International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 8s
DOI: https://doi.org/10.17762/ijritcc.v11i8s.7238
Article Received: 26 April 2023 Revised: 15 June 2023 Accepted: 30 June 2023

Strategies and Approaches for Generating Identical Extensive XML Tree Instances

[1]Dr. Bhargavi Peddi Reddy, [2]Harikrishna Bommala, [3]Dr. Srikanth Bhyrapuneni
[1]Associate professor
Department of Computer Science and Engineering
Vasavi college of engineering , Hyderabad
bhargavi.peddyreddy@gmail.com
[2]Department of Computer Science and Engineering
KG Reddy College of Engineering & Technology
Moinabad, Hyderabad, Telangana, India
haribommala@gmail.com
[3]Associate professor
Department of Computer Science and Engineering
Koneru Lakshmaiah Education Foundation
Vijayawada, AP, India
bsrikanth@kluniversity.in

Abstract— In recent years, XML has become the de facto internet wire language. Data may be organized and given context with the use of XML. A well-organized document facilitates the transformation of raw data into actionable intelligence. In B2B1 applications, the XML data is sent and created. This implies the need for fast query processing on XML data. The processing of XML tree sample queries (XTPQ) that provide an efficient response (also known as sample matching) is a topic of active study in the XML database field.DOM (Parser) may be used to transform an XML document into a tree representation. Extensible Markup Language (XML) query languages like XPath and XQuery use tree samples (twigs) to express query results.XML query processing focuses mostly on effectively locating all instances of twig 1 samples inside an XML database. Numerous techniques for matching such tree samples have been presented in recent years. In this study, we survey recent developments in XTPQ processing. This summary will begin by introducing several algorithms for twig sample matching and then go on to provide some background on holistic techniques to process XTPQ.

Keywords- XML Tree Generation, Identical XML Instances, Extensive XML Structures, TwigStack.

I. INTRODUCTION

The necessity for XML data in enterprise data transfer systems is expanding. Twig samples (also known as sample matching) are the results of evaluating XML tree sample queries (XTPQ). The growth of this 1 B2B point calls for effective sample matching techniques on massive XML data sets in order to evaluate tree samples (twigs). The DOM parser creates an XML tree to represent the document. Once again, you may choose between ordered (ancestor and left-to-right ordering among sibling connections matter) and unordered (only ancestor relationship matters) XML trees. The results of certain algorithms are presented as unordered XML trees, whereas the results of others are presented as ordered, labelled XML trees (twigs).In the past, XML trees were analysed as a series of ordered, tagged twigs. Ordered matching ignores the possibility that the sub-elements first name and last name can be in reverse order when looking for a child node of the element student with those nodes and their associated values. It's possible, however, that this is the pupil we've been seeking. The only important connections in the query twig are those between ancestors and descendants; the other links (axes) between ancestors and siblings, between siblings, and between siblings and the ancestors are all unimportant. As XML becomes the go-to format for representing data, there has been a surge of interest in query processing over tree-structured data models. Since trees are often used to represent data objects in many computer languages (e.g. XPath [1], XQuery [2]), twig sample matching is of paramount importance. Book [title='XML'] is the query.//author [name='Jane']'s sample may be visualised as a twig. Book elements having a child title element whose value is "XML" and whose sub-element name includes the string "Jane" are matched. In the above query, "/" represents a parent-child connection and "//" represents an ancestor-descendant one. In practise, XML data may be quite large, complex, and include nested elements. Finding all twig samples in a database of

XML documents rapidly is, thus, an important part of processing XML queries. There have been several proposals for matching such twig samples ([3, 4]) in recent years. These techniques first establish a tagging scheme to record the structure of XML files, and then use this information to Use just the labels to match tree samples, rather than the XML files themselves.

In the past, solutions to this problem have focused on developing labelling systems based on tree-traversal ordering, textual placements of start and end tags (such as region encoding [5]), route expressions (such as Dewey ID [6]), or prime numbers (such as [7]). These labelling techniques allow us to examine the labels of two XML document components and infer their relationship to one another (ancestor, descendant, parent, child, etc.).

