Volume 5, Issue 4, 2015 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Optimized Indexes for Data Structured Retrieval

Yosvanys Aponte Báez[*]
Agrarian University of Havana
San José de Las Lajas, Cuba

Alexander Sánchez Díaz
Agrarian University of Havana
San José de Las Lajas, Cuba

Manuel Marco Such
University of Alicante
Alicante, Spain

Abstract— The aim of this work is to show the novel index structure based suffix array and ternary search tree with rank and select succinct data structure. Suffix arrays were originally developed to reduce memory consumption compared to a suffix tree and ternary search tree combine the time efficiency of digital tries with the space efficiency of binary search trees. Rank of a symbol at a given position equals the number of times the symbol appears in the corresponding prefix of the sequence. Select is the inverse, retrieving the positions of the symbol occurrences. These operations are widely used in information retrieval and management, being the base of several data structures and algorithms for text collections, graphs, trees, etc. The resulting structure is faster than hashing for many typical search problems, and supports a broader range of useful problems and operations. There for we implement a path index based on those data structures that shown to be highly efficient when dealing with digital collection consist in structured documents. We describe how the index architecture works and we compare the searching algorithms with others, and finally experiments show the outperforms with earlier approaches.

Keywords—XML, indexing, suffix array, suffix tree, rank and select

I. INTRODUCTION

XML has become the new standard for Internet data representation and exchange in the last years, and also is widely used in several applications like electronic encyclopedias, digital libraries, on-line manuals, linguistic databases, scientific taxonomies, between others. As more and more files occurred in this format, we wanted to access the stored data and search for the specific according to prior criteria. Languages like XPath [1] and XQuery [2] have been created for searching elements, attributes or text values in those kind of documents. The challenge is to find these elements rapidly and efficiently, especially when we are dealing with big and heterogeneous structured documents.

In some cases, we have some datasets with a great number of paths and nodes and making a lot of queries with content and with direct and indirect containment could produce a time-consuming task if you didn't have an efficient data structure. Indeed, indexing such large heterogeneous document collections therefore requires the implementation of memory efficient data structures which store the structural indexes.

A lot of indexing techniques have been proposed for improving the performance of index and query processing. As is explained in [3] they can be divided into different categories, but in general all of them suffer from some of the following problems: the index size require a huge space, and in some cases they could be bigger than the original document and also could have a big scalability problem. There is a high cost with respect of index time construction and the query evaluation procedure. Finally some of them cannot support complex queries efficiently.

This paper describes how the structural index can be implemented as a *suffix array* [4] and complemented, for faster response, with a *ternary search tree* [5] and *rank and select* succinct data structures, for that propose we implemented a prototype named **SATRS**. On one hand, *suffix arrays* have proved to be extraordinarily effective in the indexing of plain text, and they combine fast access ---particularly when locating sequences of words--- with moderate memory requirements. On the other hand, *ternary search trees* combine the attributes of binary search trees and digital search tries, and are also space efficient. And finally the data structures *rank and select* are widely used in information retrieval and management, being the base of several data structures and algorithms for text collections, graphs, trees, etc. Where ***rank*** of a symbol at a given position is equal to the number of times the symbol appears in the corresponding prefix of the sequence and ***select*** is the inverse, retrieving the positions of the symbol occurrences.

The reminder of the paper is organized as follow. In section **Related work** we briefly present some research results in the literature of XML indexing. Section **Structural index** describes our implementation of the structural index for XML retrieval based on *suffix arrays*, *ternary search trees* and *rank and select* succinct data structures. Section **Experimental results** shows the outperforms with earlier approaches and section **Conclusions** summarizes our conclusions and future work.

II. RELATED WORK

A lot of indexing techniques have been proposed in the literature for improving the performance of query processing. They can be divided into three main categories: *structural summaries*, *structural join* and *sequence-based indexes*.

The *structural summary* approach reduces the portion of the XML to be scanned during query processing. For example *Dataguide* [6], the most referenced method of its type, summarized all label paths in the XML document and provide key benefits afforded by a schema, such as guidance to the user for query formulation and guidance to the query processor for query optimization. The problem with *DataGuide* is when is used with highly heterogeneous tree collections that contains considerable structural redundancy and does not support multiple and branching path expressions. Another approach is *IndexFabric* [7] that eliminates the step of intersection between the path index and the keyword index, the author uses only one index to content and structure, one of the weakness of this approach is that the resulting index is too large to be held in main memory, due to the author proposed a paging strategy for partitioning the *IndexFabric* on disk in order to restrict the number of page faults during the index look-ups.

