Journal of Statistics, Computing and Interdisciplinary Research
ISSN (P): 2707-7101
J. Stat. Comp. & Interdis. Rese.
Volume 6(1) (2024), 75-89
http://scir.wum.edu.pk/index.php/ojs

Enhancing XML Data Parsing and Querying Performance on Multi-Core Architectures

Muhammad Ali, Minhaj Ahmad Khan

Department of Computer Science, Bahauddin Zakariya University, Multan 60000, Pakistan
Corresponding Author: Minhaj Ahmad Khan, mik@bzu.edu.pk

Abstract

With the increasing number of computer applications, the eXtensible Markup Language (XML) has become ubiquitous and is being utilized broadly on the web for exchanging and storing information. XML files of large size take more space because of redundancy, and when parsed on machines degrade the execution. This paper proposes a novel approach aimed at enhancing the performance of parsing XML files. The proposed approach initially divides the XML file into several parts concurrently and then processes it to ensure well-formedness. Each part is subsequently encoded to save memory space, efficient parsing, and querying. Multiple parse trees are then generated in parallel in the next phase, prior to querying required data from the parse trees. The parallel parsing and efficient querying make our approach perform better than other parsing approaches. The results obtained on processing XML files of diverse sizes show an increase in performance by 22.50%, 19.41%, and 30.25% over the well-known DOM, SAX, and StAX parsing approaches, respectively.

Key Words: *XML, parsing, querying, multi-core, encoding, well-formedness*

1. Introduction:

eXtensible Markup Language (XML) uses a textual format for representation of data that is used widely for many web applications (Braganholo & Mattoso, 2014; Eckstein & Casabianca, 2001; Fawcett et al., 2012). It has now become a de facto standard for storing information and exchanging it among diverse applications on the internet (Lv & Yan, 2013; *XML Tutorial*). It has become prevalent through its wide usage in small as well as large organizations such as those for healthcare, sports, banks, business reporting, etc.

As the data of organizations increases, the size of XML files also increases because of redundant tags. When these large XML files get parsed on a computer system, it increases the processing time and finally degrades the performance of the system (Parameswaran et al., 2013; Tang et al., 2013). On a multi-core system, the efficiency is severely affected due to serial parsing of XML files and subsequent querying to obtain data from the files. Consequently, a parallel parsing and querying model could improve performance in terms of execution time along with effective resource usage (Ahmad et al., 2018; Andrei, 2009; Camacho-Rodríguez et al., 2015; Liu & Yan, 2016; Qin et al., 2017; Wu et al., 2008).

To cope with the issue of performance degradation of parsing and querying large XML files, this paper suggests a Parallel XML Parsing and Querying (PXPQ) approach that initially divides the XML file into n parts while ensuring well-formedness. The next phase creates parse tree (subtree) of each part in parallel while encoding the nodes. The querying for data search results in better performance due to compact data representation and search from the relevant subtree. Moreover, parallel parsing and querying in our approach enables us to outperform other approaches in terms of overall XML parsing and querying time.

The remaining part of this paper is organized as follows: Section 2 describes related work in terms of XML parsing techniques, compression and Binary XML format. The suggested algorithm PXPQ is described in Section 3. The experimental setup and the results obtained after evaluation of different parsing approaches are given in Section 4. The last section discusses the conclusion and future work.

2. Related Work:

XML parsing transforms serialized string data into hierarchical form. The data can then be queried efficiently for use by various applications. For parsing XML documents, different serial and parallel parsing models have been implemented in the recent past. Linghua et al. (Beck et al., 2021) present a four-phase parallel method for identifying, observing, and visualizing parallel patterns in XML texts. Zhang introduced VTD-XML (XimpleWare, 2015; Zhang, 2006), a novel non-extracted XML parsing technique. By analyzing the XML document tree, VTD-XML generated a one-dimensional array that could reduce the amount of memory required. Another parallel parsing algorithm is proposed by Lu et al. (Lu et al., 2006), comprising two main processes called pre-parsing and full-parsing. Because serial processing leads to the contemplation of more optimal designs, the pre-paring stage takes a

long time. you et al. (You & Wang, 2011) proposed another XML parsing optimization technique that does not necessitate a pre-parsing stage for parallel parsing.

