Volume 3, Issue 8, August 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

An Investigation on Constructing XML Structure Tree

Baydaa Al-Hamandai
Dept. of Computer Science, Zarqa University, Jordan

Raad Alwan
Dept. of Computer Science, Philadelphia University, Jordan

Abstract*— Nowadays, several approaches are dealing with XML documents require generating the Structure Tree (XST) for these documents. These approaches could be for compressing, integrating, Ontology representation, finding similarity of XML documents. This paper investigates the proper ways to generate XST by proposing two algorithms each one depends on different parsing techniques, SAX and DOM and explains the main differences between them. Testing the memory and the time required to generate the XST shows that using SAX is faster and can save up to half the memory required using DOM.

Keywords*—XML-Tree, XML structure Tree, Simple API for XML (API), Document Object Model (DOM) Compressing XML, XML Integration.

I. INTRODUCTION

eXtensible Markup Language (XML) is considered to be the de facto of representing and transferring data over the Web. Because of its features, XML become the main building block in managing information and exchanging this information through systems, software, and platforms. Its operability and readability features make this language to be used widely in different applications.

XML Structure tree (XST) has been used in different applications and algorithms to reach different goals. Compressing XML documents was and still an active topic. Some of the proposed algorithms aimed to only compress the XML document to reduce the storage and transferring requirements [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. On the other hand, some of these compressors focus on compressing and retrieving information from the compressed version with little or no decompression. The latest approaches need to keep the structure part of the document for retrieving purposes and to do so, they need to generate the XST either indexed or normal trees [12], [13], [14], [15], [16], [17], [18], [19], [20], [21].

XML documents are generated through heterogeneous sources. The integration of these documents is a hot topic lately. There are different approaches to integrate XML documents, most of them focuses on the semantic relationship between these documents which require the generation of XST [22], [23], [24], [25], [26], [27], [28], [29].

Find the similarity between XML documents is another use of XST to compare the trees of the documents. This technique reduces the time requires to scan the required documents and find their similarity. Moreover, these techniques find not only if the documents are similar to each others, but also the ratio of the similarity [30], [31], [32], [33], [34].

The structure pare of the XML documents describes the syntax of these documents but lack the semantic relationship between their elements. Using Ontology is one of the techniques used to find the semantic relationships and use them in different applications by mapping the XML documents or their Schema to ontology classes. Most of the mapping techniques depend on the semantic tree or graph of these documents, which can be derived from XST [35], [36], [37], [38], [39], [40], [41], [42]. With all the importance of generating the XST, most of the previously mentioned references did not specify the exact algorithm used to generate the XST. In this paper we propose two algorithms to generate the XST. The first one depends on SAX parser, and the second one used DOM parser. The features of each parser alongside with the proposed algorithms, their correctness proof, and testing the algorithms are in the following sections.

II. XML PARSING

The process of parsing XML documents, which considered being the most expensive phase through the XML processing phases, passes through 3 stages [43]: (1) *char conversion*, where all the characters in the document are converted to the character code which can be understood by the host programming language used. (2) *Lexical analysis*, uses the Finite State Machine (FSM) to group the characters in the documents into W3C-known tokens, such as start-element, start-element-name, attribute-name …etc. Figure 1 shows a sample of FSM that achieve some cases in analysing the XML document. (3) *Syntactic analysis*, this stage checks if the XML document is well- formed, especially for nesting the tags, and other validation rules. Stack-like structure is used to reach this goal.

After the three parsing stages, the role of an API starts to make the well-formed XML document available to be used through a programming language. The next paragraph illustrates the main types of XML API and the differences between them.

III. DOM vs. SAX

There are two main types of APIs to parse XML documents, either memory-based; such as DOM, jDOM, and Xerces2; or streaming-based such as SAX and StAX [44]. Memory-based parsers require the whole XML document to be stored in