Another work is *T-Index* [8] that captures the knowledge about the structure of data and the type of queries in the query mix, described as path templates. Depending on the path template, a *T-Index* may capture more or less of the document structure than the *DataGuide*. They discuss two particular variants of the *T-Index*. The first one *1-Index* covers all tag paths starting from the document root, and the second one *2-Index* locates all pairs of ancestor an descendant elements that are linked by a specific sequence of tags. One limitation is that *2-Index* can have quadratic size in the worst case.

The *PCIM* [9] (Path Clustering Indexing Method) clusters paths with the same root-to-leaf nodes and reduces the space cost of the index using two hash tables, the Structural Index and the Content Index, with tag names as hashing keys for efficient searching. The *PCIM* reduces the index space with a high compression ratio and efficiently process complex queries. But has the following disadvantages: a long time index construction and the adoption of a regional numbering scheme. The *NCIM* [10] (Node Clustering Indexing Method) is another indexing scheme, which differs from the *PCIM* by clustering the nodes with the same tag names and storing them in hash tables. Showing a good compression ratio and supporting complex queries efficiently. A limitation is that they assume that the indexes can fit into main memory and it is an issue when dealing with large XML documents. Both the *PCIM* and *NCIM* encode nodes by means of regional numbering scheme. However, the *PCIM* uses strings to represent labels and the *NCIM* uses integers where possible. As is discussed in [11], in most cases, the *NCIM* outperforms the *PCIM*, because the *PCIM* stores text content in the other tables, whereas the *NCIM* stores them in the leaf node index under the corresponding tag name, which results in reduced search time for processing queries with the selection predicates.

Most of the *structural join* indices are based on the decomposition-matching-merging processes. For example Zhang et al. [12] and Al-Khalifa et al. [13] have proposed the MPMGJN and Stack-Tree algorithms respectively to match binary structural relationships. However, these approaches still produce large intermediate results. To solve this problem, Bruno et al. [14] proposed PathStack and TwigStack: holistic path and twig join algorithms, which use a chain of linked stacks to represent the intermediate results in a compact manner, and subsequently join them to obtain the final results. This algorithm is only optimal for A-D relationships. Thus in [15] have extended TwigStack and proposed TwigStackList, which can support both P-C and A-D relationships efficiently.

And finally the *sequence-based indexes* convert XML documents and queries into structure sequences. They put the values and the structures of XML data together into an integrated index structure. For answer a query, they make a string sequence that matches the sequence of the data with the query. These methods reduce the need of joins to evaluate twig query. For example VIST [16] (Virtual Suffix Tree) labels nodes in pre-order traversal, and is based on B+ tree. VIST has the disadvantage of weakening the query operations due to the large number of nodes being checked and the use of a top-down sequence. Consequently the size of the index becomes very large when dealing with large XML documents since the top elements are added into the sequence. The PRIX [17] (Pruffer sequence for indexing XML) does a good job in decreasing the query processing time and solve the scalability problem of VIST. At last on challenge of the sequence-based methods is how to avoid false alarm and false dismissal. On the other hand Ferragina [18] also proposed *xbw* transform of a labelled tree based on Burrow-Wheeler transform [19] for strings to compress, index and process XML data. The *xbw* transform uses path-sorting and grouping to linearize the labelled tree into two coordinated arrays, one capturing the structure and the other the labels. Positive in this method: very good for queries and compression. Negative, the label construction is rather computational expensive and does not support dynamic XML.

III. STRUCTURAL INDEX

The structured text stored in XML files can be represented as a tree that can have element nodes, attribute nodes and textual nodes. For example in Figure 1 the tree shown has five element nodes (one labelled with a, two labelled with b, two labelled with c and one labelled with @d) and finally five textual nodes containing the words w1, w2, w3, w4 and w5.

