An Efficient XML Index for Keyword Query with Semantic Path in Database

Yanzhong Jin
Computer Science & Technology Department, Tianjin University of Science & Technology, Tianjin, China
Email: zongyj@pku.org.cn

Xiaoyuan Bao
Computer Science & Technology Department, Peking University, Beijing, China
Tianjin Normal University, Tianjin, China
Email: xybao@pku.edu.cn

***Abstract*—with the wide adoption of XML in many applications, people begin to manage thousands of XML documents in database. In many applications which backend data source powered by a XML database management system, keyword search is important to query XML data with a regular structure if the user does not know the structure or only knows the structure partially. Essentially, many keyword search can be rewritten to XPath query Q=[//|/]e_1[//|/]e_2[//|/]…[//|/]e_m[*text()=str*]-suppose there is a keyword search [*books William*] on XML data about publishing, the result could be the union of the results of the two queries after database system rewriting based on meta data: //*books*//*chapters*//*authors*[*text()="William"*] and //*books*//*authors*[*text()="William"*]. We propose an XML index structure BTP-Index, composed of XML structure index mechanism which backbone is a Suffix tree, for evaluation of path ([//|/]e_1[//|/]e_2[//|/]…[//|/]e_m) of Q, and XML content index mechanism which is based on Tries & Patricia tree, for the evaluation of [*text()=str*], filtering part of query Q. Using BTP-Index, we can process query Q efficiently. We have proven the effectiveness of BTP index in our Relation-XML dual engine database management system.**

***Index Terms*—XML, Suffix Tree, Index, XPath**

I. INTRODUCTION

With the wide adoption of XML in many applications, people begin to manage thousands of XML documents in database. In many applications which backend data source powered by a XML database management system, keyword search with semantic constraint is important to query XML data with a regular structure if the user does not know the structure or only knows the structure partially. Essentially, many keywords search have the form Q=[//|/]e_1[//|/]e_2[//|/]…[//|/]e_m[*text()=str*] from the perspective of XPath[9]. For example, suppose there is a keyword search [*books William*] on XML data about publishing, the result could be the union of the results of the two queries after database system rewriting based on meta data: //*books*//*chapters*//*authors*[*text()="William"*] and //*books*//*authors*[*text()="William"*]. We focus this paper on index mechanism on which query Q will be evaluated efficiently in XML database. Query rewrites and optimization is beyond this paper.

As for query Q, it is to find all the $element_m$ which has the text content *str*, and its ancestor/parent element is $element_{m-1}$, which in turn has the ancestor/parent element $element_{m-2}$, and so on. In the following, we use notation Q_s as the shortcut of the path [//|/]e_1[//|/]e_2[//|/]…[//|/]e_m of the query, and Q_c for [*text()=string*], for simplicity.

As for the efficient query evaluation in database system, constructing indexes for the data over which query will be performed is a classical and effective idea. In database research and engineering fields, many XML indexes have been proposed over the last decade. Some representative index structure such as Structure Join based Index[1,2,3], Path based Index[4,5], APEX[6] and ViST[7], Graph Indexing[8], etc, have been proposed in recent years.

To our best knowledge, there is no solution for the evaluation of query Q, especially Q_s at O(m) cost which m is the length of Q_s.

A. Related works

In structure join-based index [1], Quanzhong Li et al. proposed a new system for indexing and storing XML data based on a numbering scheme for elements. This numbering scheme quickly determines the ancestor-descendant relationship between elements in the hierarchy of XML data. Reference [2] proposed a variation of the traditional merge join algorithm, called the multi-predicate merge join (MPMGJN) algorithm, for finding all occurrences of the basic structural relationships (such as containment queries). Similarly, ref. [3] developed two families of structure join algorithms tree-merge and stack-tree, for determination of ancestor-descendant relationships.

As for Path-based Index[4,5], DataGuides[4] is the structural summaries of the source XML data, and it can be used to find elements when their full path (path from root element) is given. But for some XML data, the index volume may be bigger than the source data. In ref. [5], BF. Cooper et al. proposed an Index Fabric, it is conceptually similar to the DataGuides in that it indexes all raw paths starting from the root element.

APEX[6] is an adaptive path index for XML data. Unlike the traditional techniques, APEX uses data mining algorithms to summarize paths that appear frequently in the query workload. It maintains every path of length two,

doi:10.4304/jsw.5.10.1052-1059

therefore it also has to rely on join operations to answer path queries with more than two elements.

ViST[7] has proposed a method for indexing XML data based on pre-sequencing XML data, so query evaluation is equivalent to the sequence matching.

Xifeng Yan et al. proposed in ref. [8] a graph mining technique, different from the existing path-based methods, a gIndex was proposed to make use of frequent substructures as the basic indexing feature.

Recently, ref. [15] considers indexing support for queries that combine keywords and structure, it described several extensions to inverted lists to capture structure when it is present.

As for keyword search in XML data, it is a hot database research topic in nowadays, ref. [16~25] are samples of these; ref. [16] proposed an extension to XML query languages that enables keyword search at the granularity of XML elements; ref. [17] considered the problem of efficiently producing ranked results for keyword search queries over hyperlinked XML documents, etc.

All these proposed index structures or methods cannot process Q_s with keywords efficiently, and we don't find practical methods which incorporating database management system can be used in our application example presented above, this motivates our work on BTP-index in a real database management system.

