the IR style) and document structure. This way the user can decide the extent and the type of flexibility in identifying the relevant information items. In the above mentioned papers the addressed research problems are the following: *i)* definition of flexible constraints on the content of documents, *ii)* definition of flexible constraints on the structure of documents, in order to find close matches to structural query conditions; *iii)* focused search aimed at retrieving only the most relevant document fragments. One of the most important outcomes of this research is that the list of results produced by a flexible query evaluation is a ranked list of fragments, while the XPath query evaluation produces a set of results. In this paper we just focus on two structure based constraints first proposed in [7], namely *below* and *near*. The aim of this paper is to implement and to evaluate these constraints on a

*Funded in part by MEC (PGE and FEDER) grant TIN2009-14560-C03-02, for the Spanish group and the fourth author; by MICINN ref. AP2007-02484 (FPU Program) and Xunta de Galicia ref. IN809 A 2009/71 (INCITE Program) for the second author; and by Fondecyt grant 1-080019 (Chile), for the third author.

RIAO '10, 2010, Paris, France

compressed self-indexed representation of XML documents, the XML Wavelet Tree (XWT) [3].

Related to research in text compression, in [12, 8, 5, 6] some word-based byte-oriented semi-static statistical compression methods for text documents were presented and evaluated. All of them have the interesting property of being very useful for text retrieval purposes because the search of words and phrases can be done directly over the compressed text, without decompressing it, up to 8 times faster than over the uncompressed version [12]. Among those compressors the one called *(s,c)*-DC was used in [3] to encode XML documents using the structure we called XML Wavelet Tree.

XWT is a compressed and self-indexed representation of a XML document where the different bytes of the codewords used to represent words and *tags* of the XML document are placed in different nodes of a Wavelet Tree [4]. XWT is, therefore, a rearrangement of the codeword bytes similar to the one presented in [4]. In [3], a description of the XWT and a previous approach to its use were presented. As it was mentioned before, here we show how some useful queries that extend the XPath constraints can be efficiently evaluated by using the XWT.

2. STATE OF THE ART: XML WAVELET TREE (XWT)

In [4] we presented a rearrangement of the codeword bytes of a text compressed with any word-based byte-oriented semi-static statistical prefix-free compression method. This reorganization, called Wavelet Tree, consists basically on placing the different bytes of each codeword at different nodes of a tree instead of sequentially concatenating them, as in a typical compressed text. The XWT follows the same idea, but it encodes the text with a compressor called *(s,c)*-DC [6]. This compressor encodes each word with a sequence of bytes, in a way that allows to select a specific range of bytes to be used as the last bytes of the codewords. Those last bytes, called *stoppers*, mark the end of each codeword. XWT represents the text by using about 35% of its original size, and to build the XWT takes the same time as to compress the document, that is, the rearrangement of the codewords bytes is made with a very slight cost over the compression time.

To construct the XWT representation we create two different vocabularies, one for the *tags* and the other for the rest of the words in the XML document, and we flexibly select how many *stoppers* we want to use to identify the last byte of a *tag*, and how many stoppers we want to use to identify the last byte of *non-tag* words. This gives us the possibility of obtaining the best compression of each vocabulary. *(s,c)*-DC also allows to use specific first bytes to identify *tag* codewords. Therefore, just looking at the first byte of a codeword we can know if it is encoding a *tag* or a *non-tag* word of the text. In this way, as we will see later, specific nodes of the XWT are devoted to represent the *tags*, that is, the structure of the XML document. More details can be found in [3]; in this paper, by using comprehensive but easy examples, we will explain how a XWT is built and used.

To build a XWT we need to perform two steps or phases. The first step consists of parsing the input XML document to create the two vocabularies, one for *start-* and *end-tags* and the other for the rest of the words. In the left side of Figure 1 it is possible to see that all *tag* codewords start by the byte b_4, that the *stoppers* used for *non-tag* codewords are bytes b_1, b_2 and b_3 and that b_1, b_2, b_3, b_4 and b_5 are the *stoppers* for the *tag* codewords. The number of stoppers of each vocabulary depends on both its size and its frequency distribution of words, and it is automatically selected by the *(s,c)*-DC compressor to improve the compression [5].

The *parsing* process distinguishes different kind of words depending on their role [1]. For example, the word *love* appears in the text of the example, as an attribute value but also as a normal text word. Therefore it has two different entries in the vocabulary leading to two different codewords. This increases the vocabulary size but gives us flexibility and efficiency when querying. It is also in the *parsing* phase that some *normalization* operations take place.

Once codewords are assigned to words by using the *(s,c)*-DC encoding method, we do a second pass over the text by replacing each word by its codeword, and by storing these codeword bytes along the different nodes of the XWT. The root of the XWT is formed by a vector with all the first bytes of the codewords, following the same order as the words they encode in the original text. Each node X in the second level contains all the second bytes of the codewords whose first byte is x, following again the same order of the text. 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, and so on. Therefore, the resulting XWT has as many levels as bytes have the longest codewords.