The Document Object Model (DOM) is officially proposed by the World Wide Web Consortium (W3C) to define an interface that empowers different tools to update and access the style, contents, and structure of XML documents (*DOM, W3C Recommendation of Document Object Model*; Yang et al., 2015). When XML files are parsed using a DOM parser, it returns all the components of file in the form of a tree structure (Fraigniaud & Korman, 2016). The DOM facilitates to examine and explore the contents and structure of XML documents easily with a variety of functions. The DOM parser provides a common interface such as Node, Element, Attr, Text, and Document for manipulating the structure of a document.

The Simple API for XML (SAX) is an XML parser based on events. Unlike the DOM parser, a SAX parser does not make a parse tree (Megginson, 2004). SAX provides a streaming interface for parsing XML, which means that the SAX parser starts scanning from the start of the document to closing the ROOT element in a well-formed manner. After identifying tokens, it calls the appropriate event handler to process that token in the same sequence in which they appear. Processing of very large tree will be easy with less memory which is an important benefit of using SAX parser. The major drawback of the SAX parser is that it does not provide random access because it works in a forward-only manner.

The Java Document Object Model (JDOM) is a Java-based open-source library to parse XML documents (*JDOM*). It is an optimized API to provide an easy approach. JDOM works with SAX APIs and DOM and consolidates the best of both. JDOM gives a lot of methods for exploring the structure and contents of an XML document. It uses low memory and is nearly as fast as SAX. JDOM parser is used to know about the structure of an XML document intensively, traverse XML documents, and require information repeatedly. Document, Element, Attribute, Text, and Comment are the common classes defined by JDOM.

StAX is an API, based on Java for parsing XML documents. It uses almost the same technique as the SAX parser. However, there are two noteworthy distinctions between the two APIs. Firstly, StAX works as PULL API, whereas SAX works as PUSH API. In StAX, a client application gets data from XML whenever it wants. But in the SAX parser, a client application gets data when it is notified by the SAX parser. The other distinction is that StAX API can

provide both read and write facilities for XML documents. On the other hand, SAX API can only read an XML file.

To extract information from XML documents, XPath is recommended by the World Wide Web Consortium (W3C) (Moussalli et al., 2014). It traverses all the attributes, elements, text, comments, processing instructions, namespace, and document nodes of an XML document. Similarly, DOM4J is also an API based on Java for parsing XML documents. It is an extremely flexible and optimized API. It uses Java collections like Array and List. DOM4J works with Xpath, SAX, XSLT, and DOM. Normally, it is used to parse an XML document of large size and takes less memory.

In the past years, a lot of work has been done on XML compression (Cao et al., 2013) and decompression techniques such as Dense Codes compressors (Brisaboa et al., 2007), Huffman compression (Huffman, 1952; Silva de Moura et al., 2000), PPM-based methods (Cleary & Witten, 1984), Ziv-Lempel techniques (Welch, 1984; Ziv & Lempel, 1977, 1978), etc. Non-quarriable compressors have also been developed to achieve maximum compression ratio such as AXECHOP (Leighton et al., 2005), SCM (Adiego & Navarro, 2007), Exalt (Toman, 2003), Millau (Girardot & Sundaresan, 2000), XAUST (Subramanian & Shankar, 2005), XComp (Li, 2003), XMill (Liefke & Suciu, 2000), etc.

The compressors that can be queried, have been developed to enhance query processes such as XGrind (Tolani & Haritsa, 2002), XPRESS (Min et al., 2003), XCQ (Ng et al., 2006), XQzip (Cheng & Ng, 2004), and XQueC (Arion et al., 2007). Despite compressed formats, their performance degrades due to serial processing of data, especially on a multi-core processor. Similarly, the "Binary XML" takes much fewer bits for storing the same data. It also improves parsing time and reduces transmission time(Schneider J, 2014). Its variations have also been proposed in the literature, however, these parsing techniques do not incorporate a parallel mechanism for efficient parsing and querying, as implemented in the PXPQ algorithm suggested in this paper.