B. Related works

For evaluating query Q efficiently, we propose an XML index structure BTP-Index. In particular, the contributions of our paper can be described as follows:

We propose Suffix tree based XML structure index mechanism (B part of BTP-Index). Using this index mechanism, we can process the Basic Path Query Unit (see definition 1) of Q at time expense of O(h).

We propose an algorithm for processing Q_s by join the Basic Path Query Unit result, based on an extended code mechanism for XML data tree.

We propose an XML content index structure based on Tries & Patricia[12] tree (TP part of BTP-Index). Each leaf node in the index tree corresponds to a word in XML content, and each item of the attached inverted list to the node contains position information of the word. And the worst evaluation cost of Q_c is $O(|str|+k log_B(|L|))$. Put above structures together, we call the overval mechanism as BTP-Index.*

We have implemented part of the methods in our Relation-XML dual engine DBMS system, and our experimental result perform in the system demonstrates the efficiency of BTP-Index.

The rest of this paper is organized as follows: Some related preliminary knowledge is introduced in section 2. In section 3, we propose the suffix based XML index mechanism and the algorithm joining *Basic Path Query Unit* efficiently, for evaluation of Q_s. Section 4 proposes an XML content index structure for evaluation of Q_c. In section 5, experimental results and brief analysis are given. Section 6 concludes the whole paper.

II. PRELIMINARIES

In this section, we introduce XPath and *basic path query unit* concept in II.*A*, then present the XML data model in II.*B*, and finally define Suffix tree and give some lemmas about it in II.*C*.

A. Basic Path Query Unit

Definition 1. *Basic Path Query Unit*

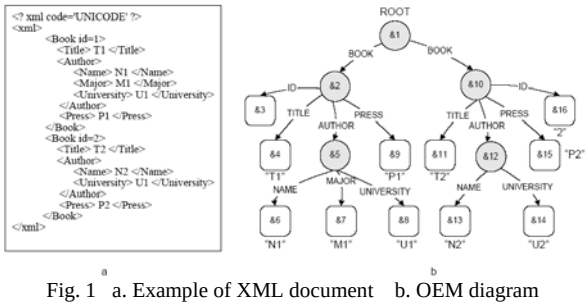
We call any XPath query of form $//e_1/e_2/\ldots/e_h$ as a *Basic Path Query Unit* of query *Q*.We will use BPQU as a shortcut for it.

Our overall idea to process structural part of query Q is to decompose the structural part of query Q into many BPQU, and processing each BPQU respectively first, then join the results of BPQU's to get the final result of structural query part of *Q*.

Please note that the "structural part of query" has the same meaning with "semantic path of query" in our paper, we will use it interchangeable in the paper followed. Its notation Q_s is also used in text sometimes.

B. XML data model

We use a semi-structured data model called Object Exchange Model[13] (OEM) to describe the content of XML document. A diagram is used to represent the data. In this diagram, nodes denote the objects and edges are tagged by attribute names. There are two kinds of OEM objects, Atomic object and Complex object. The value of Atomic object is undividable, ex. an integer, while the value of Complex object is a set of <label, id> pairs.



a b

Fig. 1 a. Example of XML document b. OEM diagram

In OEM, XML data can be expressed as follows: OEM nodes denote XML elements and the relationship like parent-child, element-attribute and references are denoted by labeled edges. The data value (Suppose all the data values are string) corresponds to OEM leaves.

Fig. 1a shows the example of XML document in this paper and Fig. 1b is the corresponding OEM diagram. &0 and &1 are identifiers of elements. &13 and &14 are Atomic objects while &1 and &2 are Complex objects.

C. Suffix Tree

Definition 2. String suffix

Given a set Σ of alphabet, $s \in \sum^+ \cup \{\varepsilon\}$; we call string *P* is the suffix of *S*, *iif*($P=S[1,i]$, $i=1,2,\ldots,|S|$, $|S|$ is the length of S. Especially, empty string ε is also the suffix of S. In fact, ε is the suffix of any string.

Definition 3. String containment ($\subseteq$)

Given two strings $S^1[1..m]$, $S^2[1..n] \in \sum^+$, $m \leqslant n$. If $S^1[1]= S^2[i]$, $S^1[2]= S^2[i+1],\ldots$, $S^1[m]= S^2[i+m-$

1],$1 \leqslant i \leqslant n-m+1$, then we say that S^1 contained in S^2 (notation is $S^1 \subseteq S^2$).

Definition 4. String suffix tree (T_S)

The suffix tree T of string S is a rooted tree. Formally, $T=\{V,root,E,L,\sum\}$, among which, root is the root node; V is the set of all nodes in T; E is the set of all labeled edges in T; L is the set of all leaf nodes and $L \subset V$. $\forall n \in V$, there is a sequence $l_1 l_2 ... l_i$, which we called the *path* of n. Actually, $l_1 l_2 ... l_i$ is the concatenation of edge labels along the *path* from root node to n. For each node $n \in V$, its *path* is unique. And for each leaf node $m \in L$, the edge path of m is one suffix of S. Fig. 2 is an example suffix tree of "ababc".

Definition 5. Suffix tree containment

Given $T=\{V,root,E,T,\sum\}$ and $T'=\{V',root',E',T',\sum'\}$, if $V \subseteq V' \wedge E \subseteq E' \wedge \sum \subseteq \sum'$, we say that T is contained in T' ($T \subseteq T'$).

Based on the definitions above and suffix tree construction algorithm[11], we have the following propositions.