In Figure 1 we can see that the fifth byte in the root is b_5 because the fifth word of the text, *Shakespeare*, is encoded as $b_5b_4b_3$. The second byte of its codeword, b_4, appears in the second position of node $B5$ because *Shakespeare* is the second word in the text encoded with a codeword starting by b_5. Its third byte is at the first position of node $B5B4$ because its second byte is the first b_4 in node $B5$, that is, *Shakespeare* is the first word in the text encoded with a codeword starting by b_5b_4.

2.1 Using XWT

The two basic procedures when using the XWT are to *search* for a word in the text, and to *decode* the word placed at a certain position. Both are easily solved using *select* and *rank* operations, respectively. That is, original codewords can be rebuilt from the bytes spread along the different XWT nodes by using *rank* operations and words can be efficiently found in the text, taking advantage of the self-indexing properties of XWT, by using *select* operations. Let B be a sequence of bytes, b_1, $b_2 \ldots b_n$. Then, *rank* and *select* are defined as:

- $rank_b(B,p) = i$ if there are i bytes b in vector B until the position p.
- $select_b(B,j) = p$ if the j^{th} occurrence of byte b in vector B is at position p.

The performance of the XWT depends on the implementation of the *rank* and *select* operations, because they are the base for any procedure over this structure [2].

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

[2] 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.

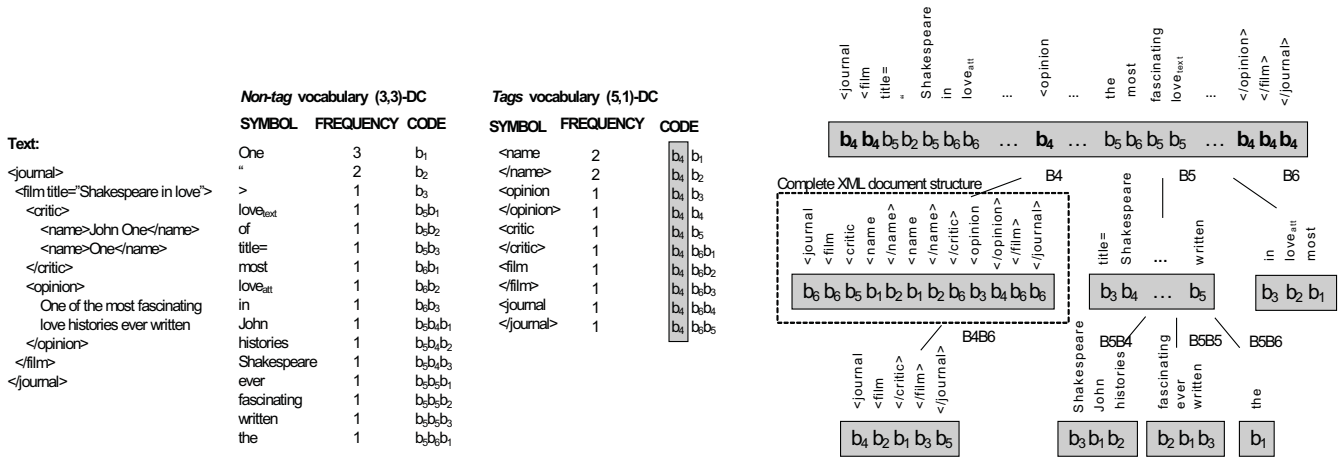

Figure 1: Example of XWT.

2.1.1 Searching in the XWT

To locate a word (typical XPath queries as *//book*, *//@title*, etc.) we search its last byte in the corresponding XWT node, and perform consecutive *select* operations up to the root. For example, to locate the first occurrence of *Shakespeare* in Figure 1, we begin looking at node $B5B4$, since the codeword of *Shakespeare* is $b_5b_4b_3$. There, we search the byte b_3 computing $select_{b_3}(B5B4, 1) = 1$. In this way, we obtain that the position of node $B5B4$ where the first occurrence of byte b_3 (the last byte of *Shakespeare*) is represented, is 1. We know that all the words whose last bytes are stored in node $B5B4$, are represented in node $B5$ with a byte $b4$, and that they are in the same text order. Therefore, the value 1 we obtained with the *select* operation indicates that the first byte $b4$ in node $B5$ corresponds to the first occurrence of *Shakespeare* in the text. Again, we compute $select_{b_4}(B5, 1) = 2$, that newly indicates that our codeword is the second one starting by b_5 in the root node. Finally, by calculating $select_{b_5}(root, 2) = 5$, we can answer that the first occurrence of *Shakespeare* is at 5^{th} position in the text.