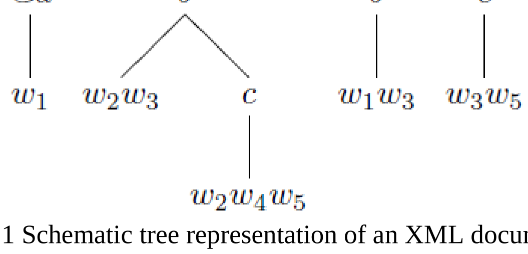

Fig. 1 Schematic tree representation of an XML document

Our structural index ---which is schematically depicted in Figure 3--- virtually indexes all the paths (see Figure 2) in the collection by using several compact data structures: the source array ***A***, the suffix array ***S***, the dictionary of tags ***D***, the dictionary of range ***D'***, the keyword list array, the ***BitsC*** binary array, the ***PosC*** array, the ***Bits Inv*** binary array and in the bottom the ternary search tree ***T***.

path	content
/a/@d	[1,1]
/a/b	[2,3], [7,8]
/a/b/c	[4,6]
/a/c	[9,10]

Fig. 2 Path index (with size N=4) for the example tree of Figure 1

The source array ***A*=(*A*$_1$,*A*$_2$,...,*A*$_M$)** stores the N paths in the collection in lexicographical order and uses a distinguished symbol (the slash character, which is not a valid XML tag) to separate the different paths. No separator is needed between consecutive tags in the same path. Every path can thus be univocally recovered with a single integer identifier: its position ***n*** in the lexicographic sequence ***/u$_1$/u$_2$/.../u$_N$*** stored in ***A***.

For the sake of clarity, every path stored in ***A*** is shown in the picture as a sequence of tags but, in practice, paths are mapped to integer sequences by creating a dictionary ***D*** of tags which is consistent with the lexicographical order. This dictionary ***D*** is enhanced with the path separator (the slash character) which is assigned the lowest integer, thus signifying that it is the first symbol in the lexicographical order.

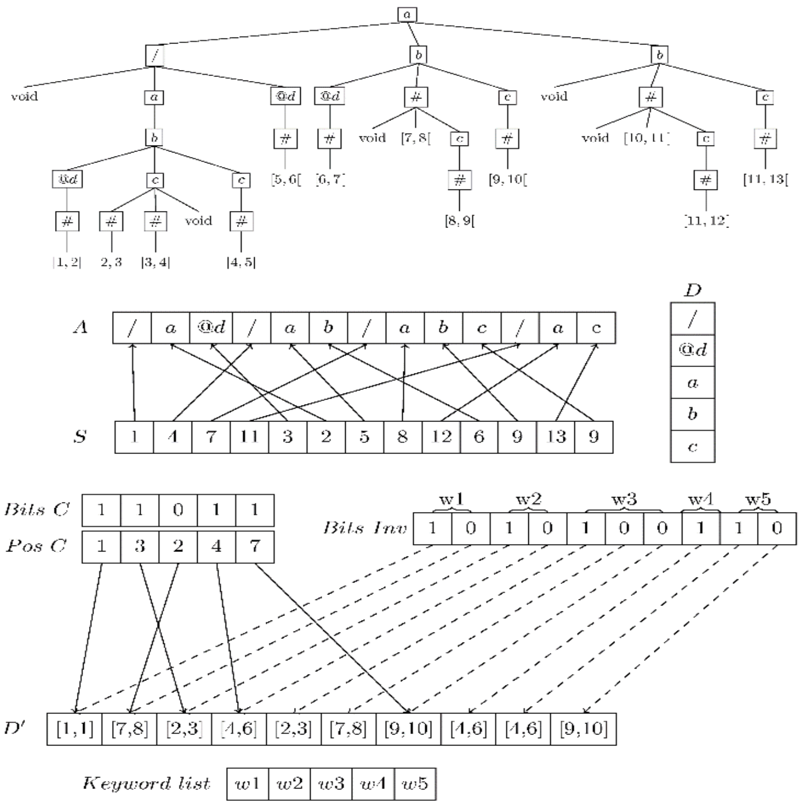

Fig. 3 Data structures integrating the index for the example tree of Fig. 1

The suffix array ***S*** is the array of integers ***(S$_1$,S$_2$,...,S$_M$)*** giving the starting positions of the ***M*** suffixes of ***A*** in lexicographical order. Note that, owing to the lexicographical ordering of paths in ***A***, there are no crossing connections between the first ***N*** elements in ***S*** and the corresponding suffixes in ***A***.

