http://ansinet.com/itj

ITJ

ISSN 1812-5638

# INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 10 (9): 1830-1835, 2011
ISSN 1812-5638 / DOI: 10.3923/itj.2011.1830.1835


# A Memory Path Index for DOM Tree Queries

[1]Qing Yang, [2]Huibing Zhang and [2]Jingwei Zhang
[1]Electronic Engineering and Automation Institute, Guilin University of Electronic Technology, Guilin, 541004, China
[2]School of Computer Science and Engineering, Guilin University of Electronic Technology, Guilin, 541004, China

**Abstract:** Large amount of content with similar structures are being integrated into one big XML document, for example, the product records crawled from Web, on which users requires a quick retrieval. DOM tree is a memory model for HTML and XML documents. There are three basic queries on this model, (1) search content located by a given absolute path, (2) search content located by a given relative path, (3) get the path locating the given content. For an XML document with great width and depth, those queries often have a long response time because of time loss on tree traversal. It will be very helpful for Web applications and XML query processing to improve the query efficiency on DOM trees. Present study proposed three index structures to deal with the three basic queries, every of them can give a quick response for corresponding queries with backward search strategy. The experiments on different XML documents show that these index structures can reduce the query time effectively.

**Key words:** Backward search, forward search, absolute path, relative path, XML query

## INTRODUCTION

With rich Web applications, Web pages are hosting more and more valuable content, such as product list pages and micro-blog pages. Web pages are not friendly for data analyzers to access because of their presentation-oriented. Many researchers are trying their best to integrate content on similar Web pages into (W3C, 2008) documents to get neat data sources, on which they can query what they need easily. A big advantage of XML is that it allows users to define flexible document format with their own tags which makes XML documents to be widely used in today's Web applications, for example (Ykhlef, 2009) used XML to represent cube data for OLAP, Wu (2008) replaced relational Database with XML documents to integrate personal information and improve information portability. Users hope to have a quick response on those XML documents though some of them are becoming larger and larger.

Some query languages on XML documents have been proposed, such as Xpath (W3C, 2007), Xquery (W3C, 2010) and Quilt (Chamberlin *et al*., 2000). XPath is a functional subset of XQuery, both of them use path as the query fundamentals, for example (Wang *et al*., 2007) made full use of XPath's characteristics to realize some aggregation functions of XML data. Quilt can group information from multiple data sources in one query. More detailed information about different XML query languages can refer to surveys provided by (Gou and Chirkova, 2007; Nath and Batanov, 2005; Bonifati and Ceri, 2000). Many valuable research on XML query processing are covered by Park *et al.* (2002), Lian *et al.* (2005), Al-Khalifa and Jagadish (2002) and Florescu *et al.* (2000), whose study mostly focus on query optimization. All above work adapt forward search strategy, a search process from root node to leaf nodes which often needs some extra search to assure accurate location.

DOM (W3C, 2004) is a memory model for HTML and XML documents which requires to load the whole HTML and XML documents into memory and to organize them into tree structures. In a DOM tree, every branch node corresponds to a tag in the XML document and every leaf node corresponds to a value. DOM tree provide one-to-one mapping between paths and values. DOM is a compact and popular model for query processing, on which much research have been done for XML queries, for example, Ng and Cheng (2007) and Brenes *et al.* (2008) tried to improve query speed by establishing different indexes, Al-Khalifa *et al.* (2002) provided two structural join methods for XML query processing, Haw and Rao (2007) combined path expressions with index for queries by substructure matching. In fact, it is a permanent work to improve query performance of XML documents on different situations.

**Corresponding Author:** Qing Yang, Electronic Engineering and Automation Institute, Guilin University of Electronic Technology, No. 1, Jinji Road, Qixing District, Guilin, 541004, China Tel: 86-773-2291522

In today's Web-related research, there are often three kinds of queries applied on DOM trees, (1) locate content according to the given absolute path, (2) find content according to the given relative path, (3) give out the corresponding paths according to the given content. These queries do not need complex query language expression but a quick query response can give users good experience. In order to carry out these queries well with limited resources, three index structures are proposed to improve the query efficiency on DOM trees; these indexes are adaptive to XML documents with different width and depth.