II. RN MATCHING TWIG

XML queries are kept in the form of twigs in both Lore [8] and Timber [9]. Storage and query processing of XML data in relational databases have been investigated [6, 7]. Processing of XML twig samples (XTPQ) has been made more efficient in recent papers [10, 11]. Ordered, a comprehensive approach to ordering XML tree queries, is presented in [10]. TwigStackListNot is a paper [11] that deals with negation questions. Data streaming algorithms were established by Chen et al. [12] to enhance the holistic nature of processing XML tree samples. There was a larger optimal class because of improved techniques for streaming data. Requests for generic examples of XML trees are likewise satisfied by Twig2Stack [13]. Here, examples of generic and extended XML trees are shown. The optional axis in this generalised XML tree example represents the LET and RETURN clause expressions in XQuery. Negative functions, wildcards, and order restrictions are only some of the numerous constraints shown in the aforementioned big XML tree sample. XML tree sample matches may also be converted into sequence matches by using ViST[14] or PRIX[15]. It is challenging to extend these two methods to handle unordered queries. The research shows that the holistic tree sample technique is reliable and guarantees performance [16]. Choi et al. [20] conducted a theoretical investigation into XML tree sample matching and found that no unified method can guarantee optimality for queries with every possible permutation of Parent-Child and Ancestor-Descendant relationships. The complexity of the query space for XML twigs was studied by Shalem et al. [21]. Based on their findings, the maximum time required for a full-edge search that includes Parent-Child and Ancestor-Descendant edges is O(D), where D is the total number of documents. Theoretically, their results show that no algorithm can effectively process any query of the form Q/, //, *. XML element tagging ensures that data relationships are correct. Most labels use a prefix or a limiting phrase. To improve query results, Zhang et al. [17] applied containment labelling. Parent-child and ancestor-descendant relationships are intricate, as seen by regional labelling. Labelling XML prefixes with an example using Dewey ID. During the query processing, the path information is kept safe. Route information, such as element identifiers and names, is encoded in Lu at el. [14]'s complex Dewey encoding [18].

III. HOLISTIC XML QUERY PROCESSING ALGORITHMS

In this work, we provide two distinct techniques for handling an XML twig query. Two-stage methods for evaluating the health of a twig Algorithms for the comprehensive assessment of twigs in a single phase

A. The TwigStack Algorithm

Bruno et al. [5] developed a unique holistic XML twig sample matching approach called TwigStack based on the confinement labelling system [17], which eliminates the need to save intermediate findings unless they contribute to the final results. Unlike the decomposition-based approach, this one doesn't involve calculating a lot of unnecessary intermediate outcomes. TwigStack's primary drawback is that it may generate a high number of "useless" intermediate results if a query has a parent-child connection. Although it has been shown that for searches involving just A-D edges, TwigStack's algorithms are I/O-optimal in terms of output size, for queries including parent-child (P-C) edges, they still lack the ability to regulate the size of intermediate results. TwigStack consists of two stages: (1) generating a list of intermediate route solutions as intermediate results, and (2) performing a merge-join on the list of intermediate path solutions to get the final solutions.

TwigStack's Algorithm:

1. Initiate a while loop that will run until the query finishes processing.
2. Extract the next nested query element.
3. See if the current element is the root.
4. If the current element is not the root, clean the parent stack by eliminating elements until you reach the beginning of the stack.
5. If this is the first item in the stack or if the parent stack is not empty:
6. Eliminate all items from the self stack until you reach the end of the current element.
7. Using the "moveToStack" method with a pointer to the parent stack's top element, copy the current element into the "Sqact" stack.

8. If this node is a leaf, do the following.
9. Using the "Sqact" stack, demonstrate your solutions, which may include blocking.
10. Take the top "Sqact" stack item and pop it.
11. If the current node is not a leaf node, then:
12. The next round of query processing has begun.
13. Iterate the loop until the inquiry is complete.
14. Combine the results of all the paths found when processing the query.

B. TwigStackList Algorithm

Our method, in contrast to the older Algorithm TwigStack [5], considers the level information of elements, which results in significantly less intermediate pathways being reported for query twig samples that contain parent-child edges. We have demonstrated analytically that the I/O cost of TwigStackList is only equal to the sum of the sizes of the input and the output when all edges below branching nodes (nodes that has more than one child) in the query sample are ancestor-descendant connections. In other words, unlike TwigStack, which only guarantees the optimality for queries with exclusively A-D relationships, TwigStackList [19] finds a bigger query class to ensure the I/O optimality. For queries involving only ancestor-descendant relationships, our method achieves similar performance to TwigStack, according to experimental results. However, for queries involving parent-child relationships, our method is significantly more efficient than TwigStack, especially for deep data sets with complicated recursive structure.