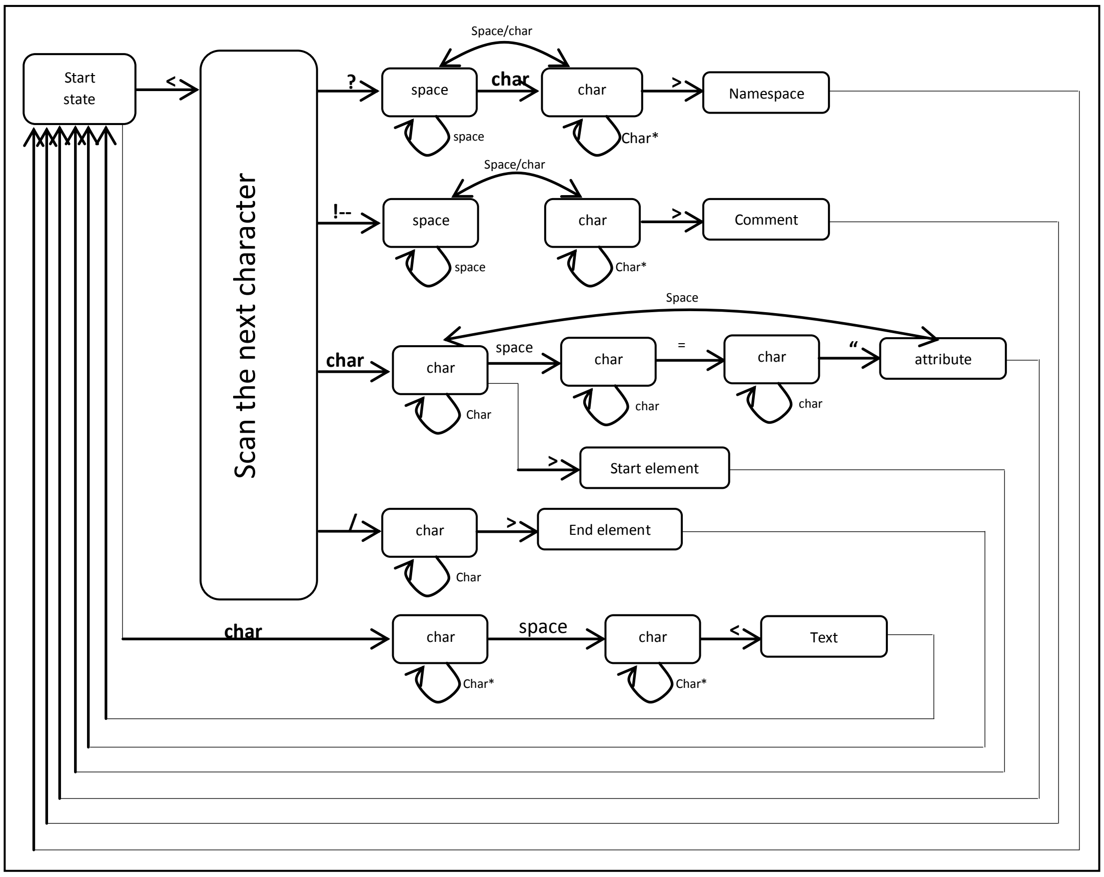


Figure 1: Sample FSM to XML lexical analyser.

the memory in tree-like structure before it can be processed. Although it suffers from a problematic memory consuming condition, the stored tree can be traversed several times back and forth making it more accessible for applications required searching these documents. Furthermore, the changes in the sequence of the nodes and the transformation of data are flexile and easy to be performed.
On the other hand, streaming-based XML parsers process these documents while they are being read without the need to store the complete document into memory. These parsers generate events according to the content of the processed document such as *start-document, start-element, end-element, and end-document* and other events. Streaming-based parsers are proved to be faster than memory-based ones; however, it is impossible to edit neither the content of the nodes or their associated data, nor the order of these nodes. [43], [44], [45]. Table I explains the main differences between DOM and SAX.

Table I: A Comparison between DOM and SAX.

Criteria	DOM	SAX
Memory used	High	Low
Access type	Random	Sequential
Created structure	Tree	None
Store whole document in memory	Yes	No
Preserve comments	Yes	No
Preserve namespace	Yes	Yes
Editing document	Yes	No

Choosing between using DOM or SAX depends on the application requirements, the available facilities (memory and speed of the used equipment), and the characteristics of the used XML documents.

IV. XST DEFINITION

Mainly, XML documents have two main parts: structure and data. The structure part syntactically organizes the data part without providing the semantic relationships between these parts. This feature makes these documents to suffer from redundancy in their structure and XML tree is a huge one even for simple XML documents.