If we want to locate all the occurrences of a word, this process is repeated for each of them. Since the traversed XWT nodes are the same for each occurrence, and these will be processed consecutively, both the select operations and consequently the whole process, can be sped up by using pointers to the already found positions in the XWT nodes.

To locate a *phrase* pattern we start by 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 the previous and next bytes of the root node. If those matches happen, we continue by validating the rest of the bytes of the corresponding codewords. But if it is not the case, we avoid going down into the XWT, and we simply locate the next occurrence of the least frequent word to be processed in a same way.

If we are interested in locating element nodes containing a certain word to solve XPath predicates over the text, like *//title* $[contains(., Egypt)]$, we use a procedure that allows to save processing time by skipping some text. Here this procedure will not be detailed [3], as this global strategy will be later explained, but applied to a more complex query (see Section 4.2).

As shown in [3] those operations are performed very efficiently requiring just some milliseconds as average, which means that any search can be done a lot much faster than over the plain XML text or even over the compressed text. In [4] we also showed that the strategy of placing codewords bytes in nodes of a wavelet tree compete with the classical inverted index when little space is available[4]. Specifically, by spending about 40% of the size of the original text to represent both the compressed text and the inverted index, and by spending the same space in the wavelet tree, the last is faster than the inverted index in any task.

2.1.2 Decompression

To decode a word we use *rank* operations. For example, to know which is the second word in the source text of Figure 1, we start by reading $root[2] = b_4$. According to the encoding scheme, we know that the codeword is not complete, so we will have to read a second byte in the second level of the XWT, more precisely, in the node $B4$. To find out which position of that node we have to read, we use $rank_{b_4}(root, 2) = 2$ that means that there are 2 bytes b_4 in the root until position 2. Therefore, $B4[2] = b_6$, gives us the second byte of the codeword. Again b_6 is not a *stopper*, so we need to continue the procedure. In the child node $B4B6$, that corresponds to the two first bytes of the codeword we are decoding, we have to read the byte at position $rank_{b_6}(B4, 2) = 2$. Finally, we obtain $B4B6[2] = b_2$. But b_2 is a *stopper* and, therefore, it marks the end of the searched codeword, that have resulted to be $b_4b_6b_2$, corresponding to the tag $\langle film$, which is precisely the second word in the source text, as expected.

If we want to decompress the whole text 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 by 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 which byte

[3]A complete explanation of the specific implementation for this case can be consulted in [3].

[4]That is, when pointed blocks of text are not to small.

of this child node sequence we have to read. It always will be the next one to process in that child node.

In [4] we showed that decompressing the whole text takes approximately the same time when the text is settled in a wavelet tree shape than when it is not rearranged.

3. FLEXIBLE EXTENSIONS OF XPATH

The W3C has defined the language XPath for selecting XML node sets via *tree traversal* expressions. XPath selection is Boolean in nature: it partitions XML nodes into those which fully satisfy the selection condition, and those which do not. However, Boolean conditions are, in some scenarios, not suitable for effectively querying XML documents. To justify this claim we note that even when XML schemas do exist, they may be not available to users. Moreover document trees with the same schema may be very different (both in used *tags* and nesting), and hence the schema will allow for diverse instantiations, making it difficult to predict a particular document structure (from the schema). Finally, the same XML tree can be sometimes described using different schemas. As a consequence, users often end up in defining *blind* queries, i.e. queries written without a precise knowledge of the schema. In these cases the availability of a flexible query language allowing for approximate queries can be of great help. In [7, 9, 10] a new approach aimed at introducing flexibility in XPath has been presented, based on the definition of some flexible constraints on both XML document structure and content; the flexible constraints can be specified as extensions of the XPath syntax.

In this paper we address the problem of implementing two flexible structure based constraints, namely *below* and *near*, on a compressed self-indexed data structure able to both represent and enquire XML documents, the XWT, explained in Section 2. In this section both the semantics and the evaluation function of the two flexible constraints are summarized.

3.1 Flexible structure based constraints

The constraint *below*, inserted as flexible axis of a path expression, is defined to the aim of selecting elements nodes, attributes or text from all nodes that are direct descendants of the current element node. For instance, the following query: *//books BELOW //author* retrieves all document fragments that contain the element node *author* which has the tag *books* in its path from the root. In Figure 3 a), 3 b) and 3 c), some examples of possible fragments retrieved in response to the previous query are shown. The constraint *near*, inserted as a flexible axis of a path expression, is defined to the aim of selecting elements, attributes or text from all element nodes that are either descendants, ancestors or that *surround* the current node.

For example, the following query: *//books NEAR //author* retrieves all document fragments that contain the element node *author* which has the node *books* in its surroundings but not necessarily in the same path. In Figure 3 d) an example of fragment that would be recovered in response to the previous query is shown. We can say that *near* represents a generalization of *below*, since the similarity evaluation is performed within any axis.