3 Parallel XML Parsing and Querying (PXPQ):

Despite the parallelism provided by multi-core architectures, the efficiency is limited due to sequential or serial computations in the applications. For efficient parsing and querying, the XML contents must be divided and compressed to form encoded subtrees. The Parallel XML Parsing and Querying (PXPQ) algorithm proposed in this paper uses this concept to improve

the performance of XML parsing and querying. It efficiently utilizes the parallelism offered by modern multi-core architectures. The mathematical formulation used by the PXPQ algorithm is given below.

3.1 Mathematical Formulation:

Let the contents of the XML file be represented by string χ. Let m=$|\chi|$ represent the number of characters. We consider parsing environment with a set of cores P, such that $P = \{P_k : k = 1,2,\ldots\ldots,n\}$, and $n = |P|$. For n cores, we have, Q = m/n and R = m mod n. The string χ is divided into n parts, α_i, such that $|\alpha_i| = Q$, $\forall\, i = 1,2,\ldots,n-1$ and $|\alpha_n| = Q + R$.

Let W_i $\forall\, i = 1,2,\ldots,n$ be transformations that take substrings and produce corresponding wellformed substrings α_{ui}, using the following equations:

$W_1: (\alpha_1, \alpha_2) \rightarrow \alpha_{u1}$ *eq. (1)*

$W_n: (\alpha_{n-1}, \alpha_n) \rightarrow \alpha_{un}$ *eq. (2)*

$W_k: (\alpha_{k-1}, \alpha_k, \alpha_{k+1}) \rightarrow \alpha_{uk}$ *eq. (3)* $\quad \forall\, k = 2,3,\ldots,n-1$

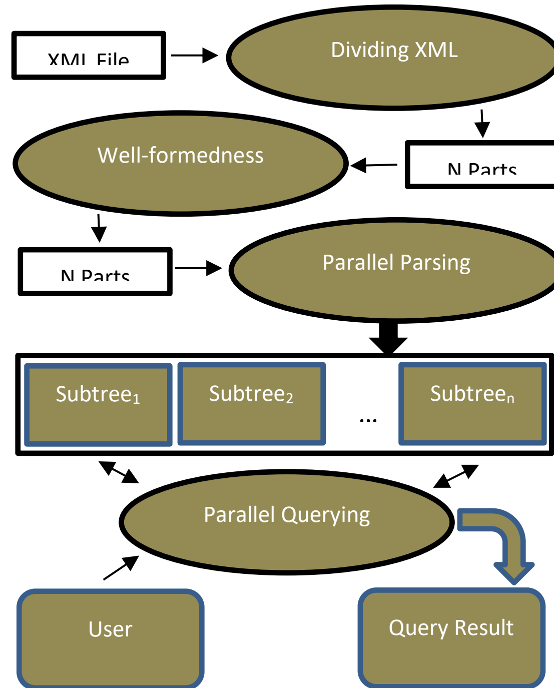
The transformations ensure that each substring can be used to generate valid parse trees. Each substring α_{ui} is encoded using function E to produce a subtree T_i as below:

$T_i = \{E(\alpha_{ui})\}, \quad \forall\, i = 1,2,\ldots,n$

The encoded subtree T_i can then be parsed and queried independently.

With parallel parsing on multi-core processors and encoded nodes, the proposed approach can perform efficiently while saving the memory space required to store XML data. The main steps performed by the PXPQ algorithm are given in Figure 1.

Fig. 1: The Main Steps of the PXPQ Algorithm.



3.2 The PXPQ Algorithm:

The PXPQ (Parallel XML Parsing and Querying) algorithm uses a top-down parsing approach that takes an XML file as input and generates an encoded parse tree that can be queried to return the result. It makes use of the ***TokenGenerator***, ***TreeGenerator***, and ***QuerySubtrees*** modules, as described below.