Definition-1: *XML-Tree is a tree that has the following set of nodes* $XST = \langle E, A, D \rangle$.

$E = \langle E_r, E_c, E_s \rangle$ *is a finite set of element nodes such that* $E_r \neq \emptyset$ *is the unique element in the root node,* $E_c = \{e_{c_1}, e_{c_2}, e_{c_3}, \dots, e_{c_n}\} \mid \forall e_{c_i} : e_{c_i} \notin \{E_r, E_s\} \,\&\, n \geq 0$ *are complex elements that have sub-elements (children),* $E_s = \{e_{s_1}, e_{s_2}, e_{s_3}, \dots, e_{s_k}\} \mid \forall e_{s_i} : e_{c_i} \notin \{E_r, E_c, \emptyset\} \,\&\, k > 0$ *are simple element in the leaves of the tree (represented as ellipses).* $A = \{@a_1, @a_2, \dots, @a_j\} \mid j \geq 0$ *are the attribute nodes in the tree (represented as squares).* $D = \{d_{e_{i_1}}, d_{e_{i_2}}, \dots, d_{e_{i_m}}\} \mid e_i \in \{E_r, E_c, E_s\}, m \geq 0$.

According to *definition-1*, all elements and attributes are represented as nodes in the XML-Tree and the data under each node (if any) are attached to these nodes. This increase the tree size and it become not useful for applications and approaches that are focus on the structure only. As an example, Figure 2 shows a simple XML document taken from a huge database for published books. The sample is for 4 books only and it is clear from Figure 3, which illustrates its XML-Tree, that the tree has big ratio of redundancy in the elements and attributes names.

```
<catalog>
  <book id="bk101">
    <author>Gambardella, Matthew</author>
    <title>XML Developer's Guide</title>
    <genre>Computer</genre>
    <price>44.95</price>
    <publish_date>2000-10-01</publish_date>
    <description>An in-depth look at creating applications with XML.</description>
  </book>
  <book id="bk102">
    <author>Ralls, Kim</author>
    <title>Midnight Rain</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2000-12-16</publish_date>
    <description>A former architect battles corporate zombies, an evil sorceress, and her own childhood to become queen
of the world.</description>
  </book>
  <book id="bk105">
    <author>Corets, Eva</author>
    <title>The Sundered Grail</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2001-09-10</publish_date>
    <description>The two daughters of Maeve, half-sisters, battle one another for control of England. Sequel
             to Oberon's Legacy.</description>
  </book>
  <book id="bk110">
    <author>O'Brien, Tim</author>
    <title>Microsoft .NET: The Programming Bible</title>
    <genre>Computer</genre>
    <price>36.95</price>
    <publish_date>2000-12-09</publish_date>
    <description>Microsoft's .NET initiative is explored in detail in this deep programmer's reference.
    </description>
  </book>
</catalog>
```

Figure 2: An XML sample example.

Definition-2: *XST is a tree similar to XML-Tree with the set of paths=* P' *as follows:*

$$\forall P \in \{p_1, p_2, \dots, p_n\} \mid P_i \rightarrow \{E_r, e_{c_1}, e_{c_2}, \dots, e_{c_k}, e_s\}$$

$$p'_{m+1} \rightarrow \{p'_1, p'_2, \dots, p'_m\}\, iff\; p'_{m+1} \notin P'$$

Definition-2 illustrates the legal set of paths that should be exist in the XST. The set of paths in XST is a subset from the original XML-Tree with no duplication.

