International Journal of Database Theory and Application
Vol.8, No.4 (2015), pp.49-58
http://dx.doi.org/10.14257/ijdta.2015.8.4.06

A Bottom-up Algorithm for XML Twig Queries

Tang Zhi-xian[1], Feng Jun[1], Xu Li-ming[1] and Shi Ya-qing[2,1]

[1]*College of Computer and Information, Hohai University*
[2]*Institute of Command Information Systems, PLA University of Science and Technology*
{hohaitangzx, fengjunhhu,xuliming.930}@gmail.com, yqshi_nanjing@163.com

Abstract

Twig query is a core operation in processing and optimizing XML structural queries. Recently, various algorithms have been proposed for finding twig patterns efficiently. Most of them are based on region labeling, pay little attention on nodes level information, and require additional caches such as stacks or lists to maintain the intermediate matching results, which cause the performance bottleneck of these algorithms. In contrast to previous work, we present a bottom-up algorithm for twig queries, which does not require additional caches and introduces idea of string searching to determine binary relationship between two nodes. Be- sides, this paper presents a node filtering mechanism–PathLevel, which can also be used in other algorithms for speeding up the query. Comprehensive experiments on several datasets demonstrate our method is an effective way for twig query.

Keywords: *Twig Pattern; Bottom-up; Node's Level; Dewey Encoding.*

1. Introduction

Twig query (Twig Pattern Matching) plays an important role in processing XML data and is a key factor that determines the efficiency of structural XML query processing. In literature, there is a great deal of work being done on how to support twig query. N. Bruno et al. [1] proposed a holistic algorithm, *TwigStack*, to the best of our knowledge, it is the first algorithm that addresses twig query without decomposing twig pattern into binary (parent-child and ancestor-descendant)relationships between pairs of nodes. Lu et al. in [2] proposed *TwigStackList*, which uses lists to store *PC* (parent-child) relationship. Lu et al. in [3] proposed a new algorithm–*TJFast*, which is based on new *entended Dewey*. Chen et al. proposed *Twig^{2}Stack* [4], introduced hierarchical stacks to enumerate the matching paths and *Twig^{2}Stack* performs better than *TwigStackList* and *TJFast*. In summary, all above algorithms have following common points:

1) Require additional caches (e.g., stack, list) to store the intermediate results;
2) Except TJFast, all of them use region-based representation of positions (e.g., Zhang[5] labeling) of XML nodes or string values;
3) Pay little attention in analyzing level information, implicated in twig pattern.

In this paper, we improve the existing Dewey[6] labeling scheme, called *Dewey**, which can easily support the determination of node's or its ancestor's label. Based on *Dewey**, we present a new algorithm, called *TwigLevel*, which is independently developed and queries twig pattern from bottom to up. The unique feature is that, *TwigLevel* converts the calculation of relationship between two nodes into searching a specific string, this change gives us a chance to develop a new efficient algorithm to match twig pattern based on current mature searching techniques.

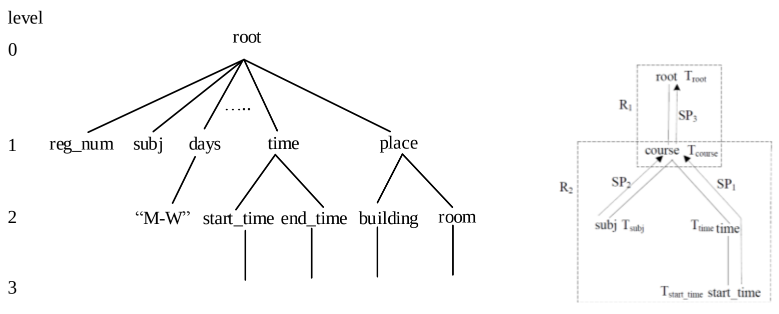
The main contributions of this paper are summarized as below. We propose a new bottom-up twig query algorithm, *TwigLevel*, which does not require to decompose twig pattern. Based on *Dewey** labeling method, we can determine the relationship between two nodes by searching specific string, and therefore we can fully use mature string searching techniques. We conduct extensive experimental studies on different datasets. Our experimental evaluations show that *TwigLevel* is an effective algorithm and is linear to the size of leaf node's streams.

The rest of this paper is organized as follows. Section 2 presents the notations used in this paper, the description of node's level and *Dewey** labeling. Section 3 introduces *TwigLevel* algorithm. We report our experimental studies in Section 4. In Section 5 we conclude our study and present our future work.