## PROBLEM SETTING

A HTML or XML document can be transformed into a tree structure according to the parent-child relationship between the tags. Figure 1 shows a simple DOM tree, in which every circle represents an element node and every rectangle is a text node. Every node can be uniquely identified by its parent and its own tag and index. The index of a node is an integer which begins from 1 and increases from left to right on the same tag with the same parent. A node can be denoted as tag[index], for example, the two nodes in the second level can be denoted as B[1] and B[2]. If dom is a DOM tree, n is a node and p is a path, then the width and depth of a DOM tree can be formalized as:

$$\text{Width (dom)} = \text{Max } \{\text{Childs (n)} | \forall n, n \in \text{dom}\}$$

$$\text{Depth (dom)} = \text{Max } \{\text{Length (p)} | \forall p, p \in \text{dom}\}$$

Childs(n) is a function that denotes the number of n's children, Length(p) denotes the number of p's steps.

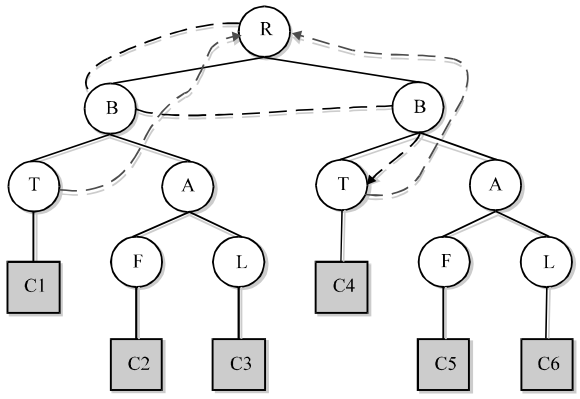


Fig. 1: An illustration of DOM tree( forward search: blue bold line; backward search: red dotted line)

Every node corresponds to one step. A path is a series of ordered steps which are concatenated by "/", for example, "/R [1]/B [2]/T [1]". "//" is used to denote zero or multiple steps. A relative path is a path with "//", otherwise, the path is an absolute path. According to the above description, some functions can be defined to deal with those three kinds of queries,

**Problem definition:** Given a DOM tree dom, p is a path of this tree, C is a piece of text, a query on DOM tree is to find their own counterparts when given p or C. The three basic queries can be formalized as following, absoluteSearch (dom, p): return the content located by p if p is an absolute path. relativeSearch (dom, p): return all content located by p if p is a relative path. Locate (dom, C): return the absolute path which locates content C.

For example, the query results on the DOM tree of Fig. 1 are:

- AbsoluteSearch ("/R [1]/B [2]/T [1]") = {C4}
- RelativeSearch ("/R [1]//T [1]") = {C1, C4}
- Locate (C2) = {"/R [1]/B [1]/ A [1]/F [1]"}

## INDEX STRUCTURES

In this section, three index structures are introduced that can be applied on the above three kinds of queries to improve query efficiency, the individual index structure and search process are presented.

**Index for absolute path search:** Given a DOM tree and an absolute path, the search process often begins from root node, then goes from parent node to child node and reaches the last location according to the given path which is called forward search strategy. The search process of "/R [1]/B [2]/T [1]" is presented by blue bold dotted line. For the above method, if the XML document has a big width, it will need to scan many nodes in the wide level, for example, you must access "B [1]" node before you arrive at "B [2]"node which consumes much time to scan some useless nodes before you stop at the specified node.

In order to avoid scanning useless nodes, the forward search is transformed into backward search by an aided index structure. A backward search begins from the last step of a path which often corresponds to one leaf node and then goes back to the path head along the reversed path.