The Keyword list ***(K$_1$,K$_2$,...,K$_i$)***, ---where ***i*** is the total number of unique word in the tree--- is constructed with the textual content of the tree, in our case with the leaf nodes in Figure 1 in lexicographical order. And the dictionary ***(D'$_1$,D'$_2$,...,D'$_j$)*** ---where ***j*** matches with total number of words in the tree--- is constructed with the range positions of each word in the keyword list, and each range is the initial and final position counting each word in the tree. Simultaneously is constructed the *Bits Inv* binary array ***(b$_1$,b$_2$,...,b$_j$)***, where the size ***j*** matches with ***D'*** length . The "1" indicates the start of each word and "0" following it, the other apparitions of the same word in the tree.

And finally we implemented an auxiliary ternary search tree ***T*** which stores the (pre-computed) results of the binary search of the suffix array. In a ternary search tree, each internal node simply stores a label while the content is stored in the leaves. In order to locate the content associated with a sequence of tags, the current tag is compared with the node label. If the tag precedes (in lexicographical order) the label, the search is transferred to the left child and if it follows the node label the search is transferred to the right child. If the tag and the node label are identical, the matched tag is removed from the sequence and the search is transferred to the central child. In the case of the tag being the end of string marker (here, the hash symbol "/"), the central child contains the result associated with the input sequence.

A balanced ternary search tree is obtained by means of recursive construction: the subpath referenced by the middle position ***m=M/2*** in ***S*** is added to the tree (and the corresponding result); the procedure is then repeated for the preceding positions in ***S***, and the procedure is finally applied to the paths referenced by $\boldsymbol{S_{\{m+1\}}}$,$\boldsymbol{S_{\{m+2\}}}$,....

The construction process is detailed in Figure 4 which shows the order in which the subpaths $\boldsymbol{\sigma_k}$ have been added to the tree, the position in ***A*** where $\boldsymbol{\sigma_k}$ starts, the input for the ternary search tree (the prefix of $\boldsymbol{\sigma_k}$ terminated with the end of string marker) and the required output (the maximal range of ***k***-values such that all $\boldsymbol{\sigma_k}$ starts with the same prefix).

Finally while processing each path (during the construction of the suffix array and the ternary search tree) with their range positions as shown in Figure 2, in addition we build a ***BitsC*** binary array $\boldsymbol{(b_1,b_2,...,b_k)}$, where "1" indicates the start of each path and the "0" following it, the remaining occurrences and a PosC array$\boldsymbol{(c_1,c_2,...,c_k)}$, having reference to the first occurrence of that range in the dictionary ***D'*** (solid arrows in Figure 3). For both arrays ***k*** is the total number of path in the tree.

k	S_k	input	output
7	5	$a\ b\ \#$	[7,9[
3	7	$/\ a\ b\ c\ \#$	[3,4[
1	1	$a\ @d\ \#$	[1,2[
2	4	$a\ b\ \#$	[2,3[
5	3	$@d\ \#$	[5,6[
4	11	$/\ a\ c\ \#$	[4,5[
6	2	$a\ @d\ \#$	[6,7[
10	6	$b\ \#$	[10,12[
8	8	$a\ b\ c\ \#$	[8,9[
9	12	$a\ c\ \#$	[9,10[
12	13	$c\ \#$	[12,14[
11	9	$b\ c\ \#$	[11,10[
13	10	$c\ \#$	[12,14[

Fig. 4 Insertion steps performed during the construction of the ternary search tree

For example for structure query, the query ***//a/b*** is first translated into ***α(q)=(a,b)*** and the search in the ternary search tree ***T*** returns the range [7,9[, then is computed the path number and the paths 2 and 3 are obtained (as is shown in Fig. 2 */a/b* and */a/b/c*). For each path the operation ***Select*** is executed in the ***BitsC*** binary array, and we got the range *[2,3]* in BitsC. With this range in the ***PosC*** array that point to ***D'*** we got the final ranges *[2,3][7,8][4,6]* for the original tree.

IV. EXPERIMENTAL RESULTS AND COMPARISONS

We have implemented our prototype named ***SATRS*** based on the data structures and algorithms presented in the previous section. All code was written in Java, and all experiments were done in Pentium Core 2 Duo Processor, 3,0 GHz and 4 GB of RAM. In our experiments we used three XML datasets XMARK, DBLP and SWISSPROT which cover a lot of XML data formats and structures. In short DBLP has many repetitive structures comparing with the others datasets and XMARK has the largest maximum depth among these three datasets. The statistical data of datasets are shown in Table 1.