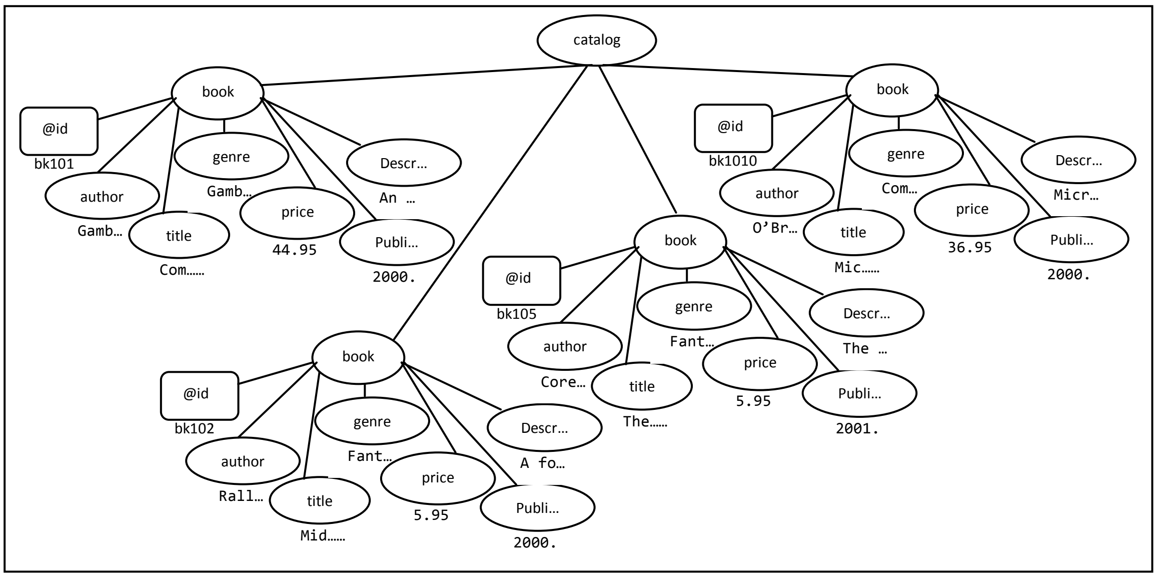


Figure 3: XML-Tree for the example in Figure 1.

V. GENERATING XST

After comparing DOM and SAX intensively, the next sections provide the algorithms that generate the XST in both APIs. Then intensive test will be used to compare the performance of the two algorithms.

A. DOM XST

As mentioned earlier, DOM API convert the whole XML document into tree-like structure and store it into the memory, the thing that takes some extra memory allocation but provides the ability to traverse the complete tree back and forth easily. Figure 4 provide the algorithm used to generate the XST using DOM API.

Generating the XST using DOM API depends on the tree-like structure constructed for the XML document. The root node of the original tree is the root node for the XST and then the complete tree nodes are visited to construct the paths from the root to each leaf and then add this path to the list of path library and constructing the XST by joining al the

```
Algorithm construct XST
  $XST = \langle\emptyset, \emptyset, \emptyset\rangle$
   $pl = \{\emptyset\}$, $i = 0$
   N=getRootNode
   $XST = \langle N, \emptyset, \emptyset\rangle$
  Step a:  $\forall n_i : n_i \in \{n.child\}$
              If ($n_i.child \neq null$)
                  $n_i \cup \{E_c\}$
                  $pl_i = pl_i + n_i$
             else
                  $n_i \cup \{E_s\}$
                     If $pl_i \notin pl$
                        $pl_i \cup \{pl\}$
   All nodes are visited?
    Yes: Construct XST such that:
       XST.root=$E_r$
       $\forall pl_i \in \{pl\}$
        $pl_{i_0}$.parent=XST.root
     No:  go to Step a
END.
```

Figure 4: Generating XST using DOM API.

paths in the path library.

B. SAX XST

In this paper, we use SAX to generate the XST. The comparison between the performance of SAX and DOM is beyond the scope of this paper. Figure 5 illustrates the algorithm used in generating XST using SAX events.

```
Function start-document (XML-doc X)
        $XST = \langle \emptyset, \emptyset, \emptyset \rangle$
        $pl = \{\emptyset\}, i = 0$
        visit($E_r$)
        $XST = \langle E_r, \emptyset, \emptyset \rangle$
        parent=$E_r$
        return $(XST, pl)$
  end start-document.
```

```
Function start-element (XML-doc X)
        visit ($e_k$)
        while $\exists$ start − element ($e_{k+1}$)
            $pl_i = pl_i + e$
            If ($e_k$) has attribute (@a)
                $pl_i = pl_i + @a$
                if $(pl_i \notin p_l)$
                    $pl_i \cup \{p_l\}$
            visit $(e_{k+1})/(e_{k+1}) \neq \emptyset$
            parent=$e_k$
            k++
  end start-element.
```

```
Function end-element(X)
        visit(e)
            if (parent= $E_r$)
                $e \cup \{E_s\}$
                $pl_i = pl_i + e$
                if $(pl_i \notin p_l)$
                    $pl_i \cup \{p_l\}$
            else
                $e \cup \{E_c\}$
end end-element.
```

```
Function end-document(X)
        Construct the XST such that
        XST.root=$E_r$
        $\forall\ pl_i \in \{pl\}$
            $pl_{i_0}$.parent=XST.root
end end-document.
```

Figure 5: Generating XST algorithms.