Unlike forward search, backward search is based on the child-parent relationship. An index holding the child-parent relationship is necessary to realize the backward

search. In this index, every node is represented by its layer, tag and index, such as *layer:tag:index*. Hash structures are used to organize the index. Every node is hashed to its parent so that child-parent relationship can be reserved and every leaf node is also hashed to its value. The basic index structure is shown in Fig. 2, in which every node is mapped to its parent and value by two hash functions.

Algorithm 1: Absolute path search index establishing

**Input:** DOM tree, *dom*
**Output:** Index structure reference

1: Initialize two list structures l1 and l2
2: While ( breadth-first traversal on dom)
3: Get the next node n
4: Get n's parent np
5: Insert (n, np) into l2
6: If (n is a leaf node)
7: Get n's value v
8: Insert (n, v) into l1
9: End if
10: End while
11: Establish two hash structures according to l1 and l2,
v=hs1(n), np=hs2(n)
12: Return hs1, hs2

In order to establish the index, the whole DOM tree will be traversed through breadth-first order, every node is mapped to its parent and its value through two hash functions. Every node is characterized by its layer, tag and index, a hash function is used to hold their child-parent relationships. In a DOM tree, every node has only one parent except root node which assures the efficiency of backward search. The detailed process to establish indexes is presented in algorithm 1, an accompanying example is shown in Fig. 3.

When searching the content by a given absolute path, the path is firstly transformed into triple form (layer: tag: index) and then are reversed. A backward search process is executed on the index structure, for every step in the reversed path, hash function is used to find its parent and then verify whether the hashed parent node and the real parent node in given path are consistent, if they are not same, the search process will stop, otherwise, the search process continues until reaching root node. Our index has two advantages to improve search efficiency, one is that the child-parent relationship is exploited which is one-to-one mapping, more compact than the one-to-many mapping in parent-child relationship, the other is that our indexes keep nodes' index information directly which must be obtained through a count process in original DOM tree. The above two improvements help to reduce greatly the search time than forward search on parent-child relationship. Because of hash structure, hash conflicts still exist in our index structure but it does not influence the search results for the uniqueness of complete absolute paths. The time complexity of backward search only depends on the length of the given path and is not affected by document width. The detailed search process is presented in Algorithm 2.

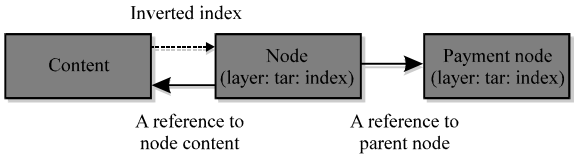


Fig. 2: Index structure (graph lines denote absolute search index, dotted line denotes inverted content index)

Algorithm 2: Absolute path search

**Input:** DOM tree dom, absolute path p
**Output:** text content C

1: Transform every step of p into the form,
"layer:tag:index" and reverse these steps into rp
2: Fetch the first step of rp into leafNode
3: Fetch the last step of rp into rootNode
4: While (i<rp. length-1)
5: fNode = the $i_{th}$ step of rp
6: cNode = the $(i+1)_{th}$ step of rp
7: If (cNode not in hs1 (fNode))
8: Return null
9: End if
10: End while
11: If (cNode! = rootNode)
12: Return null
13: End if
14: C = hs2(leafNode)
15: Return C

**Index for relative path search:** The significant difference between relative paths and absolute paths is that relative paths permit fuzzy match by "//". "//" represents zero or multiple steps which causes a relative path to be mapped