Algorithm for TwigStackList:

1. Set up a repeating loop that will run indefinitely.
2. Locate the following node in the XML file and store its contents in the qact variable.
3. To see if the present node (qact) is the root, we need to test it.
4. If this is not the root node, remove all children from the parent stack, starting at the node indicated by getStart(qact).
5. The if clause is done.
6. Verify that the parent stack (Sparent) is not empty or that the present node is the root node.
7. If the current node is the root or the parent stack is not empty, then all items in the self stack (Sqact) up to and including the node represented by getEnd(qact) should be removed.
8. Influence the current.

C. Ordered TJ Algorithm

It's an enhancement to the existing TwigStackList functionality. The following is accomplished in this work: a) We provide a new method, which we call OrderedTJ[10], for processing ordered twig samples on a global scale. In OrderedTJ, an item only counts towards the final tally if the order of its children coincides with the order of relevant query nodes.

When the ordered query only includes A-D relations from the second branching edge, we show analytically that OrderedTJ [10] is I/O optimal among all sequential algorithms that read the entire input. This terminology is used to distinguish edges between branching nodes and their children as branching edges and the branching edge connecting to the then'th child as the then'th branching edge. According to OrderedTJ's optimality, P-C edges may be present in both the initial branching edge (a node with several children) and the non-branching edge (a node with just one child).

Algorithm for OrderedTJ

1. Start a while loop until the end of the query is reached.
2. Get the next element from the root of the query and assign it to the variable 'qact'.
3. If the current element is the root element or if the parent stack associated with 'qact' is not empty, execute the following steps: a. Clean the stack associated with 'qact' by removing elements until reaching the end of 'qact'.
4. Move the stream to the stack 'Sqact' using the 'moveStreamToStack' function, with 'qact' as the input.
5. If the current element is a leaf node, execute the following steps: a. Show path solutions using the 'Sq' stack and the element obtained from 'qact'.
6. If the current element is not a leaf node, proceed to the next step in processing the query.
7. Repeat the loop until the end of the query is reached.
8. Merge all path solutions obtained during the query processing.

D. TJFast Algorithm (A Fast Twig Join Algorithm)

A Fast Twig Join (FTJ) algorithm is an efficient approach for executing twig pattern queries on XML data. Twig pattern queries are typically used to retrieve structured information from XML documents. The main idea behind the FTJ algorithm is to minimize the number of intermediate results and

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 8s
DOI: https://doi.org/10.17762/ijritcc.v11i8s.7238
***Article Received**: 26 April 2023 **Revised:** 15 June 2023 **Accepted:** 30 June 2023*

reduce the search space during query processing. It achieves this by exploiting the structural relationships and the order of nodes in the XML document. Here is a high-level overview of the steps involved in a typical FTJ algorithm:

1. To optimise query processing, an index structure is formed on the XML document. The index structure captures XML element structural connections like parent-child and ancestor-descendant.

2. Decompose the twig pattern by dividing the query into smaller subpatterns based on the structural relationships between the nodes. Both the number of intermediate results and the search space may be reduced by using decomposition.

3. The algorithm works from the bottom up, joining smaller patterns as it goes the join operation between the pertinent nodes for each sub-pattern is performed using an index structure in this approach.

4. Filtering and Optimization: The method uses filtering and optimization to minimize search space and increase query speed. Structural pruning, early result termination, and efficient index lookups are examples.

5. The procedure combines intermediate results to obtain the final result set that matches the original twig pattern query after assessing all sub-patterns.

The specific implementation details and optimizations may vary based on the FTJ algorithm variant and research advancements in the field. FTJ algorithms aim to strike a balance between query performance and resource utilization, making them efficient for processing twig pattern queries on XML data.

E. TreeMatch Algorithm

This approach increases optimum query classes. It matches results using simple encoding and minimises superfluous intermediary outcomes.

1. `locateMatchLabel(Q)`: This function is responsible for locating the next element in the hierarchical structure that matches the given query pattern, represented by 'Q'. The details of how this function locates the match are not provided in the code snippet.

2. Start a while loop until the end of the root element is reached.

3. `fact = getNext(topBranchingNode)`: Get the next top-level branching node from the root of the hierarchical structure and assign it to the variable 'fact'.

4. Check if 'fact' is a return node, indicating that it matches the query pattern.

5. If 'fact' is a return node, add the corresponding element from the hierarchical structure to the output list. The function `NAB(fact)` retrieves the element associated with 'fact', and `cur(Tfact)` represents the current position in 'Tfact'.