The algorithm has 4 main functions:

- *start-document*: this function generates the structure of the XST and creates a path library (pl). This library holds every path that will be generated through the other functions in the algorithm. The first node in the document is the root and it is added to the XST as E_r.
- *start-element*: in this function all consequences start-elements are visited and added to the same path, since they all belong to the same path from the root. In the case of attribute existing, the attribute is considered to be a leaf and a complete path is generated. This path should be checked for existence in (pl) and added to it if it is not exist.
- *end-element:* the existing of an end element represent *either* an (E_s), if the parent of this node is the root node. In this case, another complete path is generated and it should be checked for existence against (pl).
- *end-document:* at the end of the document, all the constructed paths are joined together with their root node E_r.

Figure 6 shows the implementation of the XST generation algorithm on the XML document sample in Figure 2. It is clear that the number of node and edges in the XST is much less than XML-Tree. As the XML documents are generated from heterogeneous resources, the structures of these documents are varied from well structured to bad structured documents. As well structured the XML document is, as the smaller XST is.

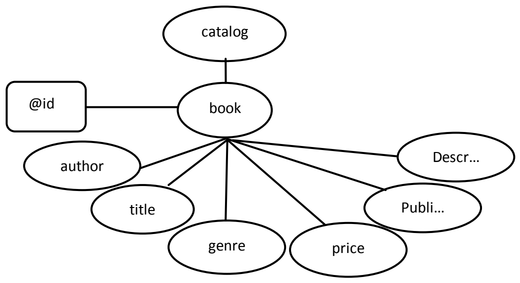
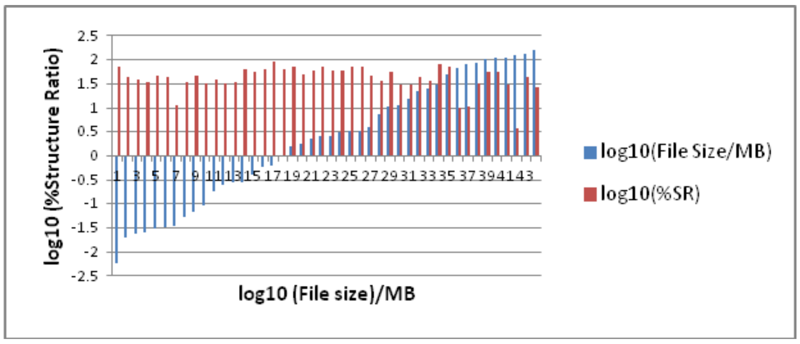


Figure 6: XST for the sample example in Figure 1.

VI. Testing

To test the proposed algorithms, a corpus of XML documents has been used. The corpus organized into 14 groups, according to the source of the documents, and each one has several XML documents with an overall size of more than 1GB.

The first test focuses on illustrating the difference between the size of the XML-Tree and its corresponding XST. It is useful especially in the approaches that need the XST for compression purposes. It is obviously clear from Figure 7 that there is no any relationship between the size of the document and structure ratio of that document, since some XML documents are structure based (with structure greater than 70%) and others are data based (with structure less than 30%). The XML corpus has files from both types.



While there is no relation between the whole size and the structure ratio of XML documents, there is a close relationship between the size of the XML-Tree and the size of its corresponding XST as shown in Figure 8.

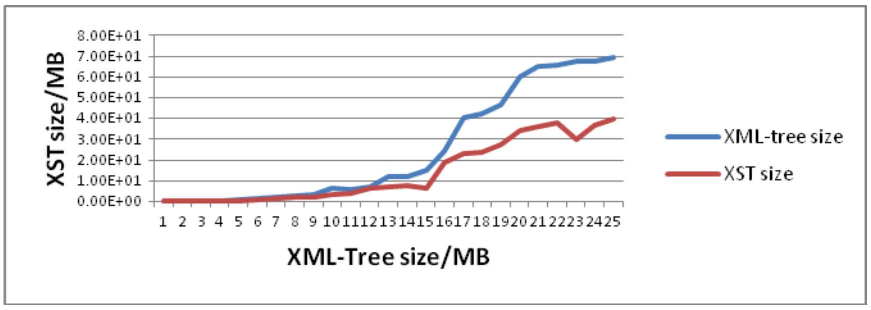


Figure 8: the relation between the XML-Tree and XST sizes.