PXPQ (XML File):

1) **//Let** ofs be the offset and **n** be the number of cores
2) **//Let** CoreID represent the Core Id number
3) **//Let** χ be the XML file as a string
4) **//Let** S_i be used for substring // i € (Chang & Lin, 2011 …, n)
5) **//Let** TK [] be the array of tokens
6) **//Let** RN_TAG, S_TAG and F_TAG be the Root Node, Structure and Field tags, respectively, as obtained from DTD
7) **//Let** TEXT be the Text Contents from χ
8) //Divide XML into n Substrings

9) m ← | χ | // number of characters
10) ofs ← m/n
11) S_i ← substring of χ (ofs *(i-1), ofs *(i)) $\forall, i = 1,2,\ldots,n-1$
12) S_n ← substring of χ (ofs *(n-1), m)
13) // Create Wellformed Substrings
14) Compute α_{u1} for substring S_1 using Equation (1)
15) Compute α_{uk} for substring S_k using Equation (3) $\forall, k = 2,3,..,n-1$
16) Compute α_{un} for substring S_n using Equation (2)
17) //Create Tokens and Encoded Parsed Tree
18) **IF** coreID is 1
19) TK←**TokenGenerator** (α_{u1})
20) TreeGenerator (TK)
21) **ELSE-IF** coreID is n
22) TK←**TokenGenerator** (α_{un})
23) TreeGenerator (TK)
24) **ELSE**
25) TK←**TokenGenerator** (α_{uk})
26) TreeGenerator (TK)
27) **END IF**
28) Get Required Data Index **r** from user
29) M = **QuerySubtrees**(**r**)
30) Return M

END PXPQ ()

In PXPQ, steps 1 to 7 of the algorithm declare necessary variables. Step 8 stores the number of characters in ***m***. The offset based on the available number of cores is calculated in step 9, whereas step 10 creates n-1 substrings which are equal in length on the base of offset. Step 11 creates n^{th} substring. Steps 12 to 14 compute well-formed substrings $\alpha_{u1}, \alpha_{uk}, \alpha_{un}$ by transformation W. Step 15 creates tokens and produces encoded parse trees by calling ***TokenGenerator*** and ***TreeGenerator*** modules. Steps 16-18 are used to obtain user data for query/search and return the required data.

3.2.1 TokenGenerator:

TokenGenerator (S_i)
  FOR *j ←1 to length of S_i* ***DO***
    Create Token
    Add Token in Array TK
  END FOR
  Return *Array TK*
End TokenGenerator()

The ***TokenGenerator*** module of PXPQ receives each chunk of file on a separate core. These chunks are produced as a result when an XML file is fed into PXPQ. The ***TokenGenerator*** module reads data character by character on each core and produces tokens such as root node, structural tags, field tags, and text. It then returns an array of tokens to PXPQ.

3.2.2 TreeGenerator:

TreeGenerator(TK[])
//Let each node contain index and data
//Let Z represent encoded node
  FOR *t ←1 to length of TK* ***DO***
    IF *TK[t] is RN_TAG // If token is Root Node*
      Create Node using RN_TAG as Parent
    ELSE IF *TK[t] is F_TAG // If token is Field Tag*
      Create Node using S_TAG as Parent, F_TAG as Child
    ELSE IF *TK[t] is S_TAG // If token is Structure Tag*
      Create Node using RN_TAG as Parent, S_TAG as Child
    ELSE IF *TK[t] is TEXT // If a token is Content*
      Create Node using F_TAG as Parent, TK[t] as Child
    END IF
    //Encode node using hash function E
    Z=E(Node)
  END FOR
END TreeGenerator()

The ***TreeGenerator*** module is called PXPQ to create encoded parse trees. It creates nodes with appropriate hierarchy corresponding to tags of XML file. The nodes added to subtrees are then encoded for efficient traversal and querying.