In order to introduce the function that defines the flexible matching between the flexible query path and a document path, we first informally analyze the constraint imposed by the *below* operator, and by the *near* operator subsequently. A query such as *//A BELOW//B* identifies all paths in which the node B is direct descendent of the node A. This means that the following paths are *compatible* with the above query: *i*) A/B, *ii*) A/∗/B, *iii*) A/∗/∗/B, *iv*) A/∗/∗/∗/B, etc. where ∗ represents a node in the path from A to B. In theory, infinite paths are identified by the flexible query *//A BELOW//B*; in practice, a maximum path length is defined, which is dependent on the considered XML document, and can be set to the maximum length that a path can assume in that document.

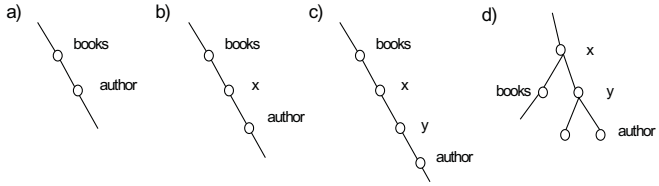

Figure 3: Sample fragments retrieved by BELOW (a), b), c)) and NEAR (d)) queries.

From a semantic point of view, the path that matches at best the flexible query (i.e. the *ideal* fragment) is the one in which B is child of A, that is, A/B. The higher the distance between A and B, the lower the relevance of the path to the query. As a consequence of this assumption, the function evaluating the match between the flexible query path and a document path should be inversely proportional to the distance between A and B. We propose to define such a function as: $Match(q_p,\ d_p) = 1/d(A, B)$, where q_p is the flexible query path, d_p is the considered document path, and $d(A, B)$ is the distance between nodes A and B in the document path d_p. We define $d(A, B)$ as the number of arcs between nodes A and B. The value $Match(q_p,\ d_p)$ is also called the *Retrieval Status Value* (RSV) of a fragment with respect to the query.

Here below an example of paths retrieved in response to the query *//location BELOW//author* is presented together with their RSVs. The query asks for an *author* element node which contains the *location* element node as ancestor in its path. The example supposes that the maximum path length in the considered document is equal to 3:

i) location/author, RSV=1
ii) location/∗/author, RSV=1/2=0.5
iii) location/∗/∗/author, RSV=1/3=0.33

In the case of the *near* constraint evaluation, the paths that match at best the flexible query are the following ones: A/B and B/A. Also in this case the higher the distance between A and B, the lower the relevance of the considered path to the query, by still implying the definition of a matching function like: $Match(q_p,\ d_p) = 1/d(A, B)$, where q_p is the flexible query path, d_p is the considered document path, and $d(A, B)$ is the distance between nodes A and B in the document path d_p (defined again as the number of arcs between nodes A and B). Like in the case of the *below* constraint, the value $Match(q_p,\ d_p)$ is also known as the *Retrieval Status Value* of the fragment with respect to the query. Given the nature of *near* constraint, that includes a broader selection context, in practice we consider a maximum path length selected by the user. For instance, *//location* $NEAR_3$ *//author* will retrieve all document fragments that contain the element node *author* which has the node *location* in its surroundings at a distance not higher than 3.

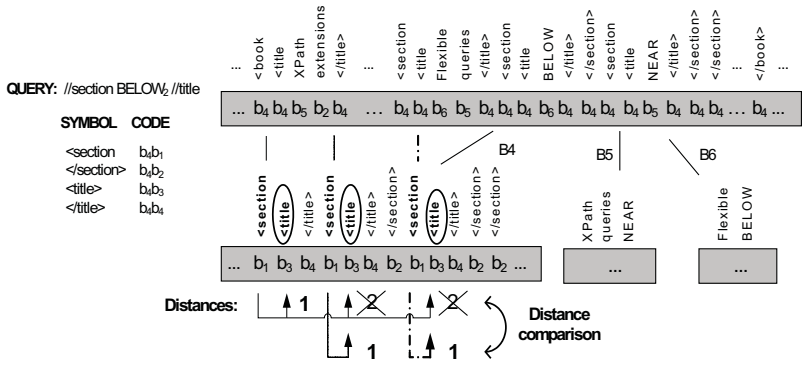

Figure 2: Example of BELOW constraint.

4. IMPLEMENTATION OF FLEXIBLE 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, and any extension of them can be answered by using the XWT but taking advantage of the implicit indexing properties provided by the XWT structure itself.

4.1 Answering flexible structure based queries

In XWT we use specific byte(s) to mark the beginning of a *tag* codeword. In Figure 1 the byte b_4 (bytes shaded in the `CODE` column of the *tags* vocabulary) is the first byte of *tag* codewords. As a result, all tags are in the same branch(es) of the XWT (branch $B4$ in the example), where they are placed following the document order. Therefore we can solve structural queries by using only those nodes of the XWT [5].