From the test above, the size of the XST is much smaller than the XML-Tree except in some cases when the redundancy of the XML structure is little. On average the ratio between the XML-Tree and XST is 3:1.

Another test is to compare the time required to generate the XST using DOM and SAX APIs. testing the time required to generate the XST and configure the relationship between the required time and the size of the XML document. Figure 9 gives a clear picture about the relation between the size of the document and the time required to generate the XST. In the cases when the size of the XML document is small, there is no big difference between the time of DOM and SAX, while this difference is significantly high, could reach to double, when dealing with large XML documents.

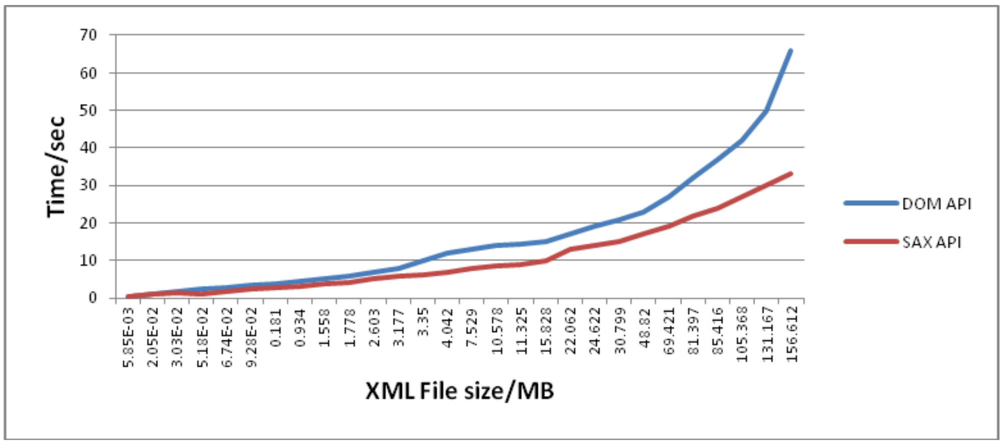


Figure 9: The time required to generate the XST.

The last test is to check the memory required to store the original XML document and to generate its XST. Figure 10 shows the differences between the required memory. It shows that using SAX requires much less memory than DOM since DOM uploads the complest XML document into memory, while SAX doesn't.

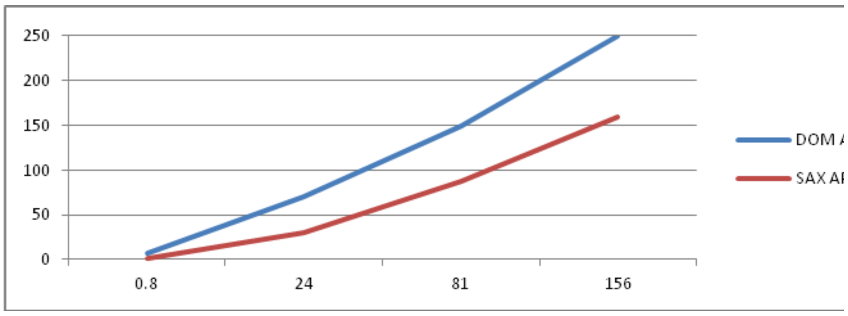


Figure 10: Memory required generating XST.

VII. CONCLUSION & FUTURE WORK

Several approaches benefit from generating the XST from the existing XML document and it is necessary to provide a clear fast algorithm to generate this type of trees. This paper proposes two algorithms one uses DOM and the other uses SAM APIs. The results of the tests show that using SAX API requires less memory and with less execution time, although there are lots of advantages of using DOM over SAX.

The two algorithms need to be developed to generate the XML Structure Graph (XSG) which provides a better way to represent the structure part of the XML document and it has been used in different approaches.