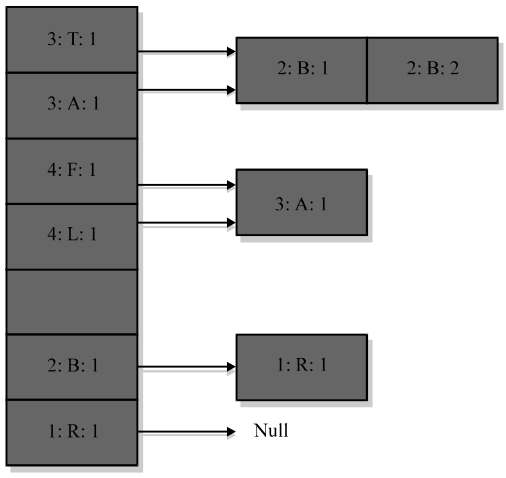


Fig. 3: A part of index of the DOM tree in Fig. 1

to zero or multiple absolute paths and has a great influence on queries on XML documents with big depth. Since the layer information is uncertain, the exact index for absolute path search is invalid for relative path search. The time complexity of forward search on a relative path is just the tree's traversal complexity. For the relative path can have zero or multiple instances, forward search can only traverse the whole DOM tree to find all matching paths with it.

In order to improve the time complexity of relative path search, the index structure for absolute path search is adjusted to adapt relative path search. In the new index structure, every node is represented by "tag:index" and then are hashed to its parent with the same representation. In this index, child-parent relationships are possible to be one-to-many mapping since layer information is lost. Obviously, the index structure introduces some conflicts because of the incompleteness of node representation, some extra search operations are needed to eliminate conflicts but the search space of backward search is still less than the forward search.

The query strategy for relative path search is that all possible absolute paths matching with the relative path are constructed through accessing the index for relative path search, then remove those illegal paths and use filtered absolute paths to locate content under the help of absolute path index. When query processing, the given path is still transformed into a series of steps, tag: index and then all steps are reversed. The backward search is carried out to combine all possible paths which means that "//" are instantiated by one or multiple steps according to the child-parent information held in indexes. Because some extra paths can be generated for hash conflicts, a final path verification is executed to assure the accuracy of queries through the absolute path index. All confirmed absolute paths are then used to locate their corresponding content. The index for relative path search can avoid the whole tree traversal to find all answers for a given relative path.

**Inverted index for content:** The third type of queries are time-consuming since only traversal on the whole DOM tree can be used to find all matching paths for a given content, just like the forward search solution for a relative path. Here, the inverted index is introduced to speed up those queries. Every value is mapped to its corresponding leaf node through a hash function. When to get the path of a given value, the hash function will be applied on this value and get the leaf node, a backward process on DOM tree can easily get the corresponding path. In fact, fuzzy queries can also be supported through establishing some indexes for hot terms.

## EXPERIMENTS

In this section, experiments on XML documents with different width and depth are carried out to evaluate the query performance on the index structures. All experiments are run under Core Duo 2.2 GHz CPU and 2G memory.

**Absolute path search on XML documents with different width:** The index for absolute path search is very suitable for absolute path queries on XML document with large width. A group of XML documents are constructed whose width varies from 100 to 3000. The 10000 queries are randomly choosed on those documents and the query time are compared between forward search on DOM tree and backward search on the index. The experimental results are reported in Fig. 4. The index takes on an excellent performance for these queries because it avoids a large number of invalid search. The invalid search in our method is only caused by hash conflicts which has been reduced greatly by the accurate child-parent relationship representation. Because of the child-parent relationship and backward search, the search time is hardly influenced by the document width which is a primary complexity factor for forward search. The time complexity of backward search on indexes only depends on the average length of query paths but the complexity of forward search are decided by both the average width of DOM tree and the average length of paths. With the increase of document width, our method has a good scalability.