4.1.1 BELOW

Although the *below* operator does not impose any constraint on the maximum distance between the involved nodes, and therefore, the only limit is the maximum length that a path can assume in the specific XML document, our implementation sets a limit (because we need to call a limited *below* function in the implementation of the *near* operator). As a consequence, in what follows we explain our implementation by using always a parameter of maximum distance allowed between the nodes. It is clear that when the user performs a *below* query, we set the maximum distance parameter to the maximum depth of the XML document.

The *below* constraint can be solved in two different ways, depending on the frequency of the element nodes that are involved into the query. For example, let us suppose to have the following query: *//section* $BELOW_2$ *//title*. That is, we are interested in retrieving all the *title* element nodes that are descendants of a *section* element node, and whose distance is up to 2. In this way, the first step is to compute the frequency of each element node, with a simple *count()* operation [3]. Once this information has been obtained, we begin locating the first occurrence of the less frequent element node. In the case of being the child element node (in this example, *title*), we simply check its ancestors until reaching the maximum path distance (in this case, 2) or until an occurrence of the parent element node (in the example, *section*) is found. In the opposite case, we locate the first occurrence of *section*, and then we look for an occurrence of *title* by checking all its descendants that are not farther than a 2 distance [6]. Then, the corresponding procedure is repeated with the next occurrence of the less frequent element node.

Remember that the *below* constraint evaluation is based on computing the distance between the nodes involved in the query to give to the retrieved element nodes a penalty proportional to it. So, if $self\ nesting$ is allowed, we have to take into account it in the global procedure, when the parent element nodes are the less frequent ones. Each time an occurrence of a child node we are looking for is found, we have to check if it has already been reported by another occurrence of the parent element node. In this case, a comparison between the previous related distance and the new one is done, to chose the best one. In Figure 2, an example of this situation is shown. There we can see that the second and third occurrence of *title* are first retrieved by the first occurrence of *section* at distance 2. However, these distances are then updated to the better value 1, when the second and third occurrence of *section* are respectively checked.

Although here we have just explained the algorithm in the case where the retrieved nodes are the specified on the right side of the query, a similar procedure can be applied when the retrieved nodes are the ones specified on the left side (e.g. *//section*$[BELOW_2$ *//title]*).

4.1.2 NEAR

This constraint not only involves descendants and ancestors, as *below* does, but also another kind of relationship. Let us consider the following example: *//award* $NEAR_3$ *//author*. In this case, we want to find all the *author* element nodes, that are *near* an *award* element node. This one can be not only in the same path from the root as a descendant or an ancestor, but also it can appears in its document surroundings. The only restriction is that the distance between both of them can not be greater than 3.

[5]We also have a bit structure to speed up the navigation through the *tags* of the document. Given a position of an element node, that very well known structure [13] , can directly report the position of its parent and its different children in constant time. We do not explain that structure here for space constraints and because it is not necessary to understand the procedure, but only its efficiency.

[6]In this situation, efficiency can be influenced by the number of descendants.

As it has been reported in Section 3.1, the *near* constraint is a generalization of *below*. Therefore, to evaluate *near* we proceed in the following way. We begin by doing two *below* operations, *//award* $BELOW_3$ *//author* and *//author* $BELOW_3$ *//award*, respectively (see Figure 4 a)). By doing this, we cover all the descendant and ancestor situations that could happen between *award* and *author* in the XML document. After that, the base algorithm consists of locating each occurrence of the less frequent element node, and checking the descendants of consecutive ancestors, until reaching the maximum distance. To better understand, let us suppose that, in our example, *award* is the less frequent element node. Once located an occurrence of *award*, we start locating its first ancestor and then we look for an occurrence of *author* between the descending element nodes up to the 2^{nd} level. That is because the first ancestor is at distance 1 and the maximum allowed distance is 3 (see Figure 4 b)). Then, the same procedure is repeated, but taking the second ancestor, and checking the descendants up to the 1^{st} level (since the second ancestor is at distance 2 and the maximum allowed distance is 3. See Figure 4 c)). This procedure will be repeated until reaching the ancestor's order that matches the maximum query distance minus one, reducing the maximum level of the descendants to be visited at each step.