3.2.3 Querying the Subtrees

QuerySubtrees(r)

//Let r be the index of the node whose data is to be searched

1. *Compute k=r/n +1*
2. *Traverse subtree k to search for node with index r*
3. ***IF** found **THEN***

   ***Return** decoded node data*

   ELSE

   ***Return** Null*

   END IF

END QuerySubtrees()

The ***QuerySubtrees*** module takes the index ***r*** whose data is to be searched. It then finds the subtree ***k*** which may contain the required data. If any node with the index ***r*** exists in the subtree, the required data of the node is returned after decoding, otherwise, the ***Null*** value is returned.

4. Experimentation: Setup and Results:

For experimentation, we have used three different architectures A1, A2 and A3, as shown in Table 1. The files of sizes 1MB, 2MB, 3MB, 4MB, and 5MB are used on architectures A1 and A2. For architecture A3 (having more memory than other architectures), we have used files of sizes 5MB, 10MB, 15MB, 20MB, and 25MB. The results obtained after executing the PXPQ algorithm in comparison with the DOM, SAX, and StAX parsers are given in this section.

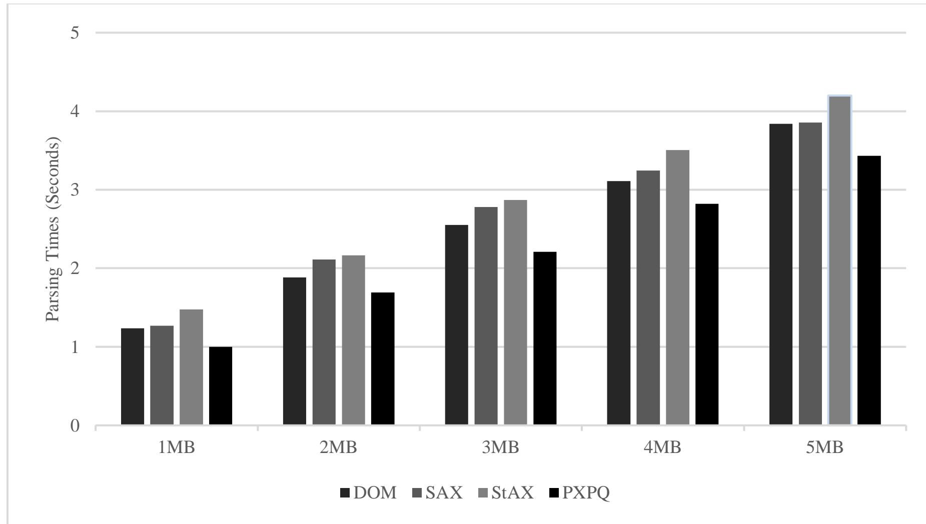
Table 1: Architectures used for Experiments.

Architecture	Processor	RAM	OS
A1	Intel Core i3, 2.40 GHz, 64-bit	4GB	Ubuntu 16.04
A2	Intel Core i7, 3.40 GHz, 64-bit	16GB	Ubuntu 18.04.4
A3	Intel Xeon Server E5 2.4 GHz	64GB	Ubuntu 16.04

4.1.Performance Results on Architecture A1:

The performance results for files of sizes 1MB, 2MB, 3MB, 4MB and 5MB are given in Figure 2. As the figure shows, the PXPQ approach performs better than the DOM, SAX and StAX approaches on architecture A1. Overall, there is an average improvement of 11.50%, 15.84% and 21.47% over DOM, SAX and StAX parsers, as given in Table 2.