2. Preliminaries

2.1 Node's Level

An XML document can be modeled as a rooted, ordered, and labeled tree, each node corresponding to an element or a value and the edges representing *PC* or *AD* relationships. For each node or value, there is a depth/level that denotes what level this node is located. Figure 1(a) shows the tree representation of a sample XML document. As shown in Figure 1, the numbers in left side present the level that some nodes or values located. In this paper, the level starts from 0, which is the level of the root node, and other nodes' level is their parents' level plusing 1.



(a) tree representation (b) a twig pattern

Figure 1. A Sample XML

TwigLevel is a bottom-up algorithm that based on node's level information. The typical feature of *TwigLevel* is to convert the determination of relationship between two elements into searching the specific label (positional representation of an element) in a stream. We derive the level of upper level node from the label of the bottom node in a twig pattern, and then find whether it exists in upper-level node's stream, if success, the two nodes meet the binary structure relationships. To meet above requirements, we propose a new labeling scheme *Dewey**, which is based on the traditional *Dewey* labeling scheme. *Dewey** encodes node through a coding array called *Code* (See Figure 2), given an element *v* in XML document (we don't need to label the root node, that is to say, the label of the root node is an empty string), its *label(v) = label(parent_label(v)) + Code[i mod 52]*; if *v* is a text node, then *label(v) = label(parent_label(v)) + @*. As shown in Figure 3, a_1 is a root node, so label (a_1) is a empty string. b_1 is the 1st child of a_1 and therefore *label*(b_1) = 1/52 + *Code*[*i mod* 52] = 0*A*; Similarly, given *label*(f_1) = 0*D*0*B*, we know that f_1 is the 2nd child of its parent whose label is 0*D* (Figure 3).

```
Code ={
        "A","B","C","D","E","F","G","H",
        "I","J","K","L","M","N","O","P",
        "Q","R","S","T","U","V","W","X",
        "Y","Z","a","b","c","d","e","f",
        "g","h","i","j","k","l","m","n",
        "o","p","q","r","s","t","u","v",
        "w","x","y","z"
      };
```

Figure 2. Coding Array

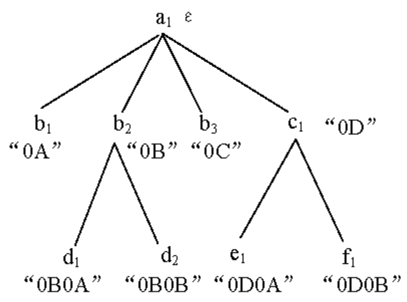


Figure 3. An Example for Dewey*Labeling

3. Bottom-up Algorithm: TwigLevel

3.1 Problem Statement

Given a twig query *Q* and an XML database *D*, a match of *Q* in *D* can be identified by a mapping from the nodes in *Q* to the nodes in *D*, such that query node predicates and structural relationships (*PC* or *AD*) between query nodes are satisfied by the corresponding database nodes. The results of a twig query with n nodes can be represented as a tuple $(d_1, d_2, \cdots, d_n)$ consists of the nodes in *D*.

3.2 Path Level Algorithm

To address the problem of matching twig pattern, we first split twig pattern into several path patterns and then merge their answers to form the results of twig query. In this section, we introduce algorithm *PathLevel* which calculates results for a path pattern and is a key part of algorithm *TwigLevel*.

The key idea of *PathLevel* is to use the label of the bottom node in twig pattern to infer the labels of its parents or ancestors, and to determine structural relationships by querying whether the specified label of parent or ancestor exists in the stream of parent of ancestor. Since the bottom node's label keeps the complete structural information from root to itself, algorithm PathLevel adopts bottom-up matching order to address path pattern. As shown in Algorithm 1, *PathLevel* can be divided two stages: Preparation stage, which finishes the work of parsing path pattern and node filtering (Algorithm 2); and Searching stage which executes searching on streams that have been processed in Preparation stage.