Lemma 1. The judgment of containment of string p in s can be implemented as the process of searching in suffix tree of s character by character from the root node, the time expense is $o(m)$, where m is the length of p.

Lemma 2. Give string s1, s2, if s1⊆s2, we have $T_{s1} \subseteq T_{s2}$.

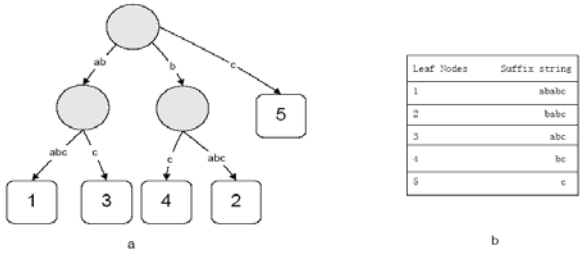


Fig. 2 Suffix tree of string "ababc"

III. XML STRUCTURE INDEX MECHANISM FOR EVALUATION OF BPQU

In this section, firstly we discuss the construction of XML structure index (*B part of BTP-Index)* for evaluation of *basic path query unit* in detail (III.*A*)(so we named it BPQU-Index), after that we analysis the query efficiency based on it (III.*B*) and (III.*C*) give the steps for evaluation of semantic path query of Q.

A. Index Construction

Suppose there is an XML tree $T_o(V_o,root_o,E_o,T_o,\sum_o)$ (as Fig. 1b) where $\sum_o$ is the set of labels of all the edges of the tree. For Fig. 1, $\sum_o$ ={ book, title, author, name, major, university, press}. Since all XML documents use "xml" as the root element, we don't make it an element of $\sum_o$. The path of nodes in T_o, for example, "book.author.name" of node "&6" and "book.author" of internal node "&5", are path strings based on $\sum_o$.

We found that node &4 and &11 are the same in semantics but different in contents. In order to represent the nodes of the same semantic, we define two kinds of path: data path and semantic path. Data path is a string of format "$l_1 d_1 l_2 d_2 \ldots l_i d_i$", where l_k (k=1,2,…,i) is the tag of edge in OEM and d_k (k=1,…,i) is the identifier of node, ex. &2 and &10. Semantic path is a string of format "$l_1 l_2 \ldots l_i$", where l_k (k=1,2,…,i) is the tag of edge in OEM. Obviously a semantic path can have several data path. l_k and d_k are basic elements of the following algorithms like the single character of processing the string.

The basic idea of constructing the suffix tree of XML semantic path is merging the data nodes with the same semantics and finding the node within limited steps. We construct a suffix tree by merging all the suffix strings of the semantic path of each node, i.e. by sharing the common suffix of the semantic path. This suffix tree is called *BPQU-Index* of OEM diagram or of corresponding XML data. The OEM node in each node of *BPQU-Index* called the extending set. In Fig. 3b, the extending set of *BPQU-Index* node corresponding to path "book" is {&2, &10}. Then we will give the formal definition of *BPQU-Index*:

Definition 6. *BPQU-Index*

Suppose $\sigma(T_o)$ ={$s_1,s_2,\ldots,s_L$} is the set of data path strings of all the nodes in T_o., for each s_i(i=1,2,…L), we use its semantic path to construct the *BPQU-Index* tree $T_{\sigma(To)}$ ={ V_{suff}, $root_{suff}$, T_{suff}, E_{suff}, $\sum_o$, F} , where V_{suff}, $root_{suff}$, T_{suff}, E_{suff} correspond to node set, root node, leaf nodes and edge set. $\sum_o$ comes from T_o. $F(v')=\{v | v \in V_o \wedge v' \in V_{suff} \wedge v_{path}=v'_{path}$, where v_{path}, v'_{path} are the semantic path of node v and v'.

Since the path of internal nodes are substrings of leaf nodes', from Lemma 2 we know that we can just building the *BPQU-Index* for all the leaf nodes in OEM diagram. In other words, $\sigma(T_o)$ only contains the path of leaf nodes, i.e. $\sigma(T_o)=\{s_1,s_2,\ldots,s_L\}$ where L is the number of leaf nodes.

The algorithm of building $s_i = l_{i1} d_{i1} l_{i2} d_{i2} \ldots l_{ih} d_{ih}$ in T_{i-1} is given in algorithm 1, in which T_{i-1} is the current *BPQU-Index* tree and root is the root node. Algorithm 1 should be applied L times.

Algorithm 1: Building *BPQU-Index* of $s_i = l_{i1} d_{i1} l_{i2} d_{i2} \ldots l_{ih} d_{ih}$ in T_{i-1}

1 Set semantic path $p \leftarrow l_{i1} l_{i2} \ldots l_{ih}$. It's also the suffix of p. Set data string $q \leftarrow d_{i1} d_{i2} \ldots d_{ih}$;
2 Searching for edge p[1] from root node. If it exists, put q[1] to the extending set of the node p[1] points to in T_{i-1};
3 Continue the searching process in 2 till there are no edges matched. Suppose the last matching edge is p[j], and the node it points to is M, $k \leftarrow j$, goto 4;
4 $k \leftarrow k+1$. If $k \leqslant |p|$, then goto 5, else goto 6;
5 Create a new node N. Put d[k] into the extending set of N. Point p[k] to N from M. Goto 4;
6 If $|p| \leqslant 1$ then goto 7, else $p \leftarrow p[2..|p|]$, $q \leftarrow q[2..|q|]$, goto 2;
7 End of procedure.

An example is given to explain Algorithm.

