American Journal of Applied Sciences 5 (9): 1199-1205, 2008
ISSN 1546-9239

TwigINLAB: A Decomposition-Matching-Merging Approach To Improving XML Query Processing

Su-Cheng Haw and Chien-Sing Lee
Multimedia University
Faculty of Information Technology
63100 Cyberjaya, Malaysia

Abstract: The emergence of the Web has increased significant interests in querying XML data. Current methods for XML query processing still suffers from producing large intermediate results and are not efficient in supporting query with mixed types of relationships. We propose the TwigINLAB algorithm to process and optimize the query evaluation. Our TwigINLAB adopts the decomposition-matching-merging approach and focuses on optimizing all three sub-processes; introducing a novel compact labeling scheme, optimizing the matching phase and reducing the number of inspection required in the merging phase. Experimental results indicate that TwigINLAB can process both path queries and twig queries better than the TwigStack algorithm on an average of 21.7% and 18.7% respectively in terms of execution time using the SwissProt dataset.

Keywords: XML, query optimization, structural query, twig query, decomposition-matching-merging

INTRODUCTION

eXtensible Mark-up Language (XML) is emerging as the *de facto* standard for data exchange over the Web. Since XML is a semi-structured data, two types of user queries namely full-text queries (keyword based search) and structural queries (complex queries specified in tree-like structure) are usually used [1]. This paper is concerned with structural queries.

Structural queries can be viewed as sequences of location steps, where each node in the sequence is an element tag or string value. Query nodes are related by either parent-child (P-C) steps or ancestor-descendant (A-D) steps. These relationships are depicted with a single line and double lines respectively. Besides, query nodes can be related adjacently with one another by sibling or ordered query relationship. Sibling (ordered query) relationship is usually denoted by "[]".

To process such queries, it may undergo a decomposition-matching-merging process. TWIG-XSKETCH [2], tree signature [3], MPMGJN [4], Stack-Tree [5] and PathStack and TwigStack [6] are examples of query processing using the decomposition-matching-merging approaches. Nevertheless, most of these approaches focus on the second sub-process: the matching phase only.

In this paper, we propose

1. a novel hybrid query optimization architecture, INLAB (combination of INdexing and LABeling techniques), which comprises an XML Parser, XML Encoder, XML Indexer and Query Engine and
2. query optimization algorithms (TwigINLAB) to process twig queries efficiently without traversing the whole XML tree.

INLAB labeling scheme size is only 12 bytes; much shorter compared to previous labeling schemes. This enables quick determination of P-C relationship between elements in the XML database. However, to check for A-D relationship, the index table need to be accessed for confirmation. Besides, INLAB labeling is integer based. Integer processing is very efficient compared to that of string or bit-vector. The index structures of INLAB allow us to efficiently find all elements that belong to the same parent or ancestor.

Our TwigINLAB approach decomposes relationships into a set of path queries. In addition, we focus on optimizing all three decomposition-matching-merging sub-processes. First, we introduce a novel robust and compact labeling scheme consisting of <*self–level: parent*> to allow quick determination and

Corresponding Author: Su-Cheng Haw, Faculty of Information Technology, Multimedia University, 63100 Cyberjaya, Malaysia, Tel: +603-83125233, Fax: +603-83125264

decomposition of the types of relationships among each path edge. Subsequently, we optimize the matching phase based on each relationship and finally reduce the number of inspection required in the merging phase.