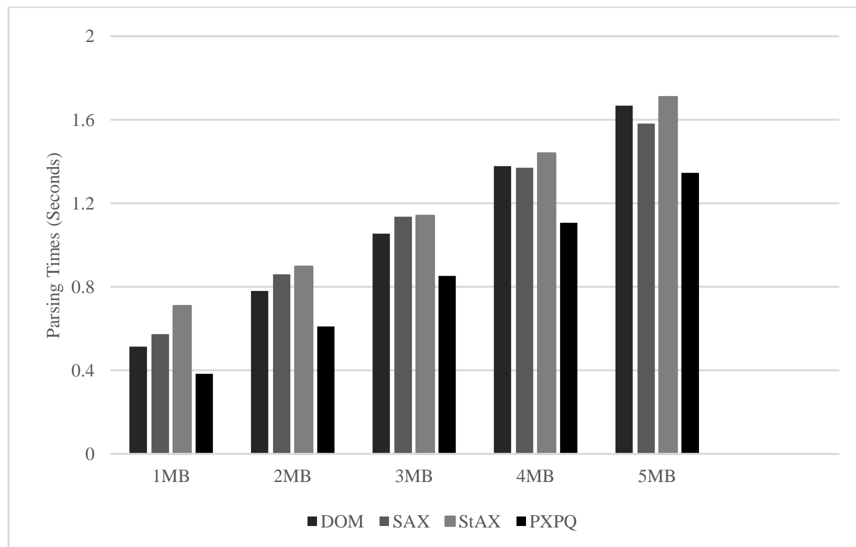
Fig. 2: Performance results on architecture A1



4.2.Performance Results on Architecture A2:

The performance results for files of sizes 1MB, 2MB, 3MB, 4MB, and 5MB are given in Figure 3. As the figure shows, the PXPQ approach performs better than the DOM, SAX, and StAX approaches on architecture A2. Overall, there is an average improvement of 20.28%, 22.10%, and 27.05% over the DOM, SAX, and StAX parsers.

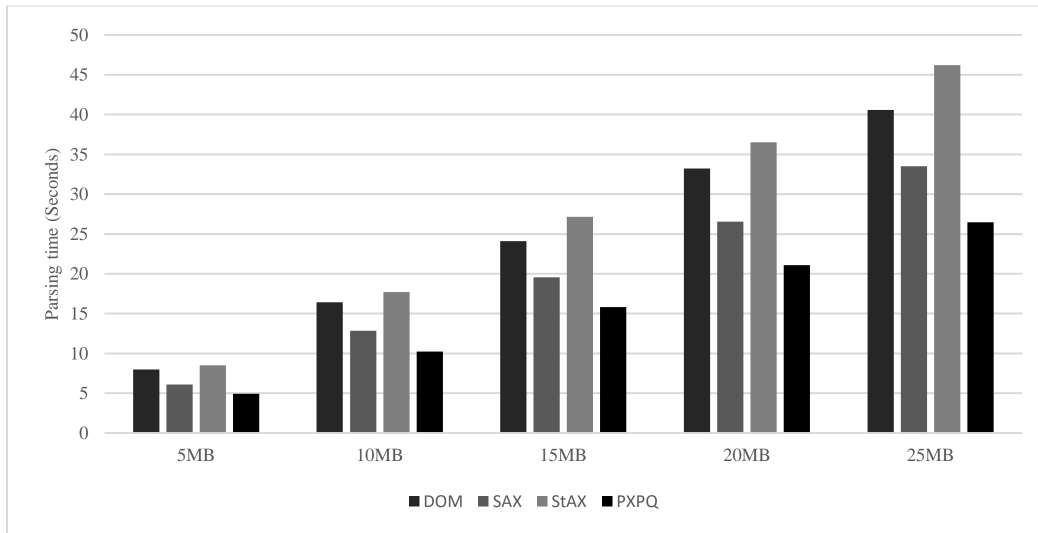
Fig. 3: Performance results on architecture A2



4.3.Performance Results on Architecture A3:

The performance results for files of sizes 5MB, 10MB, 15MB, 20MB, and 25MB are given in Figure 4.

Fig. 4: Performance results on architecture A3



As the figure shows, the PXPQ approach performs better than the DOM, SAX, and StAX approaches on architecture A3 as well. Overall, there is an average improvement of 35.74%, 20.29%, and 42.25% over the DOM, SAX, and StAX parsing approaches, as given in Table 2.