Algorithm 1 *PathLevel*(*P*)

```
Input: a path query $P$
Output: the matching results
Preparation Stage
1: $twigPatternParse(P)$
2: for all $n$ in $Nodes$ do
3:     $filterStream(T_n, n)$
4: end for
Searching Stage
1: $n \leftarrow getLeafNode(P)$
2: while $\neg eof(T_n)$ do
3: $L \leftarrow current(T_n)$
4: for $i \leftarrow (Nodes.legth-2)$ downto 1 do
5:         $m \leftarrow Nodes[i]$
6:     if the edge from $n$ to $Nodes[i]$ include only $PC$ edge then
7:         if $\neg isind(prefix(L,((depth(n)-(Nodes.legth-i-1))),T_i))$ then
```

```
8:              $delete(Tn)$
9:          end if
10:     end if
11:     if $Depths.legth = 1$ then
12:         if $\neg isFind(prefix(L, depth(m)))$ then
13:             $delete(Tn)$
14:         end if
15:     end if
16:     if $Depths(m).legth > 1$ then
17:         if all $d \in Depths_m$ that $isFind(prefix(L, d), Ti)$ return false then
18:             $delete(Tn)$
19:         end if
20:     end if
21: end for
22: end while
```

Now we turn to analyze Searching stage. As shown in Algorithm 1, line 1 gets the lead node of path pattern *P* through function *getLeafNode*(), lines 4-20 present how an element in Tn matches the path pattern. If the current element in Tn matches the given path pattern *P*, then processes the next element; otherwise, deletes this element. In the process of matching, we need to compute the level that node *m* locates. There are three situations for determining the level of node *m*: (1) if there is only a *PC* edge between node *(*n) and *m*, then *depth*(*m*)=*depth*(*n*)-*c* is the number of *PC* edge between *n* and *m* (lines 6-10); (2) if all elements in Tm locate at a same level, then we set *depth*(*m*) to this level (lines 11-15); (3) if elements in Tm locate at several levels, then *depth*(*m*) possibly equals to any levels stored in *Depths*(*m*) (lines 16-19).

After knowing the level of node *m*, *PathLevel* starts to judge the structural relationships between two elements in Tn and Tm respectively (lines 8,13,18). This step is particularly important, it is the key feature that distinguishes *PathLevel* from other twig query algorithms. Function *prefix*(*L*,*depth*(*m*)) returns the label of *L*'s ancestor that locates at *depth*(*m*) level. *isFind*(*prefix*(*L*, *depth*(*m*)),*Ti*) returns whether *prefix*(*L*, *depth*(*m*)) exists in stream Ti, if returns true, then continue to match the next element; otherwise, *L* certainly will not construct the final results. Because of function *isFind*(), *PathLevel* converts the determination of relationships between nodes into string query problem. Therefore, we can depending on nowadays mature string searching techniques to improve the efficiency of twig query (in this paper, we adopt common hashing method to implement string searching).

Having known that, through parsing of twig pattern we can get the possible or exact levels that the nodes in twig pattern locate. That is to say, those nodes that do not locate at such levels must not match the twig pattern. Thus, we can directly omit these nodes or only select the nodes that meet the requirements of level in the stage of twig pattern parse. However, only depending on the twig pattern, *Depths* may not keep the exact levels for a node (in fact, it is possible only a range that a node locates), and thus *Depths* requires further modifications, whose modification strategy is to modify *Depths* according to the node's stream.

Algorithm 2 *FilterStream*(Tn, n)

```
Input:  a node in twig query $n$, and its corrensponding stream $Tn$
Output:  a filtered stream $Tn$, and $Depthsn$ after processing
1: while $\neg eof(T_n)$ do
2:     $v \leftarrow current(T_n)$
3:   if $v \in Depths_n$ then
4:          delete $v$ from $T_n$
5:     else
```

```
6:          $Put(v, n)$
7:      end if
8:  end while
9:  procedure Put($d$, $n$)
10:   if $d \in Depths_n$ then
11:     return
12:   else
13:     Add $d$ into $Depths_n$
14:   end if
15: end procedure
```

For example, given a node *n* in twig pattern *q*, after parsing *q*, all levels are stored in $Depth_n$, then for each element *e* in T_n, if $e \in Depth_n$, add *label(e)* into T_n and add *depth(e)* into $Depth_n$; otherwise, omit *e* (line 6). Based on above ideas, we develop a level-based node Filtering mechanism, FilterStream, which is shown in Alogrithm 2. After filtering, elements in T_n all meet the requirements of level.