First of all the XML documents were parsed to extract all paths with its ranges (initial and final position) of the text contained within it. Then, a second parsing is performed to extract the common paths and create a path index (see Figure 2). At the same time were extracted all text content with its ranges (initial and final position) within the XML document.

TABLE I CHARACTERISTICS OF DATASETS

Dataset	**Size(bytes)**	**Tree depth Max/Avg**	**Leaves**	**Tags/Att(Dist)**
DBLP	133,856,133	7/3,4	7,067,935	3,735,407 (40)
SWISSPROT	114,820,211	6/3,9	8,143,919	5,166,891 (99)
XMARK	119,504,522	13/6,2	3,714,508	2,048,194 (83)

Two kind of experiments were made, for the first one were evaluated six different structural queries. Each type of query may consist on a single type parent-child (P-C) relationship, a single type ancestor-descendant (A-D) relationship or mixed types of both relationships. (See Table II) and for the final results (see Figure 5). A single backslash (/) is used to represent a parent-child edge or PC edge and a double backslash (//) is used to represent an ancestor-descendant edge or AD edge. In most cases the queries with A-D relationships are slower than the P-C queries.

TABLE II STRUCTURAL QUERIES

DBLP	
Q1	//inproceedings/booktitle
Q2	/dblp/mastersthesis/author
SWISSPROT	
Q3	/root/Entry/Ref/MedLine
Q4	//Entry/Ref//Cite
XMARK	
Q5	//item/mailbox//from
Q6	/site/regions/africa/item/location

The best results as seen in Figure 5 are shown for our proposal (**SATRS**), in all cases the response time does not exceed 25 milliseconds. First of all because the structures used (ternary search tree and suffix array) has generally very fast access and are very space-efficient. Finally the use of succinct data structures such as Rank and Select Dictionaries, improve the performance of the index, since operations on binary arrays are very efficient.

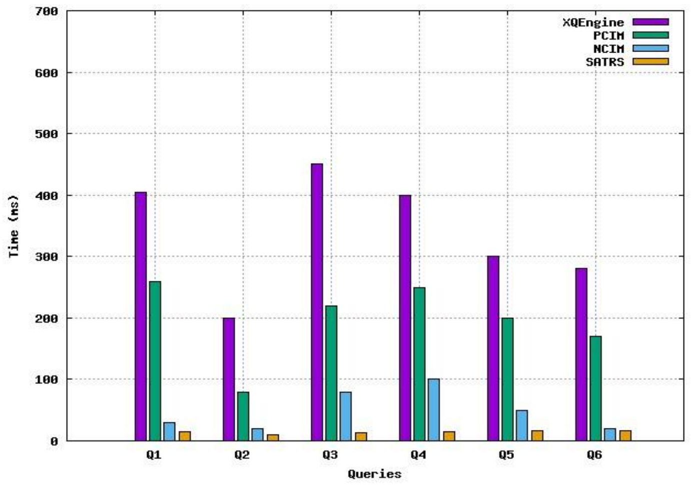

Fig. 5 Query time comparison

For the second experimental evaluation it's good to emphasize that we filter out the plain texts from each indexing method in order to measure the compression rate more accurately. In Figure 6 is shown the compression ratio used by the equation 1. XQEngine shows the worst compression ratio among the four methods and our proposal achieves the best performance, it can compresses between 95 and 99 percent for all datasets.

$$comp.ratio = \frac{uncomp\_size - comp\_size}{uncomp\_size} \quad (1)$$

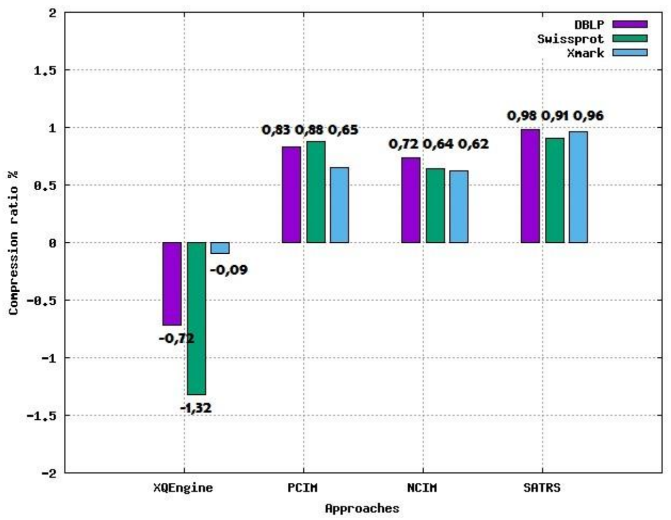

Fig. 6 Comparison of query processing time Compression ratio of different methods