REFERENCES

[1] J Cleary and I Witten. Data Compression Using Adaptive Coding and Partial String Matching. Communications, IEEE Transactions 1984; 32: 396 - 402.
[2] D Salomon. Data Compression: The Complete Reference. ed.: Springer, 2007.
[3] P Skibinski, S Grabowski and J Swacha. Effective Asymmertic XML Compression. In: Conference Name; 2007 of Conference, Conference Location.
[4] L Zuopeng, H Kongfa, Y Ning and D Yisheng. An efficient index structure for XML based on generalized suffix tree. Information Systems 2007; 32: 283-294.
[5] G Busatto, M Lohrey and S Maneth. Efficient memory representation of XML document trees. Information Systems 2008; 33: 456-474.
[6] M.Manikandan, K B Bagan and T.Prathiba. Images and Integer Compression Using Bit Based Cascade Coding. GVIP Journal 2006; Volume 6.
[7] J G Wolff. Medical diagnosis as pattern recognition in a framework of information compression by multiple alignment, unification and search. Decision Support Systems 2006; 42: 608-625.
[8] B Al-Hamadani, J Lu and R F Alwan. A new Schema-Independent XML Compression Technique. Accepted for publication in the International Journal of Information Retrieval Research 2011.
[9] C League and K Eng. Schema-Based Compression of XML Data with Relax NG. In: Conference Name; 2007 of Conference, Conference Location.
[10] R Lawrence. The space efficiency of XML. Information and Software Technology 2004; 46: 753-759.
[11] J Adiego, G Navarro and P d l Fuente. Using Structural Context to Compress Semistructured Text Collections. Information Processing and Management 2007; Vol 43: Pages 679.
[12] D Arroyuelo, F Claude, S Maneth, V M¨akinen, G Navarro, K Nguyen, J Sir´en and N V¨alim¨aki. Fast In-Memory XPath Search using Compressed Indexes. In: In Proceedings of the IEEE Twenty-Sixth International Conference on Data Engineering (ICDE 2010); 2010; California, USA.
[13] P Buneman, M Grohe and C Koch. Path Queries on Compressed XML. in: F. Johann-Christoph, L. Peter, A. Serge, C. Michael, S. Patricia and H. Andreas, editors. Proceedings 2003 VLDB Conference. San Francisco: Morgan Kaufmann, 2003. pp. 141-152.
[14] B Qian, H Wang, J Li, H Gao, Z Bao, Y Gao, Y Gu, L Guo, Y Li, J Lu, et al. Path-Based XML Stream Compression with XPath Query Support Web-Age Information Management. Springer Berlin / Heidelberg, 2012. pp. 329-339.
[15] R K Wong, F Lam and W M Shui. Querying and maintaining a compact XML storage. In: Conference Name; 2007 of Conference, Conference Location: ACM. 1073-1082.
[16] Y Lin, Y Zhang, Q Li and J Yang. Supporting Efficient Query Processing on Compressed XML Files. In: Conference Name; 2005 of Conference, Conference Location: ACM.
[17] P M Tolani and J R Haritsa. XGRIND: A Query-friendly XML Compressor. In: Conference Name; 2000 of Conference, Conference Location.

[18] F Norbert and G Kai. XIRQL: An XML query language based on information retrieval concepts. ACM Trans. Inf. Syst. 2004; 22: 313-356.

[19] J-K Min, M-J Park and C-W Chung. XPRESS: a queriable compression for XML data. In: Conference Name; 2003 of Conference, Conference Location: ACM. 122-133.

[20] J Cheng and W NG. XQZip: Querying Compressed XML using Structural Indexing. In: International Conference on Extending Data Base Technology (EDBT); 2004.

[21] T Müldner, C Fry, J K Miziołek and S Durno. XSAQCT: XML Queryable Compressor. In: Balisage: The Markup Conference 2009; 2009.

[22] N Bikakis, N Gioldasis, C Tsinaraki and S Christodoulakis. Semantic Based Access over XML Data. Visioning and Engineering the Knowledge Society. A Web Science Perspective. Springer Berlin Heidelberg, 2009. pp. 259-267.

[23] Y Cong and P Lucian. Constraint-based XML query rewriting for data integration. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data; 2004; Paris, France: ACM.

[24] R García and Ò Celma. Semantic Integration and Retrieval of Multimedia Metadata. In: In 5th International Workshop on Knowledge Markup and Semantic Annotation (SemAnnot 2005); 2005.