3.3 Twig Level Algorithm

Based on *PathLevel*, we develop a holistic twig query algorithm, *TwigLevel* (See Algorithm 3). For a twig pattern, it can be splitted into several path patterns that can be solved by using *PathLevel*. For better analysis, we introduce the concept of Query Region. Given a twig query (*Q*), then the root node of q represents a region; Besides, all the pathes from branch node to leaf node in Q compose the other regions. For example, Q1=/root/course[subj]/time/starttime, there are two regions (as illustrated in Figure 1(b)): R1 and R2.

```
Algorithm 3 TwigLevel(Q)
Input: a twig query $Q$
Output: the matching results
1: Preparation Stage
2: Searching Stage
3: for $i \leftarrow BranchingNodes.legth - 1$ down to 1 do
4:     $b \leftarrow BranchingNodes[i]$
5:     for all $P$ do
6:         $n \leftarrow leafNode(P)$
7:         delete all the elements $u(u \in T_n)$
8:         and elements $v(u \in T_b)$ that do not match the path query $P$
9:     end for
10: end for
```

TwigLevel processes twig query from bottom to up, that is, *TwigLevel* first processes the bottom region, which is most closed to the bottom node, and then processes the region above it. Take *Q*1 as an example, *TwigLevel* will first processes region *R*2 and then processes region *R*1. The main difference between *PathLevel* and *TwigLevel* is that *TwigLevel* not only needs to alter the streams of leaf nodes but also requires to alter the streams of branch nodes in each region dynamically. Just as *PathLevel*, *TwigLevel* needs to delete the elements that surely will not contribute to the final answers both in leaf nodes and branch nodes in each region. *TwigLevel* also includes two stages: Preparation stage and Searching stage. Lines 3-9 show that Algorithm *TwigLevel* begins to process each region from bottom to up, and then matches each path pattern included in each region.

4. Experimental Evaluation

4.1 Experimental Setting

We implemented all algorithms in MyEclipse 6.5 with JDK 1.6 using the Bekelery DB as the storage engine. All experiments were run on a computer with 3.1GHZ Pentium(R) 4 processor, 1GB main memory, and 150GB disk. The Operating System is XP. We used following datasets for experimental evaluations.

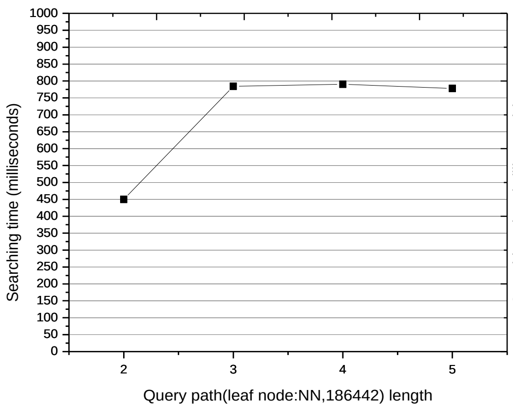
1) Reed is a dataset with 10546 elements which describles the courses information of Reed College, the maximum depth is 4 and the average depth is 3.19979.
2) SigmodRecord depicts the papers corresponding to XML published in SIGMOD, and there are 11526 elements in SigmodRecord. The maximum depth is 6 and the average depth is 5.014107.
3) Treebank is a deep dataset and has many self-recursive elements with the same label. The maximum depth is 36 and the average depth is 7.87279. It is about 82MB.

4.2 Path Query

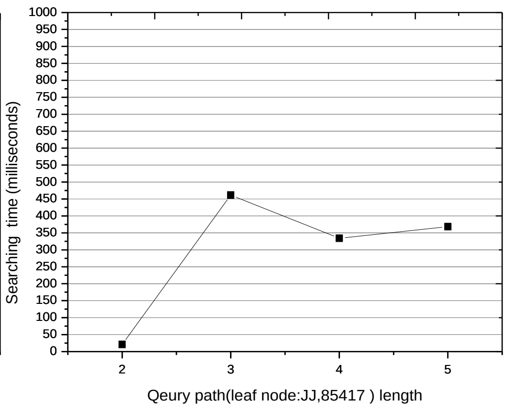
As shown in Table 1, there are four groups of path queries, each group has different leaf nodes and each query in a group has the same leaf node but with different path length. Figure 4 shows that in each group, processing time of each query almost has no relation with path length. Furthermore, as shown in Figure 4, the group with relatively smaller size of leaf nodes' stream consumes much less time than others. For example, Figure 3-(d) spends time from 10ms to 50ms, Figure 4-(c) from 50ms to 200ms. Therefore, combined with former analysis, the path length will almost have no influence on *PathLevel*.