Twig Query Processing: With the increasing popularity of XML data representation, XML query processing and optimization has attracted a lot of research interest [7, 8, 9, 10, 11, 12]. In this section, we summarize the related work. There are typically two types of decomposition-matching-merging process. First, a complex query pattern can be decomposed into a set of basic binary relationships between each pair of nodes or second, it can be decomposed into a set of path queries, followed by subsequent matching and merging processes. Our INLAB adopts the latter approach and focuses on optimizing all three sub-processes; introducing novel compact labeling scheme, optimizing the matching phase and reducing the number of inspection required in the merging phase. In the first sub-process, most researchers use the labeling of *(docno, begin : end, level)* for an element and *(docno, wordno, level)* for a text word as the positional representation of XML elements and texts. However, we use <*self – level : parent*> as the positional representation instead. The details on this will be explained in the next section. MPMGJN [4], Stack-Tree [5] and TwigStack [6] algorithms are based on *(docno, begin : end, level)* labeling of XML elements. These algorithms accept two lists of sorted individual matching nodes and structurally join pairs of nodes from both lists to produce the matching of the binary relationships. Another similar approach is to decompose the twig query into a set of path queries instead. Polyzotis et al. propose methods to reduce the number of intermediate results by introducing a filtration step based on some notion of synopses to facilitate query-approximate answers [2]. They propose both TREESKETCH and TWIG-XSKETCH. Another work, done by Amer-Yahia et al is to preprocess the query patterns before the matching phase is executed [13]. Since the efficiency of tree pattern matching depends on the size of the pattern, it is essential to identify and eliminate redundant nodes in the pattern before the matching phase takes place. On the other hand, Zezula et al. propose a novel technique, tree signature, to represent tree structures as ordered sequences of pre-order and post-order ranks of the nodes [3]. They use tree signatures as index structure and find qualifying patterns through integration of structurally consistent path query. Merging together the structural matches in the final process poses the problem of selecting a good join ordering. Wu et al. propose a cost-based join order selection of structural join [7]. Kim et al. suggest partitioning all nodes in an extent into several clusters [14]. Given two extents to be joined, they propose filtering out unnecessary clusters in both extents prior to the joining process.

Our TwigINLAB algorithm is a generalization of the stack-based algorithm first mentioned by Bruno et al. [6] to match twig query. However, we enhance the query processing by utilizing indexes (built only once) to speed up the matching and merging phases. Further elaboration can be found in next section.

MATERIALS AND METHODS

Fig. 1 shows the INLAB architecture, which consists of the XML Parser to check the well-formedness of the XML document, the XML Encoder to generate the labeling based on a <*self-level:parent*> scheme, the XML Indexer to create index storing each node parent and child information and the XML Query Engine for pattern query matching. This paper concentrates only on the XML Query Engine (the optimizer). Other components such as XML Parser, XML Encoder and XML Indexer have been reported in [15, 16]. The criterion for assessing TwigINLAB is execution time.

Fig. 2 depicts an example of XML document labeled based on <s*elf-level: parent*>. Structural relationships between element nodes can be efficiently determined from the label as follows:

1. P-C relationship
   $node_1$ is the parent of $node_2$ if and only if $node_1.self = node_2.parent$.
2. Sibling relationship
   $node_1$ is the sibling of $node_2$ if and only if $node_1.parent = node_2.parent$.
3. Ordered query relationship (predecessor and successor)
   a. $node_1$ is the predecessor node of $node_2$ if and only if $node_1.self < node_2.self$.
   b. $node_1$ is the successor node of $node_2$ if and only if $node_1.self > node_2.self$.
4. A-D relationship
   $node_1$ is possible as an ancestor of $node_2$ if and only if $leveldiff = node_2.level - node_1.level >= 1$. A multiple look-up via PCTable is necessary as long as the $leveldiff > 1$ is true to confirm the A-D relationship.