Suppose σ = {book.&2.author.&5, book.&10.author. &12}. First construct *BPQU-Index* for string "book.&2. author.&5" whose semantic path is "book.author". "book. author" has two suffixes: "book.author" and "author".

Then,

a). Apply algorithm to suffix "book.author". Because *BPQU-Index* is empty at the beginning, we

construct root node first. Then create node "&2", edge "book", node "&5" and edge "author";

b). Apply algorithm to suffix "author". Because the root node doesn't have an edge labeled "author", we create edge "author" and node "&5". Fig. 3a shows the resulting *BPQU-Index*. Then process "book.&10.author. &12" based on the *BPQU-Index* now;

c). Apply algorithm to suffix "book.author". Edge "book" is found from the root node and it points to node "&2". Merge "&10" into it and get {&2,&10}. Then edge "author" is found to match the following edge. Add "&12" into the node which "author" points to and get {&5,&12};

d). Apply algorithm to suffix "book.author". Similar to *c)*, "&12" is merged into node which "author" points to and get {&5,&12}. The resulting *BPQU-Index* is shown in Fig. 3b.

In Fig. 1b, because most of the data path of node "&6" and "&7" are overlapped, if first adding the suffix of path of node "&6" as algorithm 1, the constructing of suffix of node "&7" contributes little to the extending set of the corresponding path. Actually, for leaf nodes in OEM diagram, if some of the data paths are overlapped, there will be some redundant work for algorithm 1. The time complexity of algorithm 1 is $O(h^2)$.

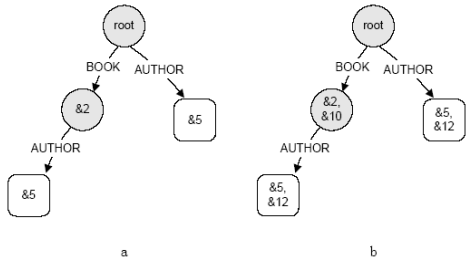


Fig 3. a. *BPQU-Index* of "book.&2.author.&5"
b.*BPQU-Index* after processing "book.&10.author.&12"

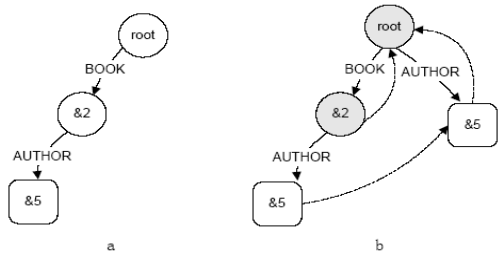


Fig. 4 a. data Diagram of book.&2.author.&5
b. *BPQU-Index* of book.&2.author.&5

Another constructing algorithm based on pre-order traveling is proposed to improve the efficiency of constructing *BPQU-Index*. Fig. 4 is used to explain the main idea of the new algorithm: Visit edge "book" and node "&2" from root, build *BPQU-Index* for "book.&2". Then point *p*' to the leaf node with the longest path (Use the number of nodes to measure the length of the path) in *BPQU-Index*. Continue traveling, after visiting edge "author" and node "&5", add them to the root node and the node which *p*' points to. These steps construct the *BPQU-Index* of path "book.&2.author.&5". It's the same with algorithm 1.

The precondition of this method is that as for semantic path "book.author", its *BPQU-Index* can be made by modifying the *BPQU-Index* of "book". The modification can be done by adding edge "author" and the node "author" points to the leaf nodes and root in *BPQU-Index* of "book". For Fig. 4b, the shadowed nodes are *BPQU-Index* of "book", the other two nodes are added based on it.

Whether we can find the node of *BPQU-Index* to be operated on is critical for the efficiency of construction. So we introduce Suffix Link. Suppose in *BPQU-Index*, the path of node v is $x\alpha$, where x is the basic element of the path (ex. "book" and "author" of "book.author") and α is the substring of the path. If the path of node $s(v)$ is α there is a pointer points to $s(v)$ from v. It is called Suffix Link of node v.

Lemma 3. In *BPQU-Index*, each node has a Suffix Link points to its suffix node. Please notice that the Suffix Links of all the child nodes of root point to the root node. In other words, the root node stands for the null suffix of all the strings. The suffix of root is null.

Fig. 4 is used to demonstrate the construction of *BPQU-Index* with Suffix Link. It is also an explanation of Lemma 3.

a). *BPQU-Index* is empty initially. There is only a root node tagged "root". Point *p*' to it. Travel OEM tree from the root node. Visit the edge "book" and node "&2". Create node "&2" in *BPQU-Index* and add an edge "book" to the node which *p*' points to, point the edge from "root" to "&2". Because "&2" is the child of root, it should have a Suffix Link points to root. We point *p*' to "&2".

b). Continue traveling, visit "author" and node "&5". Create edge "author" and node A with an extending set of {&5} in *BPQU-Index*, point "author" from {&2} (which *p*' points to) to A. If the node which *p*' points to has a Suffix Link, then visit the node N which the Suffix Link points to (It's the root in Fig. 4). Create an edge "author" and a node B with an extending set of {&5}. Point "author" from N to {&5} and point the Suffix Link of A to B. If node N has a Suffix Link points to the next node, mark that node and use the similar procedure of create edge, nodes and set the Suffix Links. Repeat these steps until the Suffix Link of the current node points to the root (as in the example).

c). Point *p*' to node B which is the *BPQU-Index* node corresponding to the path ("book.author") of current visited node.