Table 1. Path Queries with Same Leaf Node but with Different Path Length

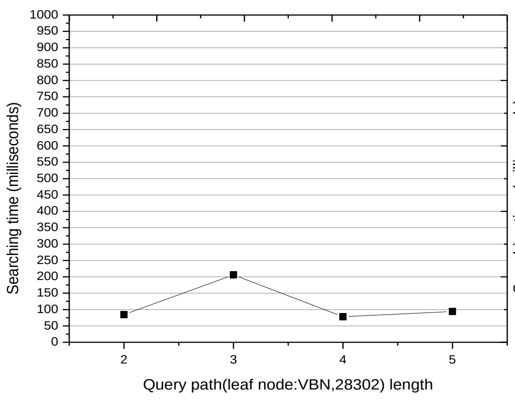
Path Query With Different Length and Leaf Node in TreeBank				
Path Length	**NN: 186442**	**JJ: 85417**	**VBN: 28302**	**NNPS: 3250**
2	//S//NN	//S//JJ	//S//VBN	//S//NNPS
3	//S/NP/NN	//S//VP//JJ	//S//VP//VBN	//S//VP//NNPS
4	//S/VP/NP/NN	//S/VP/NP/JJ	//S//VP/PP/VBN	//S//VP//NP//NNPS
5	//S/VP/PP/NP/NN	//S/VP/PP/NP/JJ	//S//VP/PP/NP/VBN	//S//VP//PP//NP/NNPS



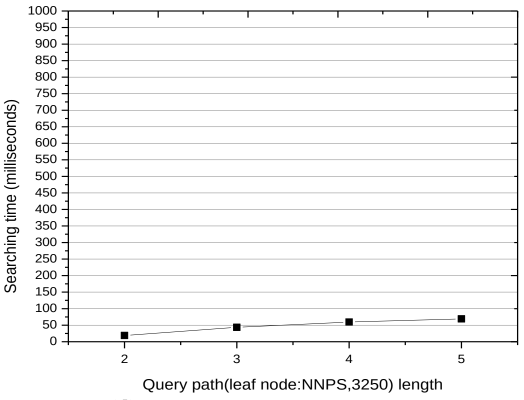
(a) NN:186442



(b) JJ:85417



(c) VBN:28302



(d) NNPS:3250

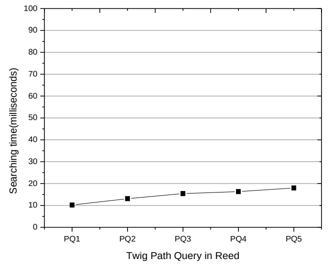
Figure 4. Comparison of Run Time over Different Queries with the Same Leaf Node (ms)

4.3 Twig Query

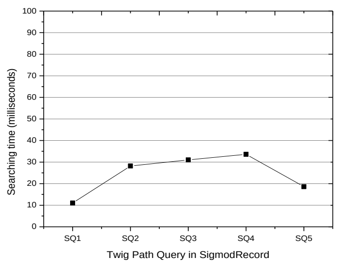
Figure 4 presents the processing time of twig queries given in Table.2. Because *TwigLevel* is built based on *PathLevel,* the processing time is also linear to the number of the elements in the streams of leaf nodes. As shown in Figure 5, in each group of query, the processing time has no direct connection with the length of twig pattern but has obvious relationship with the leaf nodes. Illustrated by the case of Figure 5-(a) and Figure 5-(b), the processing times of five twig queries have no obvious fluctuations, but for Figure 5-(c), fluctuation is apparent. Compared with Reed and SigmodRecord datasets, Tree-Bank is an irregular dataset that the size of stream of each node has big difference, which causes the above phenomenon. For example, TQ4 and TQ5, which have the same leaf nodes, need almost the same processing time, but compared with other queries, the times they consumed are apparently different.

Table 2. Twig Queries on Reed, Sigmod Record and Treebank

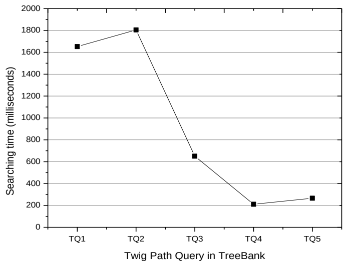
Twig Queries in Reed, SigmodRecord, and Treebank XML datasets		
PQ1 /root[.//reg_num]//title	**SQ1** //issue[number]//title	**TQ1** //S[NP]//JJ
PQ2 //course[title]/subj	**SQ2** //articles/article[title]/initPage	**TQ2** //VP[DT]//NN
PQ3 /root/course[subj]/time	**SQ3** //aricles[author]/article/endPage	**TQ3** //S/NP[NNS]/PP/IN
PQ4 //course[title]/place/building	**SQ4** //issue/articles[.//title]//author	**TQ4** //S[.//VP/IN]//NP/VBN
PQ5 /root/course[subj]/time/end_time	**SQ5** //SigmodRecord/issue/articles/article[initPage]/author/author	**TQ5** //S/VP/PP[IN]/NP/VBN