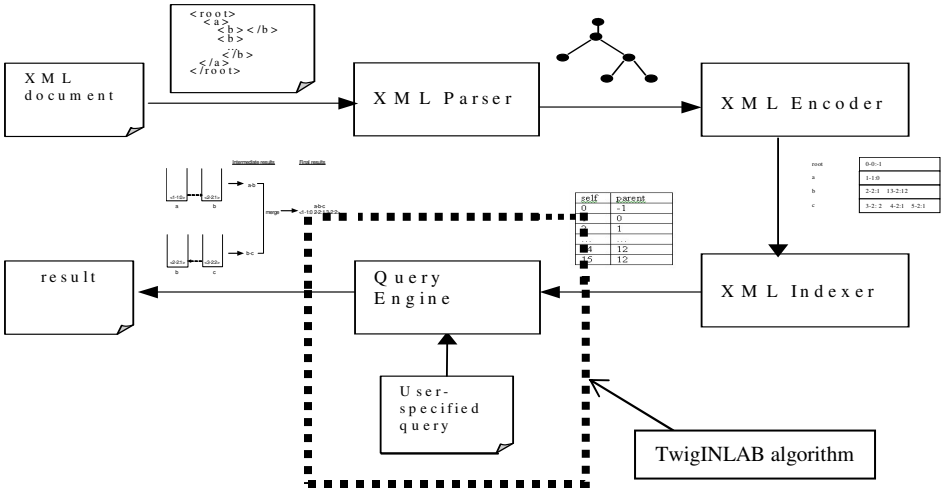

Fig. 1: Query processing component architecture

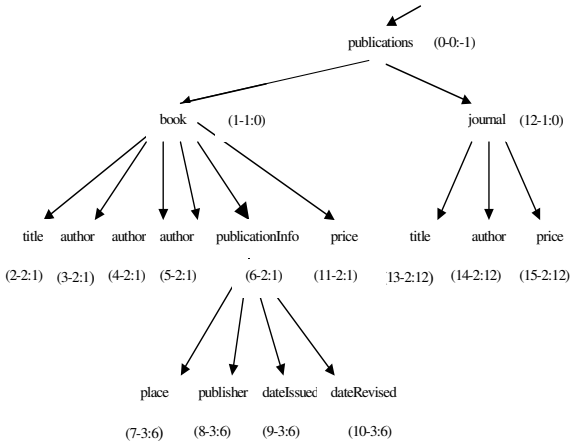

Fig. 2: A sample XML document with <self-level: parent> label.

For example, let *publications* (0-0:-1) be $node_1$ and *author* (14-2:12) be $node_2$. The *leveldiff* between the two nodes is two. This means that we need to trace up the PCTable twice starting from the *self* attribute of *author* to check whether *publications* is ancestor of *author* as illustrated in Fig. 3. The *parent* attribute of the retrieved node is equal to the *self* attribute of *publications*. Thus, *publications* and *author* is of A-D relationship.

self	parent
0	-1
1	0
…	…
12	0
13	12
14	12
15	12

Fig. 3: Fragment of PCTable index table.

Fig. 4 illustrates the overall processes involved in TwigINLAB processing. Initially, the query pattern is analyzed using the *analysisQueryPattern()* function. For each query edge, if the twig is of P-C relationship, the parent and child details will be updated in the *twigPC* (a hashtable to store parent and child) repository. During this process, each node in the twig query is associated with a stream. Each stream contains the positional representations of the node appearance in the XML tree (as shown in Fig. 5). The nodes in the stream are sorted by their *self* attribute, and thus, this will determine the order of the node to be processed. Associated with each stream is a stack. Stack is used to store the possible intermediate results.