According to the above constructing process, each node in *BPQU-Index* has a Suffix Link pointing to a corresponding node except the root node. The motivations of introducing Suffix Link is that when edge l and node n are visited, we just need to add them to the nodes which are in the Suffix Link of the node which *p*' points to. The constructing efficiency is improved by removing the procedure of searching and matching.

Algorithm 2 constructs *BPQU-Index* based on pre-order traveling.

Algorithm 2 OEM Pre Travel based XML *BPQU-Index* Construction
begin
Input： *XML Oem Tree*
Output： *BPQU-Index Tree*
1 p←OEMTree.root
2 Create *BPQU-Index*.root; *p*'←*BPQU-Index*.root
3 **While p<>null do**
4 p←p.nextNode, p's semantic path string is *w* //in pre-travel order
5 *n* ←p.nodevalue, *l*←p.edgelabel, A←Null
6 q←*p*' ←Node in *BPQU-Index* which semantic path m string is *w-l*
7 **While q.sufflink<>Null do**
8 **Call** Create_Or_Find_ Node_B
9 **If A<>Null then**
10 A.sufflink←B
11 **End If**
12 A←B; q←q.sufflink
13 **End of while**
14 **Call** Create_Or_Find_ Node_B
15 B.sufflink←*BPQU-Index*.root
16 **If A<>Null then** //A memorize the previous created or found node B
17 A.sufflink←B
18 **End If**
19 **End of while**
20 **Procedure Create_Or_Find_ Node_B**
21 **If not exists node(edge=l)** ∉ **q.child then**
//create a node with edge l points to it
22 CreateNode B(edge=*l*,extent={*n*})
23 q.child←q.child∪B
24 **Else**
25 B←q.child(edge=*l*) //B as node with edge l points to it from q
26 B.extent←B.extent∪{*n*}
27 **End If**
28 **End of Procedure**

In algorithm 2, the subprogram of line 20-28 examines whether there is a node with edge l in the child node set of the node which q points to. If it exists, put the node value n into its extending set. Otherwise, create node B and point edge l to B from node q. The procedure of traveling the OEM tree and constructing the *BPQU-Index* is shown from line 4 to line 19. The next data node n and edge l of OEM tree are visited (line 4-5). Point q to the node corresponding to the longest suffix (itself) of the previous node in *BPQU-Index* (line 7-13), and call the subprogram (line 20-28) to insert a node <edge=l, nodevalue=n> to *BPQU-Index*. Then call the subprogram (line 20-28) in turn with each node in the Suffix Link of q. Each time of inserting the node in the above procedure except the first time, the Suffix Link of the node added the previous time should be pointed to the latest added node, in order to make sure the semantic path α of node B which A's *BPQU-Index* points to is the suffix of semantic path *x*α of A. The situation that q points to the root of *BPQU-Index* is managed in line 14-18. Suppose the semantic path of node <edge=l, nodevalue=n> is *w*, *p*' points to a node whose semantic path is *w-l* in the *BPQU-Index* (line 6). This is obvious because if *w-l* is an empty string *ε*, *p*' points to the root of *BPQU-Index*. And this node will be the node where the construction of *BPQU-Index* begins with at the next step.

When using algorithm 2 to construct *BPQU-Index*, if node B already exists in the *BPQU-Index* and B's semantic path is the semantic path or its suffix of node <edge=l, nodevalue=n> in OEM tree, there is no need to operate the Suffix Link of B again. This process isn't given in the algorithm for the continuity of logic.

The time complexity of algorithm 2 is $O(N^2)$ where N is the number of nodes in OEM tree. In the implementation, we can improve the algorithm by merging the nodes with the same content. For some given data, the time complexity of the improved algorithm can be $O(N)$ or even less, when the number of nodes with the same semantic path is big.

B. Query evaluation of BPQU

In fact, the following result is true for *BPQU-Index*:

Theorem 1: For queries of format $//e_1/e_2/\ldots/e_h$, the query can be processed by searching for the node with semantic path "$e_1.e_2\ldots.e_h$". If such node exists, its extending set must be the query result.

Proof: Known from algorithm 2, there exists a suffix tree in *BPQU-Index* for each node *v* (sPath,&x) in OEM tree. All the nodes with the same semantic path have the same suffix tree. According to the construction of extending set in *BPQU-Index*, the extending set "extent" of each node *A*(sPath=*w*, extentSet) contains all the nodes in OEM tree with the same semantic path *zw* ($|z|\geqslant 0$) . The characteristic of suffix tree tells that there is only one semantic path *w* in *BPQU-Index*. The theorem can be proved by setting $w=e_1.e_2\ldots.e_h$.

In the implementation, the time complexity of processing query on each node can be O(1), if the child nodes are searched using Hash table. So the total time expense of processing "$//e_1/e_2/\ldots/e_h$" is $O(h)$.

C. Query evaluation of Semantic path query (Q_s*) in* Q

As we have mentioned in section 2.1, the evaluation of Q_s can be fulfilled as following steps:

a).Decompose Q_s into many *BPQU* queries. As for *[//|/]*element$_1$*[//|/]*element$_2$*[//|/]…[//|/]*element$_m$, we suppose that the resulting BPQUs are //element$_1$/…/element$_{i1}$, /element$_{i1+1}$/…/element$_{i2}$, //element$_{i2+1}$/…/element$_{i3}$, …, //element$_{ij-1+1}$/… /element$_{ij}$, where $i_j=m\geqslant i_{j-1}\geqslant\ldots\geqslant i_2\geqslant i_1\geqslant 1$ That is to say, there are *j* BPQUs in Q_s.

b).Processing each BPQU respectively, suppose we the resulting sets are R_1, R_2, …, R_j

c).Join R_1, R_2, …, R_j with the condition $e_1//e_2//e_3//\ldots//e_i$ where $e_i\in R_i$, i=1,2,…,j. Suppose we get the final result of query Q_s as R_s

d).Output the R_s as the semantic query of *Q*.