[25] R Kienast and C Baumgartner. Semantic Data Integration on Biomedical Data Using Semantic Web Technologies. in: M. A. Mahdavi, editor. Bioinformatics: trends and Methodologies. InTech, 2011.

[26] R Meersman, Z Tari, W Viyanon, S Madria and S Bhowmick. XML Data Integration Based on Content and Structure Similarity Using Keys. On the Move to Meaningful Internet Systems: OTM 2008. Springer Berlin Heidelberg, 2008. pp. 484-493.

[27] W Papers. The Next Generation of Data Integration for Data Warehousing. In: Conference Name; 2011 of Conference, Conference Location: Kalido.

[28] P RodrÃŒguez-Gianolli and J Mylopoulos. A Semantic Approach to XML-based Data Integration. Conceptual Modeling Springer Berlin Heidelberg, 2001. pp. 117-132.

[29] Z Zhang, V B Bajic, J Yu2, K-H Cheung and J P Townsend. Data Integration in Bioinformatics: Current Efforts and Challenges. in: A. M. Mahdavi, editor. Bioinformatics: Trends and Methodologies. InTech, 2011.

[30] T Dalamagas, T Cheng, K-J Winkel and T Sellis. A methodology for clustering XML documents by structure. Information Systems 2006; 31: 187-228.

[31] Y Li, D L Olson and Z Qin. Similarity measures between intuitionistic fuzzy (vague) sets: A comparative analysis. Pattern Recognition Letters 2007; 28: 278-285.

[32] T Lindholm, J Kangasharju and S Tarkoma. Fast and simple XML tree differencing by sequence alignment. In: Conference Name; 2006 of Conference, Conference Location: ACM. 75-84.

[33] R Nayak and W Iryadi. XML schema clustering with semantic and hierarchical similarity measures. Knowledge-Based Systems 2007; 20: 336-349.

[34] T Schlieder. Similarity Search in XML Data using Cost-Based Query Transformations. In: In Proceeding of ACM SIGMOD WebDB; 2001. pp. 19-24.

[35] N M Yahia and A Sahar A.; Ahmed. Automatic Generation of OWL Ontology from XML Data Source. International Journal of Computer Science Issues (IJCSI) 2012; 9: 77.

[36] D Schober, M Boeker, J Bullenkamp, C Huszka, K Depraetere, D Teodoro, N Nadah, R Choquet, C Daniel and S Schulz. The DebugIT core ontology: semantic integration of antibiotics resistance patterns. Stud Health Technol Inform 2009; 160: 1060-1064.

[37] S Yang, J Wu, A He and Y Rao. Derivation of OWL Ontology from XML Documents by Formal Semantic Modeling. ed., 2013.

[38] J Wang, J Lu, Y Zhang, Z Miao and B Zhou. Integrating Heterogeneous Data Source Using Ontology. JOURNAL OF SOFTWARE 2009; 4.

[39] I F Cruz, X Huiyong and H Feihong. An ontology-based framework for XML semantic integration. In: Database Engineering and Applications Symposium, 2004. IDEAS '04. Proceedings. International; 2004. 217-226.

[40] B Amann, C Beeri, I Fundulaki and M Scholl. Ontology-Based Integration of XML Web Resources. The Semantic Web â€" ISWC 2002. Springer Berlin Heidelberg, 2002. pp. 117-131.

[41] P Thuy, Y-K Lee and S Lee. A Semantic Approach for Transforming XML Data into RDF Ontology. Wireless Personal Communications 2013: 1-16.

[42] P T T Thuy, Y-K Lee and S Lee. S-Trans: Semantic transformation of XML healthcare data into OWL ontology. Knowledge-Based Systems 2012; 35: 349-356.

[43] L Tak, J J Ding and J C Liu. XML Document Parsing: Operational and Performance Characteristics. Computer 2008; 41: 30-37.

[44] B Oliveira, V Santos and O Belo. Processing XML with Java – A Performance Benchmark. International Journal of New Computer Architectures and their Applications (IJNCAA) 2013; 3: 72-85.

[45] V M Deshmukh and G R Bamnote. An Empirical Study: XML Parsing using Various Data Structures. International Journal Of Computer Science And Applications 2013; 6: 400- 405.