Table 2: Parsing Performance Improvement Achieved by PXPQ over DOM, SAX and StAX Parsers

	DOM (%)	SAX (%)	StAX (%)
A1	11.50	15.84	21.47
A2	20.28	22.10	27.05
A3	35.74	20.29	42.25
Average	**22.50**	**19.41**	**30.25**

4.4.Performance Summary:

Considering all the architectures A1, A2, and A3, the overall average performance improvement achieved by PXPQ is 22.50%, 19.41%, and 30.25%, over the DOM, SAX and StAX parsers, as shown in Table 2.

5. Conclusion and Future Work:

In this paper, we have proposed the parallel XML parsing and querying (PXPQ) algorithm to enhance the performance of XML data on multi-core architectures. The PXPQ algorithm exploits the parallelism of modern architectures by dividing XML files while maintaining well-formedness. The contents are then encoded for efficient parallel parsing and querying. We have used three diverse architectures for parsing files of sizes 1MB, 2MB, 3MB, 4MB, and 5MB. The results obtained show that the proposed PXPQ algorithm can outperform the well-known DOM, SAX, and StAX parsing approaches by 22.50%, 19.41%, and 30.25%, respectively. As future work, we target to enhance the approach to exploit the parallelism of the modern GPU-based architectures.

References:

Adiego, J., & Navarro, G. (2007). Lempel-Ziv compression of highly structured documents. Journal of the Association for Information Science and Technology, 58(4), 461-478.

Ahmad, I., Patil, S., & Sarangi, S. R. (2018). HPXA: A highly parallel XML parser. 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE),

Andrei, S. (2009). Parallel Parsing-based Reverse Engineering. Computer Science and Information Engineering, 2009 WRI World Congress on,

Arion, A., Bonifati, A., Manolescu, I., & Pugliese, A. (2007). XQueC: A query-conscious compressed XML database. ACM Transactions on Internet Technology (TOIT), 7(2), 10.

Beck, M., Schubotz, M., Stange, V., Meuschke, N., & Gipp, B. (2021). Recognize, Annotate, and Visualize Parallel Content Structures in XML Documents. 2021 ACM/IEEE Joint Conference on Digital Libraries (JCDL),

Braganholo, V., & Mattoso, M. (2014). A survey on xml fragmentation. ACM SIGMOD Record, 43(3), 24-35.

Brisaboa, N. R., Fariña, A., Navarro, G., & Paramá, J. R. (2007). Lightweight natural language text compression. Information Retrieval, 10(1), 1-33.

Camacho-Rodríguez, J., Colazzo, D., & Manolescu, I. (2015). Paxquery: Efficient parallel processing of complex xquery. IEEE Transactions on Knowledge and Data Engineering, 27(7), 1977-1991.

Cao, J., Rao, F.-Y., Kuzu, M., Bertino, E., & Kantarcioglu, M. (2013). Efficient tree pattern queries on encrypted xml documents. Proceedings of the Joint EDBT/ICDT 2013 workshops,

Chang, C.-C., & Lin, C.-J. (2011). LIBSVM: a library for support vector machines. ACM transactions on intelligent systems and technology (TIST), 2(3), 1-27.

Cheng, J., & Ng, W. (2004). XQzip: Querying compressed XML using structural indexing. International conference on extending database technology,

Cleary, J., & Witten, I. (1984). Data compression using adaptive coding and partial string matching. IEEE transactions on Communications, 32(4), 396-402.

DOM, W3C Recommendation of Document Object Model. Retrieved 14/06/2024 from www.w3.org/DOM .

Eckstein, R., & Casabianca, M. (2001). XML pocket reference. " O'Reilly Media, Inc.".

Fawcett, J., Ayers, D., & Quin, L. R. (2012). Beginning XML. John Wiley & Sons.

Fraigniaud, P., & Korman, A. (2016). An optimal ancestry labeling scheme with applications to XML trees and universal posets. Journal of the ACM (JACM), 63(1), 6.

Girardot, M., & Sundaresan, N. (2000). Millau: an encoding format for efficient representation and exchange of XML over the Web. Computer Networks, 33(1-6), 747-765.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098-1101.