**Path locating by inverted index:** In this group of experiments, the above dataset is still used and 10000 values are generated randomly for test which are used to find all paths locating the given values and compute the search time. The query time comparison is presented in Fig. 5. Inverted index does not show advantage on XML

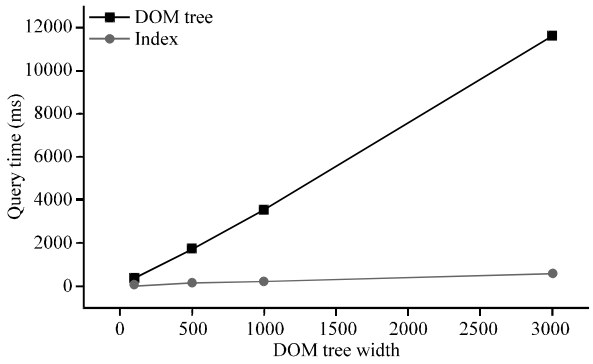


Fig. 4: Query time comparison for absolute path search

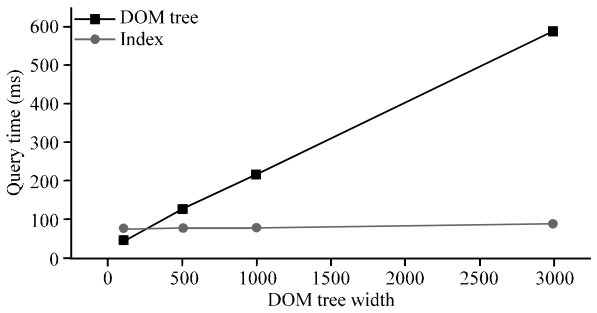


Fig. 5: Query time comparison for content search

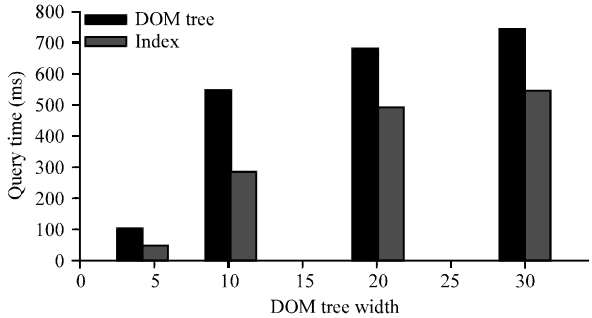


Fig. 6: Query time comparison for relative path search

documents with small width. With the increasing document width, the time on tree traversal shows a great influence on search performance because there is not any proof to tell us that all paths related with the given content have been found before reaching the end of DOM tree. The inverted index has a steady performance which are not constrained by the document width. The time complexity of query on inverted index are only decided by the average length of searched paths but the time complexity of forward search is proportional to the scale of DOM trees because of traversal requirements.

**Relative path search on XML documents with different depth:** The index for relative path search has a big advantage for relative path queries on XML documents with great depth. Another group of XML documents are constructed whose depth varies from 4 to 30. These documents are used as test data and 1000 relative paths in those documents are generated randomly for query time evaluation. Those paths cover different relative path characteristics, such as "a [2]//b [1]", "//a [1]", "//a [2]/b [3]//c [2]" and so on. All paths are applied on the relative path index and use backward search to query their corresponding values, the query time is compared with the direct query on DOM tree, the comparison is presented in Fig. 6. The relative path index and backward search reduce the query time effectively than the direct query on DOM tree. Because of lack of nodes' index information and hash conflicts, some invalid paths will be generated when instantiating those relative paths, a disambiguation process must be carried out to remove those false paths, in fact, most of query time are consumed on the generation of the invalid paths and identifying those false paths. The direct query time complexity on DOM tree is decided by the size of DOM tree since the traversal on the whole tree is the only way to find all matching paths with the given relative path, the index can reduce the search space effectively.

## CONCLUSIONS AND FUTURE WORK

In present study, three index structures are proposed to improve the query performance on DOM trees. Based on those index structures, a backward search process could deal with less search space than the forward search which token on great advantages especially for those XML documents with large width and depth. A big advantage of the indexes was that the nodes' index information was embedded into the tree index explicitly which could avoid some invalid search effectively for absolute path search and relative path search. The backward search made full use of the child-parent relationship, a one-to-one mapping and showed better performance than forward search which must compute the node index through traversal operations. The index for relative path search still has some conflicts because of the uncertainty of nodes' index information which maybe need some other measures to reduce those conflicts and improve query performance further.