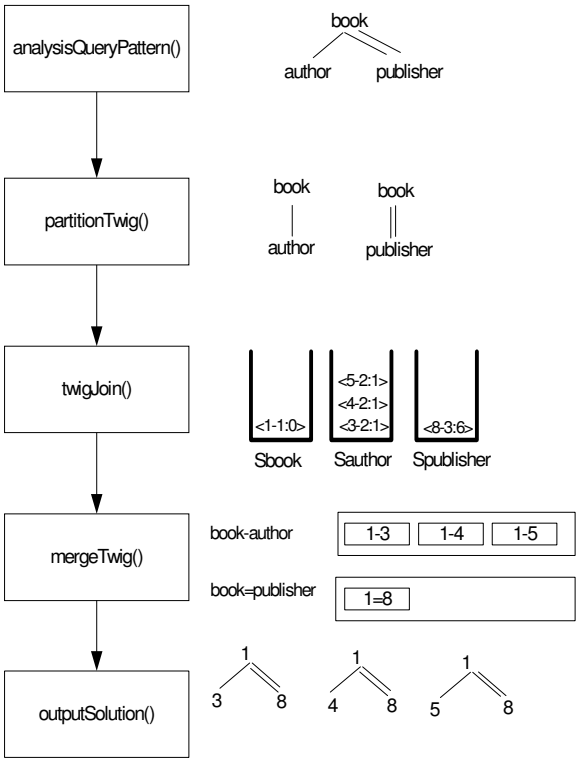

Fig. 4: Overall flow of TwigINLAB

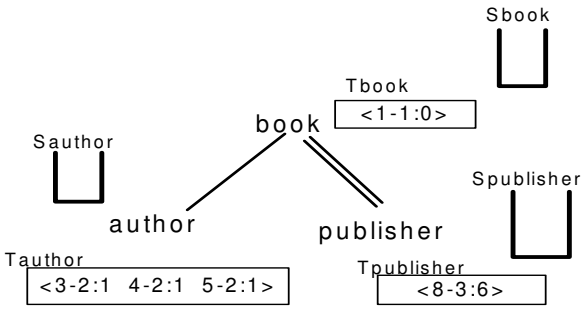

Fig. 5: Stack and stream assigned to each query node during the analysisQueryPattern() function.

Next, the *partitionTwig()* function takes place. If the query is path query (only one leaf node), this function is skipped and it will proceed to the *twigJoin()* function. However, if it is twig query, during this function, the twig pattern is decomposed into two or more path queries. Starting from the root of twig query pattern, for each start tag event, it pushes the tag into *twigStack* (a stack to keep track of twig query sequence). When it reaches an end tag event, it checks whether the current entry at the top of *twigStack* is a leaf node. If it is a leaf, the query node will be added one by one to the *vpathList* (a vector to store query nodes in leaf-to-root order) until it reaches the root. Finally, it will be output in reverse order by the function *reverse()*. The final output of this function is a set of path queries in root-to-leaf order in *pq* (a hashtable to keep each distinct path query).

For each path query, it recursively calls the *twigJoin()* function to find the possible path matches. Each possible match is pushed into the stack in the *twigJoin()* function. For instance, using the twig query in Fig. 5 as an example, after the *partitionTwig()* function, there are two path queries: *book-author* and *book=publisher*. Initially, the path query *book-author* is to be processed first. Based on the *self* attribute in each first occurrence in $T_{book,}$ and $T_{author,}$ query node *book* is being processed first. Element <1-1:0> is then pushed into $S_{book.}$ The next returned query node is the immediate child of *book*, which is *author*. Element <3-2:1> is pushed into S_{author} because *parent* attribute of *book* is equal to *self* attribute of *author*. Since *author* is the leaf query node, a partial solution is formed between *book-author*. Based on the next occurrences, the next returned node is element <4-2:1> as it has the next smallest *self* attribute. This element is then pushed into S_{author} because the *parent* attribute of *book* is equal to the *self* attribute of *author*. Since *author* is the leaf query node, another partial solution is formed between *book-author*. This process repeats until it reaches the leaf node of the all paths as illustrated in Fig. 6.

Next, these matches are merged back through the *mergeTwig()* function. In the *mergeTwig()* function, all partial solutions from the *twigJoin()* function are merged together to generate the final solutions. This function begins by comparing each entry in the partial solutions of two path queries at a time. All the occurrences in the partial solutions are in sorted order of their self-attributes. If each entry first node is equal, or if the query edge is of P-C relationship and the second query node is of sibling and predecessor relationship, the partial solution will be added to the final solutions. For query edge with A-D relationship, if the second query node is a predecessor, it will be added as a final solution. In both cases, the inner loop begins the iteration from the current *j* position. Hence, this function skips the unnecessary iteration of non-feasible partial solutions. However, if the first node in the second path query is greater than node1, the next inner loop will begin from position *j-1* (for cases where *j* > *0*). Fig. 7 illustrates the merging process.

Finally, the final solutions are output through the *outputSolution()* function.