In order to carry out step *c)* efficiently, we use the simple region code for start and end tag of XML elements. That is to say, there is a pair (*start*, *end*) attribute attached with each node in XML data tree (see Fig. 1b). For every node N($start_N$, end_N), each code pair ($start_d$, end_d) of its descendant elements must be contained in ($start_N$, end_N), i.e. $start_d>start_N$ and $end_d<end_N$. Based on this technique, we can use MPMGJN[2] to process the join of R_1, R_2, …, R_j with the condition $e_1//e_2//e_3//\ldots//e_i$ ($e_i\in R_i$, i=1,2,…,j) efficiently, please reference detail in Ref. [2].

IV. XML CONTENT INDEX STRUCTURE

After we have R_s in our hand, how to find *string* quickly is the problem we have to solve next.

A naïve method is like this: let $R_s=\{e_1,e_2,\ldots e_k\}$ be the result of query Q_s, then for each e_i(i=1,2,…,k), we have to search in the XML tree to find whether the content of the element is the *string*. If it is, then e_i is the element we are looking for. Using this method, for each e_i, we may have to read in the corresponding part of the XML tree, so the I/O operation has to be done k times in the worst case. When k is big, fox example, 10000 or more(and this is always true in inverted list based methods), the process becomes very slow.

The above is the motivation of our XML content index structure based on Tries & Patricia[12] tree (we named it *TP*-Index). First we introduce a simple example XML tree for describing *TP-Index* followed.

For the XML data tree in Fig. 5, let C={"efficient", "xml", "index", "interval", "tree", "survey", "stream", "data"} be the vocabulary of words appeared (notice: we omitted "an"). And we have the query *Q* of "//paper/keyword[text()="*Interval Tree*"].

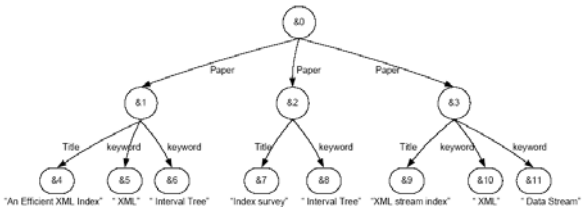


Fig. 5 XML data tree used for *TP-Index*

Fig. 6 is the *TP-Index* for C. For simplicity, we only show nodes related to words "interval" and "tree".

In Fig. 6, the backbone is a Tries & Patricia tree (directed edges). Each edge in it is labeled with a character, and there is an attribute *pre* of each node (inner and leaf nodes). Taking words "interval" and "index" for example. Because they have the same prefix "in" (length of "in" is 2), for the target node N of the branch labeled "i", the value of its *pre* attribute is 2. From N, the next character after prefix "in" in words "index" and "interval" are "d" and "t", respectively, so the outgoing edges are labeled accordingly. For each leaf node, its *pre* attribute is a word string it corresponding to, and there is an inverted list organized as B+ tree attached to it.

Now, we search "interval tree" in *TP-Index* to describe the detail of *TP-Index*. Let w_1="interval" and w_2="tree", searching w_1 in *TP-Index* can be as follows: from the root node, travel along the edge of label w_1 [0]= "i" (if it exists), if the target node is an inner node (its attribute *pre* is 2) , then continue the traveling along the edge labeled w_1[N.*pre*]=w_1[2]= "t", there we arrive at leaf node M. Be carefully, we have to do the final comparison of the pre content of M with w_1, because the traveling process can not definitely prove that the corresponding word of M is w_1 (may be word "i*nteger*" or like). The searching process of w_2 is the same.

To node M, there attached an inverted list organized as B+ tree. In the inverted list, each item has the structure of (parentnode, startpos). Take node &6 in Fig. 5 for example, its value is "Interval Tree", and its parent is node &1. In the string, the start position of word "Interval" is 0 and the word "Tree" starts at 9, so there is an item of (&1, 0) in the inverted list. As for (&1, 9) of the word "Tree", it is in the inverted list attached to node G.

Put the BPQU-Index and TP-Index together, we have the overall index mechanism *BTP*-Index.

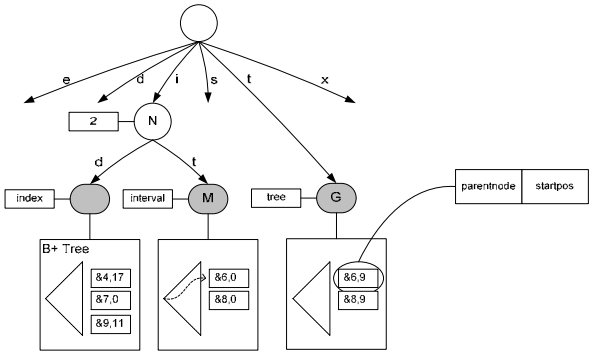


Fig. 6 *TP-Index* for C

Return to query example *Q* of //paper/keyword [text()="*Interval Tree*"]. The evaluation of it can be fulfilled as follows:

Sample query evaluation process.
Query=//paper/keyword[text()="Interval Tree"]
1 Processing the structure part ("//paper/keyword") of the query based on *BPQU-Index*, the result is R_S={&5, &6, &8, &10, &11};
2 Processing the content part (keyword[text()="*Interval Tree*"]) of the query;
2.1 Searching w_1 in *TP-Index* and we arriving at node M; For each $e_i \in R_s$, searching in the inverted list with the condition $\exists (p,s) \in InvertedList$, $p=e_i$, and get the corresponding (parentnode, startpos) item;
2.2 With the same procedure as 2.1, we get the result of searching w_2;
2.3 Combining the above two tables together, we get the result shown in Table 1.
3 For each row in Table 1, judge if the item (p_i, s_i) of w_i field and (p_{i+1},s_{i+1}) of w_{i+1} has the equality of s_i+len(w_i)=s_i+1, i=1,2,…,j. Here in the example, j=1;
4 Finally, mark all rows in column result with "Y" which match the condition in step 3.
5 Output all nodes whose corresponding result column is "Y".

Table 1. Result of step 2.3

e	w_1,len=9(including space)	w_2	result
&5	N/A	N/A	N
&6	(&6,0)	(&6,9)	Y
&8	(&8,0)	(&8,9)	Y
&10	N/A	N/A	N
&11	N/A	N/A	N

Please note that the *len* value of column w_i is $|w_i|$+1, because we add a separator between words to the preceding one.

The above procedure can be applied to string of many words, the process is the same and we will not discuss this any further.

As we have said, the inverted list is organized as a B+ tree. so for each word, the cost of *parentnode* searching is $\log_B|L|$, where L is the average length of inverted lists. For all words and all elements in R_s, the total cost is $k*j*\log_B|L|$, in which k is the cardinality of R_s, j is the word count of string and B is the fan out factor of B+ tree.

Put the structure evaluation and content evaluation together, the total cost is $O(t+|str|+c*\log_B|L|)$ (note that we replace $k*j$ with c), in which $|string|$ is the worst cost of searching string in backbone (Tries & Patricia tree) of *TP-Index*, t is the overall time expense of Q_s evaluation.

V. Experiments and Analysis

Since the widely used of structure index, we only use the index structure proposed by ref. [2] for comparison. The data set is DBLP[14]. The hardware and software environment are CPU-2GMHZ， RAM-512MBytes， Windows XP Professional.

Fig.7 shows the experiment result of the query processing. The time expense shown in *y*-axis is unitary (relative cost). The number of *x*-axis is the length of the query, i.e. the value of m in *Q*.

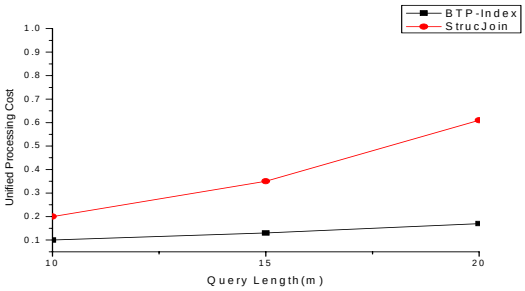


Fig.7 Experiment result of query *Q* processing

Following we analysis of the experimental result.

For the index of using structure Join[2] , all the index information is stored in a table whose items are of format <ParentID ； ElementTag ； ChildID>. Each item corresponds to an edge in the OEM diagram. When we use this structure to process the structure part of query of form *Q*, the index table has to be done the procedure of "Self-Join" like operation *m*-1 times. So the time expense increases rapidly with the growing length of the query path. As for the index structure based on path[4], the matching procedure must be done for all the paths of the index tree. The worst case is traveling the whole index tree. So the time expense is much larger, too. According to Theorem 1, The time complexity of query evaluation based on *BPQU-Index* is O(*h*). So with the growing length of query path, the time expense doesn't increase much. (Even with several join of *BPQU* result sets)

In all, The time expense of *BTP-Index* is much lower than other index structure when processing query *Q*.

VI. Conclusion and Future Works

We propose an XML index structure *BTP-Index* for efficiently process query *Q*, $Q=[//|/]e_1[//|/]e_2[//|/]\ldots[//|/]e_m[text()=str]$ which is used frequently in IR's XML data query and retrieval. Using *BPQU-Index* of *BTP-Index*, the evaluation of structure part of query Q will be fulfiled at time cost of O(*t*), combining the evalution of content part $element_m$[text()=*str*] by *TP-Index* structure (*TP* part of *BTP-Index*) of XML content, the whole worst time cost is $O(t+|str|+c*\log_B(|L|))$.

We will focus our future work on integrating other effective XML keyword search algorithm with BTP-Index to support interactive query on our dual-engine database management system and query optimization.

Acknowledgment

The authors wish to thank Ling Wu. This work was supported in part by a grant from Tianjin Natural Science Fund Project No. 07JCYBJC14400, China; China Purpose Oriented High Technology Project 863 Plan Project No. 2009AA01Z150.

References

[1] Q. Li, B. Moon. Indexing and querying XML data for regular path expressions. In: Proc of the 27th Int'l Conf on Very Large Databases (VLDB'01). Rome: Morgan Kaufmann, 2001, 361~370.

[2] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On Supporting Contianment Queries in Relational Mangement Systems.

[3] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, Y. Wu. Structural joins: A primitive for effcient XML query pattern matching. In: Proc of the 18th Int'l Conf on Data Engineering (ICDE' 02), 2002, 141~152.