(a)Reed



(b)SigmodRecord



(c) TreeBank

Figure 5. Comparsion of Run Time over Different Twig Queries (ms)

5. Conclusion and Future Work

In this paper, we propose a bottom-up twig pattern matching algorithm that effectively queries twig pattern. *PathLevel* introduces string searching techniques into path query, which converts the determination of the binary relationship between two nodes into finding whether thecertain string exists. Based on *PathLevel*, we develop *TwigLevel*, which is used to match twig pattern. Experimental study shows that *TwigLevel* is an effective algorithm whose time complexity has no connection with twig pattern and is linear to the size of leaf nodes' streams. In our future work, we are planning to further study the performance of *TwigLevel* and *PathLevel*, and to extend this algorithm in distributed environment.

Acknowledgements

This work is supported by the National Natural Science Foundation of China under Grant No.61370091 and No.61170200, Jiangsu Province Science and Technology Support Program (industry) Project under Grant No.BE2012179, Program Sponsored for Scientific Innovation Research of College Graduate in Jiangsu Province under Grant No. CXZZ12_0229.

References

[1] N. Bruno, N. Koudas and D. Srivastava, "Holistic twig joins: optimal XML pattern matching[A]", In Proceedings of the 2002 ACM SIGMOD international conference on Management of data[C], **(2002)**; Madison, Wisconsin.

[2] J. Lu, T. Chen, T. W. Ling, "Efficient processing of XML twig patterns with parent child edges: a look-ahead approach[A]", In Proceedings of the thirteenth ACM international conference on Information and knowledge management[C], Washington, **(2004)**; D.C, USA.

[3] J. Lu, T. W. Ling and C-Y. Chan C-Y, "From region encoding to extended dewey: on efficient processing of XML twig pattern matching[A]", In Proceedings of the 31st international conference on Very large data bases[C], **(2005)**;Trondheim, Norway.

[4] S. Chen, H-G. Li and J. Tatemura, "Twig^{2}Stack: bottom-up processing of generalized-tree-pattern queries over XML documents[A]", In Proceedings of the 32nd international conference on Very large data bases[C], **(2006)**; Seoul, Korea.

[5] C. Zhang, J. Naughton and D. Dewitt, "On supporting containment queries in relational database management systems[J]", SIGMOD Rec, vol.30, no.2, **(2001)**, pp.425-436.

[6] I. Tatarinov, S. D. Viglas and K. Beyer, "Storing and querying ordered XML using a relational database system[A]", In Proceedings of the 2002 ACM SIGMOD international conference on Management of data[C], **(2002)**;Madison, Wisconsin.

Authors

Tang ZhiXian, he received the B.S. degree and M.S degree in Computer Science and Technology from Hohai University, China, in 2007 and 2010, respectively. Now, he is a Ph.D. candidate in College of Computer and information, Hohai University. His research interests include spatiotemporal indexing and searching methods. E-mail: hohaitangzx@gmail.com.

Feng Jun, he received the Ph.D. degree in Information Engineering from University of Nagoya, Japan, 2004. She is currently a professor in College of Computer and information, Hohai University, Nanjing, China. She has been worked as a visiting scholar twice in University of Nagoya, from March 2005 to January 2006 and from December 2011 to February 2012, respectively. Her research interests include data management, spatiotemporal indexing and search methods, ITS and domain data mining. E-mail: fengjunhhu@gmail.com.

Xu LiMing, he received the B.S. degree from Jinling Technology Institute, China, in 2009, and received M.S degree in Computer Science and Technology from Hohai University, China, in 2012. Now, he is a Ph.D. candidate in College of Computer and information, Hohai University. His research interests are mainly about XML indexing and searching, and data management. E-mail: Xuliming.930@gmail.com.

Shi YaQing, he received the M.S degree in Computer Application Technology from Hohai University, China, in 2007. Now, she is a Lecturer in PLA University of Science and Technology and a Ph.D. candidate in College of Computer and information, Hohai University. Her research interests include moving objects indexing and searching methods. E-mail: yqshi_nanjing@163.com.