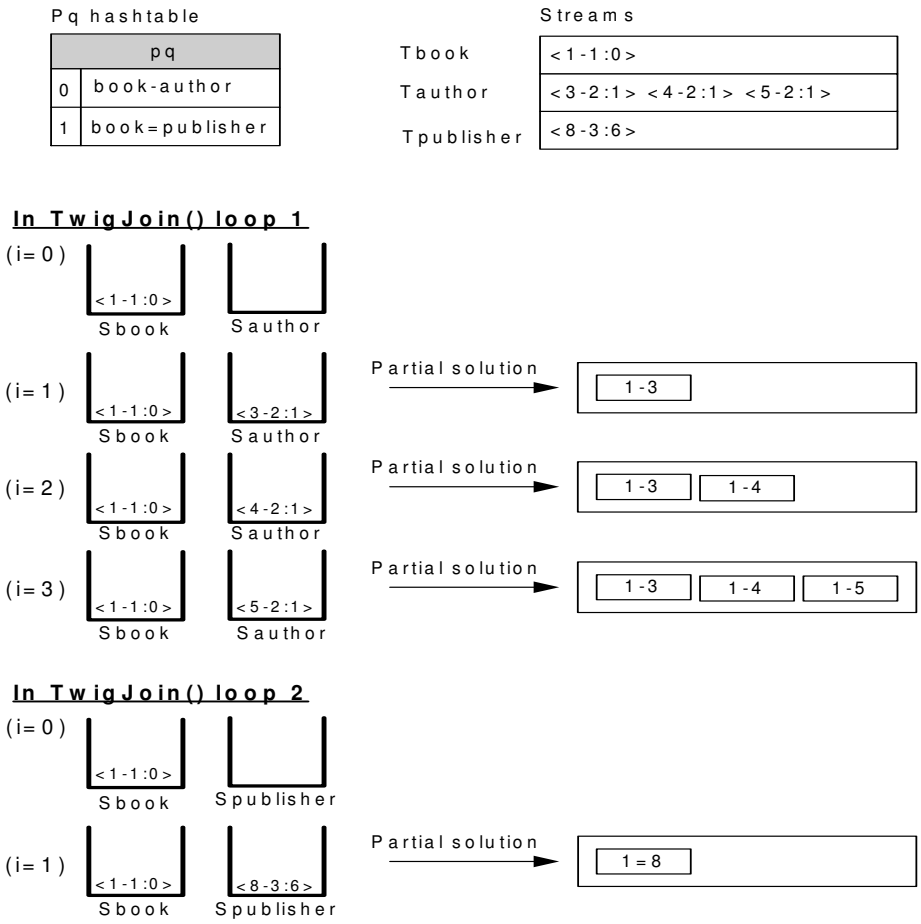

Fig. 6: Matching process in twigJoin() function.

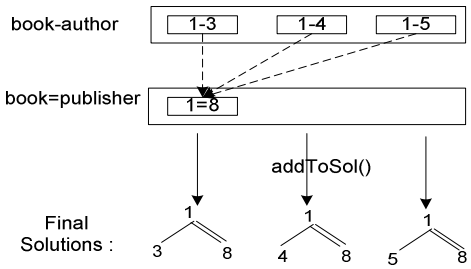

Fig. 7: The merging process scenario.

RESULTS AND DISCUSSION

We have implemented TwigINLAB using Java API for XML Processing (JAXP). Experiments have been carried out on the *SwissProt* dataset (112MB) obtained from the University of Washington XML repository [17]. We modified the *SwissProt* dataset into various file sizes ranging from 10MB until 110MB for the purpose of measuring the scalability of both approaches in supporting large-scale dataset.

We evaluated the performance of TwigINLAB as compared to TwigStack on two main types of queries namely, path query and twig query. For each type of query, we measure the performance of both algorithms on (a) Q1:-Query with P-C relationship (b) Q2:-Query with A-D relationship and (c) Q3:-Mixed query.

All our experiments are performed on 1.7GHz Pentium IV processor with 512 MB SDRAM running on Windows XP systems. All numbers presented here are produced by running the experiments multiple times and averaging the execution times of several consecutive runs.