6. Advance to the next element in 'Tfact', which is the hierarchical structure being processed.

7. Update the set encoding, possibly related to the matching or processing of the element.

8. `locateMatchLabel(Q)`: Locate the next element in the hierarchical structure that matches the given query pattern. This function is called recursively to continue matching the query pattern.

9. Empty all sets related to the root element. This step may involve resetting or clearing any sets used for encoding or tracking information during the matching process.

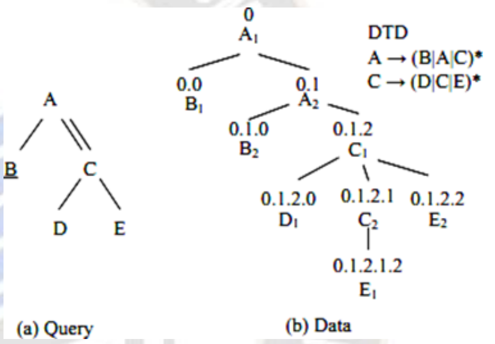

Figure (a and b): Illustration to Algorithm TreeMatch for class Q/,//,*

It's important to note that the specific details of functions like `locateMatchLabel`, `addToOutputList`, `updateSet`, `NAB`, and `emptyAllSets` are not provided in the code snippet. Additional context or information would be required to understand the functionality and behavior of these functions and the overall algorithm.

TABLE I.

Table Column Head		
Current Elements	***set encoding s a***	***set encoding s c***
B1,D1,E1	<0, "10",Q>	<0.1.2, "10",Q> <0.1.2.1, "01",Q>
B1,D1,E2	<0, "11","0.0">	<0.1.2, "11",Q> <0.1.2.1, "11",Q>
B2,D1,E2	<0, "11","0.0"> <0.1, "11","0.1.0">	<0.1, "11","0.0"> <0.1.2.1, "10",Q>

TABLE1 lists access, settings encoding, and output elements. Queries branch twice. Scan B1, D1, and E1 first. C1 and C2 are added to SC with bit Vectors "10" and "01," indicating they have one child each. TJFast may yield route solutions A1/A2/C1/D1 and A1/A2/C1/C2/E1 that are not suitable for final findings. TreeMatch eliminates unnecessary I/O. Since C1 has two progeny, E2 is scanned and the bit Vector (C1) becomes "11." 11 is A1's bitVector. A1 matches the sample tree since bitVector (A1) has all 1s. SA gets A2 after scanning B2. Treematch yields B1 and B2 results.

TJFast and TreeMatch have two differences.

1) TreeMatch uses bitVector encoding to solve TJFast's unnecessary intermediary path A1/A2/C1/C2/E1.

2) TreeMatch provides return nodes (node B in the query) to reduce I/O cost, whereas TJFast outputs the route solution for