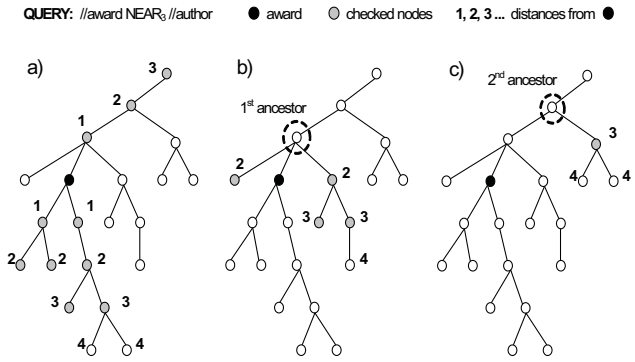

Figure 4: Example of NEAR constraint: a) BELOW operations, b) Descendants of 1^{st} ancestor, c) Descendants of 2^{nd} ancestor

The slight differences of performance of this global procedure, depend on which element node of the two ones involved in the query is the less frequent. In case it is the one that appears in the left side of the query, each time an occurrence is checked, we can save processing time by avoiding to visit ancestors already visited by another previous occurrence at the same distance. In case the less frequent is the one on the right side, we stop the procedure as soon as an occurrence of the left side element node is found, and if it is not possible to find another one at a better distance. In this last situation, it is also possible to avoid visiting an ancestor and its corresponding descendants, but only when it was visited by a previous occurrence, and any left side node was found.

Again, as it has been mentioned for the *below* constraint, we always keep the best distance found for each occurrence of the retrieved element nodes. Also, a similar procedure is performed in case the left side element nodes of the query are the ones we want to retrieve (e.g. *//award*$[NEAR_3$*//author]*).

4.2 Answering flexible structure based queries with content based constraints

Queries involving only flexible structure based constraints are answered using the procedures explained in Section 4.1. To evaluate queries with constraints on both the structure and the content, we can use a more sophisticated strategy.

In this case, we are interested in solving queries like for example: *//city* $[contains(., Paris)]$ $BELOW_2$ *//museum* or *//description* $[contains(., economy)]$ $NEAR_3$ *//ref*. The first query looks for all *museum* element nodes that are *below* a *city* at a distance not greater than 2 but also containing the word *Paris* (e.g. $\langle city\rangle \ldots Paris \ldots \langle museum\rangle \ldots \langle/museum\rangle \ldots \langle/city\rangle$). In the last example, we want to retrieve all ref element nodes that are *near* a *description* element node not farther than 3 arcs, and which must contain the word *economy* (e.g. $\langle book\rangle \ldots \langle description\rangle \ldots \langle summary\rangle \ldots$ *the national economy in* $\ldots \langle/summary\rangle \ldots \langle/description\rangle \ldots \langle ref\rangle \ldots \langle/ref\rangle \ldots \langle/book\rangle$).

We will use the first example (that is, *//city* $[contains(., Paris)]$ $BELOW_2$ *//museum*) to explain the procedure to answer this kind of queries (see Figure 5). First, we begin by locating the first occurrence of *Paris* in the root node. Then, by counting the number of occurrences of the *start-tag* $\langle city$ placed before that position and that of the corresponding *end-tag*, $\langle/city\rangle$, we know how many *city* element nodes contain that occurrence of the word. However, we are only interested in those occurrences of *city* that also satisfy the *city* $BELOW_2$ *museum* constraint, so this operation is performed with each located occurrence of *city*. In the example, we can easily see that the first occurrence of *Paris* is surrounded by the first occurrence of *city*. Nevertheless this one does not fulfill the *below* constraint, and therefore it is not considered.

Now, instead of performing the same process with the next occurrence of the word (in the example, the 2^{nd} occurrence of *Paris*), we can skip some text looking for the next occurrence of $\langle city$ placed after the position of the just located occurrence of *Paris*, but which also must satisfy the constraint *city* $BELOW_2$ *museum*. As it is shown in Figure 5, the third occurrence of $\langle city$ is the one we are looking for, not the second one, because it does not fulfill the structural constraint. Then we look for an occurrence of *Paris* inside it. Given that there is one occurrence (the 5^{th} occurrence of *Paris*), the *museum* element nodes related to that occurrence of *city* (the ones marked in bold face in Figure 5) are reported as results to be ranked. Notice that by doing this, we skip the occurrences of *Paris* that are before the third occurrence of $\langle city$, and which are not interesting for the search (see the occurrences of *Paris* inside the striped rectangle in Figure 5). After that, the whole process is repeated by taking the first occurrence of *Paris* placed after the position of the just located third occurrence of $\langle city$ (in case $self\ nesting$ is allowed) or after the position of its corresponding *end-tag* (if $self\ nesting$ is not allowed). Again, this allows skipping those occurrences of the *city* element node that not contain any occurrence of *Paris*, thus avoiding to process them (see the occurrences of *city* inside the striped rectangle in Figure 5).

Note that this same procedure can be used not only with the flexible structural constraints *below* and *near* presented in this paper, but also with any other structural constraint, and when the content based constraint is applied over a phrase or even a certain attribute value, as well.