## ACKNOWLEDGMENTS

We gratefully acknowledge the support of Education Department Foundation of Guangxi under grants No. 201010LX154.

## REFERENCES

Al-Khalifa, S. and H.V. Jagadish, 2002. Multi-level operator combination in XML query processing. Proceedings of the eleventh international conference on Information and knowledge management, (CIKM'02), ACM New York, NY, USA., pp: 134-141.

Al-Khalifa, S. H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava and Y. Wu, 2002. Structural joins: A primitive for efficient XML query pattern matching. Proceedings of the 18th International Conference on Data Engineering, Feb 26, San Jose, CA, USA., pp: 141-152.

Bonifati, A. and S. Ceri, 2000. Comparative analysis of five XML query languages. ACMSIGMOD Rec., 29: 68-79.

Brenes, S., Y. Wu, D. Van Gucht and P.S. Cruz, 2008. Trie indexes for efficient XML query evaluation. Proceedings of the 11th International Workshop on Web and Databases, June 13, Vancouver, BC, Canada, pp: 1-6.

Chamberlin, D., J. Robie and D. Florescu, 2000. Quilt: An XML query language for heterogeneous data sources. Proceeding of the Third International Workshop WebDB 2000 on the World Wide Web and Databases, (TIWWWWD'00), Springer-Verlag, London, UK, pp: 1-25.

Florescu, D., D. Kossmann and I. Manolescu, 2000. Integrating keyword search into XML query processing. Comput. Networks, 33: 119-135.

Gou, G. and R. Chirkova, 2007. Efficiently querying large XML data repositories: A Survey. Transact. Knowledge Data Engineer, 19: 1381-1403.

Haw, S.C. and G.S.V.R.K. Rao, 2007. Path query processing in large-scale XML databases. J. Applied Sci., 7: 2736-2743.

Lian, W., N. Mamoulist, David W.L. Cheung and S.M. Yiu, 2005. Indexing useful structural patterns for XML query processing. IEEE Trans. Knowledge Data Eng., 17: 997-1009.

Nath, U.K.D. and D.N. Batanov, 2005. Comparative analysis of three promising XML query languages and some recommendations. Inform. Technol. J., 4: 439-444.

Ng, W. and J. Cheng, 2007. An efficient index lattice for XML query evaluation. Proceedings of the 12th International Conference on Database Systems for Advanced Applications, (DASFAA?07), Springer-Verlag, Berlin, Heidelberg, pp: 753-767.

Park, S., Y. Choi and H.J. Kim, 2002. XML Query processing using signature and DTD. Proceedings of the Third International Conference on E-Commerce and Web Technologies, (EC-WEB'02), Springer-Verlag, London, UK, pp: 162-171.

W3C, 2004. Document Object Model (DOM) Level 3 Core Specification (1.0) W3C Recommendation. http://www.w3.org/DOM/

W3C, 2007. XML Path Language (XPath) 2.0 W3C Recommendation. http://www.w3.org/TR/2007/REC-xpath20-20070123/

W3C, 2008. Extensible Markup Language (XML) 1.0 W3C Recommendation. http://www.w3.org/TR/2008/REC-xml-20081126/.

W3C, 2010. XQuery 1.0: An XML Query Language W3C Recommendation. http://www.w3.org/TR/xquery/.

Wang, H., J. Li and H. Gao, 2007. Flexible and effective aggregation operator for XML data. Inform. Technol. J., 6: 697-703.

Wu, C.F., 2008. Design of portable personal information management system with XML technique. Inform. Technol. J., 7: 615-622.

Ykhlef, M., 2009. On-Line Analytical Processing Queries for eXtensible Mark-up Language. Inform. Technol. J., 8: 521-528.