[4] Roy Goldman, Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: Proc of the 23th Int'l Conf on Very Large Databases (VLDB'97),1997，436~445

[5] BF Cooper, N. Samle, MJ Franklin, GR Hjaltson, M.Shadmon. A fast index for semistructured data. In: Proc of the 27th Very Large Databases(VLDB' 01), 2001, 341~350

[6] C. Chung, J. Min, K. Shim. APEX: An adaptive path index for XML data. In: Proc of the 2002 ACM SIGMOD Int'l Conf on Management of Data, 2002, 121~132.

[7] Haixun Wang, Sanghyun Park, Wei Fan, Philip S Yu. ViST: A dynamic index method for querying XML data by tree structures. In: Proc of the 2003 ACM SIGMOD Int'l Conf on Management of Data, 2003, 110~121

[8] Xifeng Yan, Philip S. Yu, Jiawei Han. Graph Indexing: A Frequent Structure-based Approach. In: Proc of the 2004 ACM SIGMOD Int'l Conf on Management of Data, 2004.

[9] J. Clark and S. DeRose. XML path language (XPath). In: W3C Recommendation,1999, http://www.w3.org/TR/xpath.

[10] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J.Robie, and J. Sim¥eon. XQuery 1.0: An XML query language. In: W3C Working Draft, 2002, http://www.w3.org/TR/xquery/.

[11] Esko Ukkonen. On-line construction of suffix trees. Algorithmica, 1995, 14(3): 249~260.

[12] D.R. Morrison. Patricia-Practical Algorithm to Retrieve Information Coded in Alphanumeric. In: Journal of the ACM, Vol15, No.4, 1968, 514~534

[13] Y. Papakonstantinou, H. Garcia-Molina, J. Widom. Object Exchange Across Heterogeneous Information Sources. In: Proc of the 11th Int'l Conf on Data Engineering (ICDE' 95). Taipei: IEEE Computer Society, 1995, 251~260.

[14] DBLP, XML Data. At http://dblp.uni-trier.de/xml/.

[15] X. Dong and A. Halevy. Indexing dataspaces, In SIGMOD 2007

[16] Daniela Florescu, Donald Kossmann, Ioana Manolescu: Integrating keyword search into XML query processing. Computer Networks (CN) 33(1-6):119-135 (2000)
[17] Lin Guo, Feng Shao, Chavdar Botev, Jayavel Shanmugasundaram: XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003:16-27
[18] Yu Xu, Yannis Papakonstantinou: Efficient Keyword Search for Smallest LCAs in XML Databases. SIGMOD 2005:537-538
[19] Andrey Balmin, Vagelis Hristidis, Nick Koudas, Yannis Papakonstantinou, Divesh Srivastava, Tianqiu Wang: A System for Keyword Proximity Search on XML Databases. VLDB 2003:1069-1072
[20] Vagelis Hristidis, Yannis Papakonstantinou, Andrey Balmin: Keyword Proximity Search on XML Graphs. ICDE 2003:367-378
[21] Mayssam Sayyadian, Hieu LeKhac, AnHai Doan, Luis Gravano: Efficient Keyword Search Across Heterogeneous Relational Databases. ICDE 2007:346-355
[22] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In VLDB, 2002
[23] Kamal Taha, Ramez Elmasri: KSRQuerying: XML Keyword with Recursive Querying. XSym 2009:33-52
[24] Jiang Li, Junhu Wang, Mao Lin Huang: : effective and efficient keyword search in XML databases. IDEAS 2009:121-130
[25] Ziyang Liu, Yi Chen: Answering Keyword Queries on XML Using Materialized Views. ICDE 2008:1501-1503

Yanzhong Jin, 7/4/1972, Lanzhou, China. Master, physics, North West Normal Univ., Lanzhou, China, 1995; Post-graduate, computer science, Lanzhou Univ., China, 2000.

She works in Tianjin Sci.&Tech. Univ., Tianjin, China, Teacher. Her research interests includes XML data management, database system, and software engineering.

Publication:

[1]ArithBi+ --An XML Index Structure on Reverse Arithmetic Compressed XML Data, J.Computer Science, 2005(11)

She is a member of CCF(China Computer Federation).

Xiaoyuan Bao, 5/10/1971, Gansu, China. Master, physics, North West Normal Univ., Lanzhou, China, 1993; Post-graduate, computer science, Lanzhou Univ., Lanzhou, China, 1998; Doctor, computer science, Peking Univ. Beijing, China.

He work in Peking Univ., Beijing, China, Post-doctoral researcher. His research interests includes XML data management, database system, and information retrival.

Publications:

[1]Bao Xiaoyuan, Tang Shiwei, Yang Dongqing. Interval^{+}—An Index Structure on Compressed XML Data Based on Interval Tree[J].Journal of Computer Research and Development, 2006(07).

[2]Bao Xiaoyuan, Tang Shiwei, Wu Ling, Yang Dongqing, Song Zaisheng, Wang Tengjiao. ArithRegion—An Index Structure on Compressed XML Data[J].Acta Scicentiarum Naturalium Universitatis Pekinesis, 2006(01).

[3]Bao Xiaoyuan, Tang Shiwei, Yang Dongqing, Song Zaisheng, Wang Tengjiao. BloomRouter: A Framework for Dissemination of Compressed XML Stream[J]. Acta Scicentiarum Naturalium Universitatis Wuhan, 2006(01).

He is a senior member of CCF.