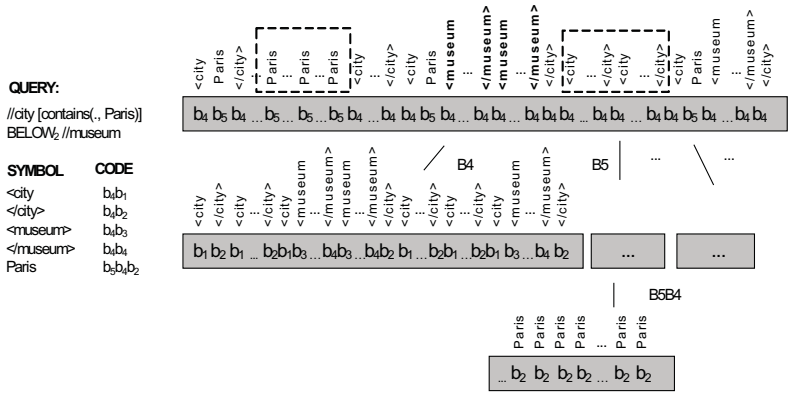

Figure 5: Example of a flexible structure based query with a content based constraint.

5. EXPERIMENTAL RESULTS

We have implemented the flexible constraints detailed in Section 4, and run some experiments aimed to evaluate the efficiency of the XWT representation in answering flexible queries.

An isolated Intel® Pentium® Core 2 Duo 2.13 GHz system, with 4 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. We used three different XML documents to run our experiments:

- `nasa`: file from the *NASA XML Project* (http://xml.nasa.gov/).
- `0.5d` and `1d`: files generated with *xmlgen*, an XML data generator developed inside the *XMark Project* (http://monetdb.cwi.nl/xml/).

A general description of the XML documents used is shown in Table 1. It presents, for each document, its size in MBytes, its number of XML element nodes (EN)(x10^3), its maximum depth level (MD), the number of different words in *tag* (VT) and *non-tag* (VNT)(x10^3) vocabularies, and the number of total words (x10^3), following that division (#T and #NT), that compose the document. The last three columns of Table 1 also show, respectively, the compression ratios (in %) obtained by XWT (R), as well as the compression (CT) and decompression (DT) times (in seconds). Notice that XWT represents each XML file using only about 30%-35% of its original size. To run the experiments, we have used a XWT implementation with a waste of 3% of extra space for the structures of partial counters used to speed up *rank* and *select* operations.

Table 1: Documents and compression properties.

doc.	size	EN	MD	VT	VNT	#T	#NT	R	CT	DT
nasa	23,89	476	8	122	78	953	4,236	31.64	1.99	0.28
0.5d	55,32	832	12	148	85	1,665	9,468	32.18	4.28	0.66
1d	111,12	1,666	12	148	128	3,332	18,991	31.91	8.28	1.32

On the one hand, we evaluated the XWT performance in answering flexible structure based queries containing the *below* and *near* constraints (e.g. *//section* $NEAR_3$ *//book*). On the other hand, we also used sets of flexible structure based queries with a content based constraint (e.g. *//section* [*contains*(., *Europe*)] $NEAR_3$ *//book*), like the sample queries seen in Section 4. In both cases, we distinguished 2 groups of queries depending on the frequency f (high or low) of the involved element nodes, according to each document features. Indeed, we have also set a maximum distance parameter in all the experiments, as the execution time depends on it.

In the first scenario (see Table 2), 25 different pairs of element nodes were randomly chosen from each frequency group for the `nasa` and `0.5d` XML documents. In this way, we created two sets of 25 flexible queries (one for each document), that were then evaluated by using the *below* and *near* constraints, with maximum path distances of 2 and 3. Notice that the same set of queries used for the evaluation of the `0.5d` XML document, was also used for the evaluation of the `1d` XML document, since both documents share the same structure and *tag* vocabulary.

Table 2: Average execution times of flexible structure based queries.

		nasa		0.5*d*		1*d*	
		BLW (ms)	NEAR (ms)	BLW (ms)	NEAR (ms)	BLW (ms)	NEAR (ms)
Dist.2	f low	7.30	18.33	37.02	228.84	74.34	504.81
	f high	33.26	109.26	93.16	286.43	194.73	582.24
Dist.3	f low	7.73	25.93	37.81	**577.41**	75.14	**1121.54**
	f high	33.46	137.73	97.64	**429.96**	200.52	**1097.09**

Table 2 summarizes the average times obtained to answer those queries. The results show the good performance of the XWT representation, but also some other singular features, like the influence of the maximum path distance, that deserves a particular discussion. As expected, the greater the maximum path distance the greater the time consumed to answer the same group of queries, but note that this difference of time is lower in *below* queries than in *near* ones. That is because, in *below* queries, each increment of the distance implies to check the descendants of one more level down, but only in case of element nodes that have enough nesting level; or to check only the ancestor of one more level up, in the best scenario (when the less frequent element nodes are the ones on the right side of the query). However, when we work with the *near* flexible constraint, to increase the distance means to visit the ancestors of one or more levels up and then to check all its descendants until a certain level. The upper the ancestor to be visited the higher the possibil-

ities to be closer to the root, and hence to have to check a greater number of descendants. What is even more, this is also the reason for finding a change of performance between queries involving low frequent element nodes and queries involving high frequent element nodes. As it can be seen, the average times of the first ones is lower than the average times of the second ones when using a distance of 2, since there is a lower number of element nodes to be evaluated. However, the opposite situation is also possible, when the distance is increased (see values marked in bold face in Table 2). Since the less frequent element nodes use to lie in the levels close to the root in the XML document structure (unlike the most frequent ones), and they usually have a higher number of descendants; to increase the distance can imply to visit a considerably higher number of descendants element nodes consuming more processing time. While, in the case of the most frequent element nodes (normally placed at the deepest levels of the XML document structure), this situation is less probable.