Figures 8, 9 and 10 show the execution time of TwigINLAB and TwigStack for both path and twig

query. Fig. 8 shows the execution time of: Q1PQ= *Entry/Organelle* for path query and Q1TQ= *Entry[/Organelle]/Prints* for twig query over *Standard* dataset by varying the file sizes. From the result, TwigINLAB outperforms TwigStack in all the test cases by about 26.8% for path query and 24.5% for twig query.

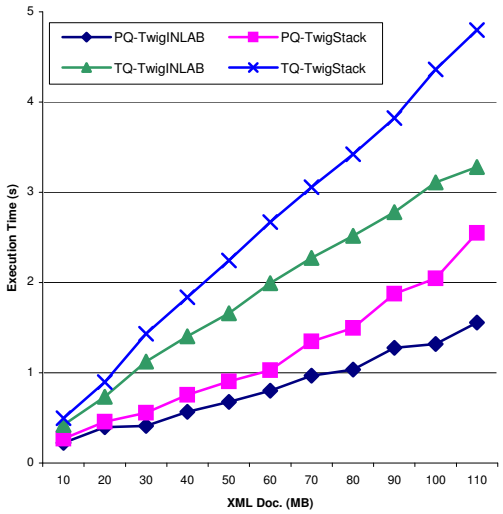

Fig. 8:Test results for Q1.

Fig. 9 shows the execution time of: Q2PQ = *Entry//MedlineID* for path query and Q2TQ= *Entry[//MedlineID]//Comment* for twig query respectively. TwigINLAB performs by about 21.2% better than TwigStack for path query and 17.8% for twig query.

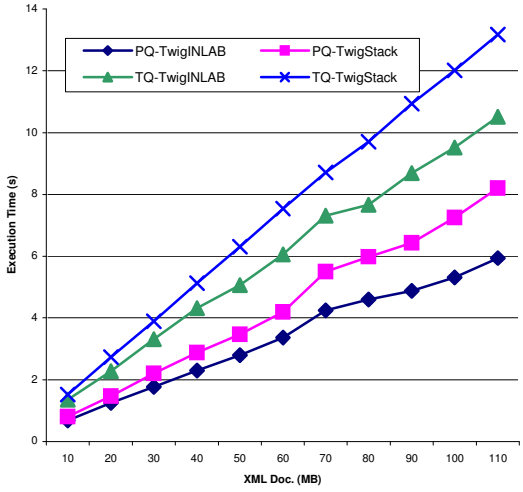

Fig. 9: Test results for Q2.

Fig. 10 shows the execution time of: Q3PQ = *Features//MUTAGEN//Descr* for path query and Q3TQ = *Features[//MUTAGEN//Descr]/Site* for twig query respectively. TwigINLAB performs about 17.1% better than TwigStack for path query and 13.7% for twig query.

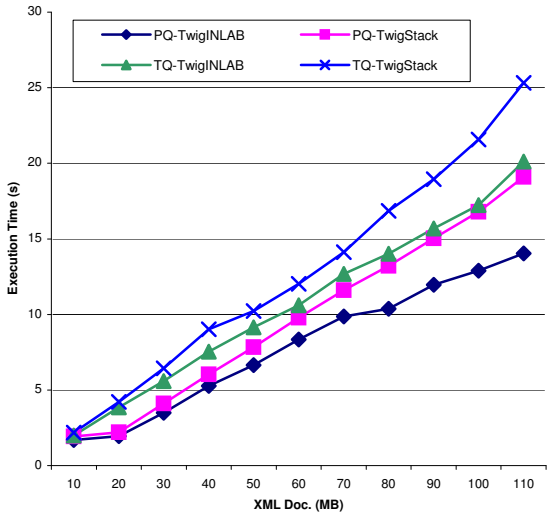

Fig. 10: Test results for Q3.