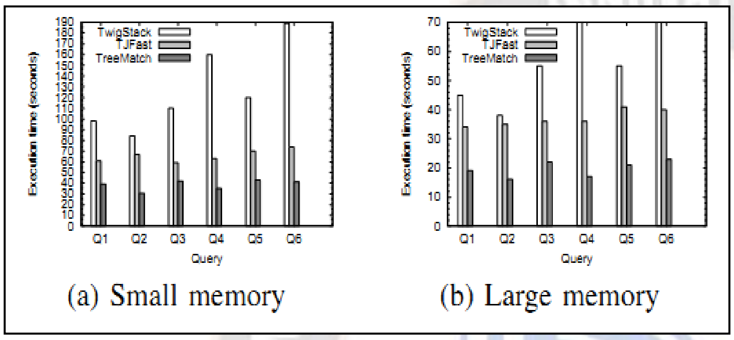

Figure. (b) Execution time of Q/,//,* on random data

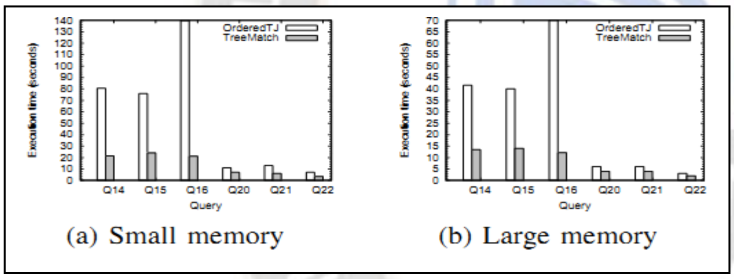

Figure (c): Execution time of Q/,//,*,< on random data

IV. CONCLUSION

The current investigation addressed the XML twig sample matching issue and reviewed previous techniques. TwigStack, TwigStackList, OrderedTJ, TJFast, and TreeMatch are introduced. TreeMatch runs fast and processes generalised tree samples. TwigStack, TwigStackList, OrderedTJ, and TJFast work on two-phase query evaluation, whereas TreeMatch works on one-phase query assessment. TreeMatch twig sample matching method may answer complex questions and perform well.

REFERENCES

[1]. Berglund, S. Boag, and D. Chamberlin. XML path language (XPath) 2.0. W3C Recommendation 23 January 2007 http://www.w3.org/TR/xpath20/.

[2]. S. Boag, D. Chamberlin, and M. F. Fernandez. Xquery 1.0: An XML query language. W3C Working Draft 22 August 2003.

[3]. Choi, M. Mahoui, and D. Wood. On the optimality of the holistic twig join algorithms. In Proceeding of DEXA, pages 28–37, 2003.

[4]. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In Proc. of SIGMODConference, pages 425-436, 2001.

[5]. H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for querying XML data by tree structures. InSIGMOD, pages 110-121, 2003.

[6]. Singh, P. ., & Sharma, D. V. . (2023). Pre-Processing of Mobile Camera Captured Images for OCR . International Journal of Intelligent Systems and Applications in Engineering, 11(2s), 147–155. Retrieved from https://ijisae.org/index.php/IJISAE/article/view/2518

[7]. T. Yu, T. W. Ling, and J. Lu. Twigstacklistnot: A holistic twig join algorithm for twig query with not-predicates on xml data. In DASFAA, pages 249-263, 2006.

[8]. H. V. Jagadish and S. AL-Khalifa. Timber: A native XML database. Technical report, University of Michigan, 2002.

[9]. X. Wu, M. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered XML trees. In Proc. Of ICDE,pages 66-78, 2004.

[10]. N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: optimal XML sample matching. In Proc. of SIGMOD Conference, pages 310-321, 2002.

[11]. Avinash, D. & Nalebuff, B. (1991). Thinking Strategically. New York: Norton & Co.

[12]. Bicchieri A & Cristina J. (2007). Game Theory: Some Personal Reflections. In V.F. Hendricks and P.G.Hansen (eds.), Game Theory: 5 Questions, Copenhagen: Automatic Press.

[13]. Binmore, K. (1991). Fun and Games: A Text on Game Theory. D.C.: Heath Lexington, MA.

[14]. Binmore, K. (2007). Playing for Real: A Text on Game Theory. New York: OxfordUniversity Press.

[15]. Brook, T. (2007). Computing the Mixed Strategy Nash Equilibria for Zero-Sum Games. Bath, U.K.: University of Bath.

[16]. Davis, M. (1983). Game Theory: A Nontechnical Introduction. New York: Basic Books.

[17]. Davis, M. (1997). Game Theory: A Non-Technical Introduction. New York: Dover Books.

[18]. De Bruin, B. (2009). Over mathematisation in game theory: Pitting the Nash Equilibrium Refinement. Studies in History and Philosophy of Science, 40, 2-10.

[19]. Dixit, A. K. and Nalebuff, B. J. (1991). Thinking Strategically: The Competitive Edge in Business, Politics, and Everyday Life. NewYork: Norton.

[20]. Gilbbons, R. (1992). Game Theory for Applied Economics. New Jersey: Princeton University Press, Princeton.

[21]. Gipin, A. & Sandholm, T. (2007). Lossless abstraction of imperfect information games. Journal of the ACM, 54(5), 2-29.

[22]. Kelly, A. (2003). Decision Making using Game Theory: An Introduction for Managers. Cambridge, United Kingdom: Cambridge University Press.

[23]. Klempere, P. (1999). The Economic Theory of Auctions. London: Edward Elgar Lim, J.J. (1999). Fun, Games & Economics: An appraisal of game theory in economics. Undergraduate Journal of Economics, 3(1), 2-22.

[24]. Fibonacci technique for privacy and security to sensitive data on cloud environment H Bommala, S Kiran, T Venkateswarlu, MAA Sheela - International Journal of Advanced Networking and …, 2020.