V. CONCLUSIONS

One limitation of the **SATRS** is that we assume the index can fit in main memory. Although the **SATRS** has a great compression ratio, but it may not be suitable when the index size of the XML document exceeds the size of main memory, however it shows the best results for all experimental tests. Among all methods, NCIM outperforms the PCIM query time, because the PCIM stores text content in tables, whereas the NCIM stores them in the leaf node index under the corresponding tag name. For all experiments XQEngine shows the worst performance in terms of query response time and in index compression ratio. For future work, the authors would like to compare with others approaches like [14] [11] and finally we recommend the use of indexes that store the information on a secondary memory like in [20] and the use of parallel and distributed systems like [21] to solve our scalability problem.

REFERENCES

[1] Consortium, W.W.W., *XML path language (XPath) 2.0.* 2010.
[2] Boag, S., et al., *XQuery 1.0: An XML query language*. 2002.
[3] Zemmar, I., A. Benouareth, and L. Souici-Meslati, *A survey of Indexing techniques in Natives XML Databases.* 2011.
[4] Manber, U. and G. Myers, *Suffix arrays: a new method for on-line string searches.* siam Journal on Computing, 1993. **22**(5): p. 935-948.
[5] Bentley, J.L. and R. Sedgewick. *Fast algorithms for sorting and searching strings*. in *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. 1997. Society for Industrial and Applied Mathematics.
[6] Goldman, R. and J. Widom, *Dataguides: Enabling query formulation and optimization in semistructured databases.* 1997.
[7] Cooper, B. and M. Shadmon, *The index fabric: Technical overview.* RightOrder Inc, 2001.
[8] Milo, T. and D. Suciu, *Index structures for path expressions*, in *Database Theory—ICDT'99*. 1999, Springer. p. 277-295.
[9] Hsu, W., et al. *An efficient XML indexing method based on path clustering*. in *Proceedings of the 20th IASTED International Conference on Modelling and Simulation*. 2009.
[10] Liao, I.-E., W.-C. Hsu, and Y.-L. Chen, *An efficient indexing and compressing scheme for XML query processing*, in *Networked Digital Technologies*. 2010, Springer. p. 70-84.
[11] Hsu, W.-C. and I.-E. Liao, *Cis-x: A compacted indexing scheme for efficient query evaluation of xml documents.* Information Sciences, 2013. **241**: p. 195-211.
[12] Zhang, C., et al. *On supporting containment queries in relational database management systems*. in *ACM SIGMOD Record*. 2001. ACM.
[13] Al-Khalifa, S., et al. *Structural joins: A primitive for efficient XML query pattern matching*. in *Data Engineering, 2002. Proceedings. 18th International Conference on*. 2002. IEEE.
[14] Bruno, N., N. Koudas, and D. Srivastava. *Holistic twig joins: optimal XML pattern matching*. in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 2002. ACM.
[15] Lu, J., T. Chen, and T.W. Ling. *Efficient processing of XML twig patterns with parent child edges: a look-ahead approach*. in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. 2004. ACM.
[16] Wang, H., et al. *ViST: a dynamic index method for querying XML data by tree structures*. in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003. ACM.
[17] Rao, P. and B. Moon. *PRIX: Indexing and querying XML using prufer sequences*. in *Data Engineering, 2004. Proceedings. 20th International Conference on*. 2004. IEEE.
[18] Ferragina, P., et al. *Structuring labeled trees for optimal succinctness, and beyond*. in *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*. 2005. IEEE.
[19] Burrows, M. and D.J. Wheeler, *A block-sorting lossless data compression algorithm.* 1994.
[20] Işıkman, Ö.Ö., et al., *TempoXML: Nested bitemporal relationship modeling and conversion tool for fuzzy XML.* Information Sciences, 2012. **193**: p. 247-274.
[21] Dede, E., et al. *Scalable and distributed processing of scientific XML data*. in *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*. 2011. IEEE Computer Society.