JDOM. Retrieved 11/06/2024 from www.tutorialspoint.com/java_xml/java_dom_parser.htm

Leighton, G., Diamond, J., & Muldner, T. (2005). AXECHOP: a grammar-based compressor for XML. Data Compression Conference, 2005. Proceedings. DCC 2005,

Li, W. (2003). Xcomp: An XML compression tool University of Waterloo [School of Computer Science]].

Liefke, H., & Suciu, D. (2000). XMill: an efficient compressor for XML data. ACM Sigmod Record,

Liu, J., & Yan, D. (2016). Answering approximate queries over XML data. IEEE Transactions on Fuzzy Systems, 24(2), 288-305.

Lu, W., Chiu, K., & Pan, Y. (2006). A parallel approach to XML parsing. Proceedings of the 7th IEEE/ACM International Conference on Grid Computing,

Lv, T., & Yan, P. (2013). A framework of summarizing XML documents with schemas. Int. Arab J. Inf. Technol., 10(1), 18-27.

Megginson, D. (2004). Simple API for XML. In.

Min, J.-K., Park, M.-J., & Chung, C.-W. (2003). XPRESS: A queriable compression for XML data. Proceedings of the 2003 ACM SIGMOD international conference on Management of data,

Moussalli, R., Salloum, M., Halstead, R., Najjar, W., & Tsotras, V. J. (2014). A study on parallelizing XML path filtering using accelerators. ACM Transactions on Embedded Computing Systems (TECS), 13(4), 93.

Ng, W., Lam, W.-Y., Wood, P. T., & Levene, M. (2006). XCQ: A queriable XML compression system. Knowledge and Information Systems, 10(4), 421-452.

Parameswaran, A., Kaushik, R., & Arasu, A. (2013). Efficient parsing-based search over structured data. Proceedings of the 22nd ACM international conference on Information & Knowledge Management,

Qin, Z., Tang, Y., Tang, F., Xiao, J., Huang, C., & Xu, H. (2017). Efficient XML query and update processing using a novel prime-based middle fraction labeling scheme. China Communications, 14(3), 145-157.

Schneider J, K. T., Peintner D, Kyusakov R. (2014). Efficient XML Interchange (EXI) Format 1.0. In (Second ed.).

Silva de Moura, E., Navarro, G., Ziviani, N., & Baeza-Yates, R. (2000). Fast and flexible word searching on compressed text. ACM Transactions on Information Systems (TOIS), 18(2), 113-139.

Subramanian, H., & Shankar, P. (2005). Compressing XML documents using recursive finite state automata. International Conference on Implementation and Application of Automata,

Tang, J., Liu, S., Liu, C., Gu, Z., & Gaudiot, J.-L. (2013). Acceleration of xml parsing through prefetching. IEEE Transactions on computers, 62(8), 1616-1628.

Tolani, P. M., & Haritsa, J. R. (2002). XGRIND: A query-friendly XML compressor. Data Engineering, 2002. Proceedings. 18th International Conference on,

Toman, V. (2003). Compression of XML data. MFF UK.

Welch, T. A. (1984). Technique for high-performance data compression. Computer(52).

Wu, Y., Zhang, Q., & Yu, Z. (2008). Jianhui Li. A hybrid parallel processing for XML parsing and schema validation. Balisage: The Markup Conference,

XimpleWare. (2015). VTD-XML:The Future of XML of Processing. In.

XML Tutorial. Retrieved 16/07/2024 from www.w3schools.com/xml/default.asp

Yang, D., Wei, Z., & Yang, Y. (2015). A novel implementation of a Hash function based on XML DOM parser. Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2015 International Conference on,

You, C.-H., & Wang, S.-D. (2011). A data parallel approach to XML parsing and query. 2011 IEEE International Conference on High Performance Computing and Communications,

Zhang, J. (2006). Simplify XML processing with VTD-XML. In.

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. IEEE Transactions on information theory, 23(3), 337-343.

Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. IEEE transactions on Information Theory, 24(5), 530-536.