Table 3: Average and standard deviation of the execution times of flexible structure based queries with a content based constraint.

*Dist.*3		*nasa*		0.5d	
		BLW (ms)	NEAR (ms)	BLW (ms)	NEAR (ms)
Avg	f low	2.81	4.21	**202.13**	**3790.61**
	f high	21.79	**472.84**	49.32	**2140.43**
Std	f low	4.10	7.83	**503.16**	**4438.68**
	f high	20.09	**1033.16**	37.02	**2088.35**

To run the experiments in the second scenario, related to the flexible queries mixing the structure based constraints, *below* and *near*, with a content based constraint, we used the same sets of pairs of element nodes that in the queries of the previous scenario, and we also used words of frequency between 50 and 100 randomly chosen from the *non-tag* vocabularies. Table 3 shows the average times obtained to answer the queries by using a maximum path distance of 3. On the one hand, we can see the benefits of applying the jumping strategy explained in Section 4.2, that saved processing time, if we compare the underlined values in Table 3 with respect to the corresponding ones in Table 2. On the other hand, and to properly valuate the data in which this benefit is not appreciated, it is important to remark that the benefits of that strategy may not be always profited. This is the case of queries where the difference of frequency of the involved element nodes is significant and the container element node is the most frequent, and also when there are a few number of structurally related element nodes through the *below* or *near* constraints. These kind of queries can have a strong influence over the average. As a consequence, we have also computed the standard deviation, corresponding to the different average times, that are shown in Table 3, as well. Notice that the values of this table where the benefits of the jumping strategy are not easily appreciated, correspond exactly to the values with the highest standard deviation (see values marked in bold face in Table 3).

6. CONCLUSIONS AND FUTURE WORK

In this paper we have shown how different flexible extensions of the XPath query language can be efficiently implemented over the compressed and self-indexed structure we called XWT. This structure represents the XML document compressed in about 35% of its original size giving, at the same time, some interesting implicit self-indexing properties.

In XWT, the structure of the documents, provided by the XML element nodes, is represented in a very compact way in specific nodes of the XWT. The XWT also uses a bit structure to speed up the navigation through the element nodes providing, in a efficient way, the parent of any element node or even the i-th child. In this way, pure structural queries as well as queries involving constraints on both document structure and textual content can be efficiently solved.

We have also explained how any query can be answered by using XWT because it is really a representation of the XML document and therefore any query that can be performed over the XML document can be also answered using its XWT representation, but a lot more efficiently, because XWT is compressed, and has self-indexing capabilities. Some experiments were done to prove the efficiency of using the XWT over flexible queries involving *below* and *near* constraints, but more work needs to be done to improve the execution plan for any query and more extensive experiments will be needed by taking into account some other interesting features, like the depth level of the element nodes involved in the queries, or even their number of children.

7. REFERENCES

[1] INitiative for the Evaluation of XML Retrieval. `http://inex.is.informatik.uni-duisburg.de/`.

[2] XML 1.0, XPath 2.0 and XQuery 1.0, W3C Recommendations. `http://www.w3.org/TR`.

[3] N. R. Brisaboa, A. Cerdeira-Pena, and G. Navarro. A compressed self-indexed representation of xml documents. In *ECDL'09*, pages 273–284, 2009.

[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] A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi, and P. Spoletini. A fuzzy extension of the xpath query language. *J. Intell. Inf. Syst.*, 33(3):285–305, 2009.

[8] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *SPIRE'05*, pages 1–12, 2005.

[9] E. Damiani, S. Marrara, and G. Pasi. A flexible extension of xpath to improve xml querying. In *SIGIR'08*, pages 849–850, 2008.

[10] E. Damiani, S. Marrara, and Ga. Pasi. Fuzzyxpath: Using fuzzy logic an ir features to approximately query xml documents. In *IFSA'07*, pages 199–208, 2007.

[11] E. Damiani and L. Tanca. Blind queries to xml data. In *DEXA'00*, pages 345–356, 2000.

[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.

[13] K. Sadakane and G. Navarro. Fully-functional static and dynamic succinct trees. *CoRR*, abs/0905.0768, 2009.