From these figures, we draw several observations and conclusions:-

- When the twig query contains only P-C edges, TwigINLAB performs around 24.5% better as compared to TwigStack (shown in Fig. 9). This may be due to the INLAB labeling scheme, which is optimal to support P-C relationships.
- Although TwigINLAB still outperforms TwigStack for query with edges of A-D relationship by around 17.8%, the difference is less significant as compared to query with edges of P-C relationships. This may be due to the extra time needed to determine whether the two nodes is in A-D relationship by multiple lookups on the index table until the ancestor level is reached.
- For each test case, TwigINLAB increases less drastically as compared to TwigStack. This shows that TwigINLAB is more scalable in processing large-scale datasets efficiently.

CONCLUSION

In this paper, we have presented the TwigINLAB algorithm to optimize all the sub-processes involved in the decomposition-matching-merging approaches. Experimental results show that, in terms of execution time, on average, TwigINLAB performs about 21.7% better for path query and about 18.7% better for twig query compared to the TwigStack. Also, TwigINLAB is more scalable compared to TwigStack. As such, TwigINLAB supports large-scale query of datasets efficiently.

ACKNOWLEDGEMENTS

This work was partially supported by funding from eScienceFund, Ministry of Science, Technology and Innovation, Malaysia.

REFERENCES

1. Haw, S.C. and Rao, G.S.V.R.K., 2005. Query Optimization Techniques for XML Databases. International Journal of Information Technology, 2(1): 97-104.
2. Polyzotis, N., Garofalakis, M., Ioannidis, Y., 2004. Approximate XML Query Answers, Proceedings of ACM SIGMOD, pp: 263–274.
3. Zezula, P., Mandreoli, F., Martoglia, R., 2004. Tree Signatures and Unordered XML Pattern Matching, Proceedings of SOFSEM, pp:122–139.
4. Zhang, C., Naughton, J., DeWitty, D., Luo, Q., Lohman, G., 2001. On Supporting Containment Queries in Relational Database Management Systems, Proceedings of ACM SIGMOD, pp: 425-436.
5. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava D., Wu, Y., 2002. Structural Joins: A Primitive for Efficient XML Query Pattern Matching, Proceedings of ICDE, pp: 141-152.
6. Bruno, N., Srivastava, D., Koudas, N., 2002. Holistic Twig Joins: Optimal XML Pattern Matching, Proceedings of ACM SIGMOD, pp: 310-321.
7. Kim, J., Lee, S.H., Kim, H-J., 2004. Efficient structural joins with clusters extents. Information Processing Letters, 91: 69-75.
8. Yao, J.T. and Zhang, M., 2004. A Fast Tree Pattern Matching Algorithm for XML Query, Proceedings of IEEE/WIC/ACM, pp: 235-241.
9. Chen, Q., Lim, A., Ong, K., Tang, J., 2003. D(k)-index: An adaptive structural summary for graph-structured data, Proceedings of SIGMOD, pp: 134–144.
10. Li, Q. and Moon, B., 2001. Indexing and Querying XML Data for Regular Path Expressions, Proceedings. of VLDB, pp: 361-370.
11. Bayardo, R.J., Gruhl, D., Josifovski, V., Myllymaki, J. 2004. An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing, Proceedings of WWW, pp: 345-354.
12. Diao, Y., Fischer, P., Franklin, M., To, R. (2002). YFilter: Efficient and Scalable Filtering of XML Documents, Proceedings of ICDE. Demo paper.
13. Amer-Yahia, S., Cho, S., Lakshmanan, L.V.S., Srivastava, D., 2002. Tree pattern query minimization. VLDB Journal ,11(4): 315-331.
14. Wu, Y., Patel, J. M., Jagadish, H.V., 2003. Structural join order selection for XML query optimization, Proceedings of ICDE, pp : 443-454.
15. Haw, S.C. and Rao, G.S.V.R.K. 2007. A Comparative Study and Benchmarking on XML Parsers, Proceedings of IEEE-ICACT, pp: 321-325.
16. Haw, S.C. and Rao, G.S.V.R.K., 2007. An efficient Path Query Processing support for Parent-Child Relationship in Native XML Databases. Journal of Digital Information Management, 2(2): 82-87.
17. University of Washington XML Repository. Available http://www.cs.washington.edu/research/xmldatasets/