International Journal of Database Theory and Application
Vol.8, No.4 (2015), pp.169-178
http://dx.doi.org/10.14257/ijdta.2015.8.4.17

Storing and Updating XML Data Tree based on Linked Lists

Teng Lv[1] and Ping Yan[2, *]

[1]*School of Electronic and Communication Engineering, Anhui Xinhua University, Hefei 230088, China*
[2]*School of Science, Anhui Agricultural University, Hefei 230036, China*
Lt0410@163.com, want2fly2002@163.com

Abstract

XML has become the de facto standard for data exchange and transformation on the World Wide Web and is widely used in many applications of various fields, so it is urgent to develop some efficient methods to manage, store, query, and update XML data. There are two main methods to do this: the first method is a native approach which uses native XML databases to store XML data, and the second method use other mature commercial databases approaches to store and manage XML data considering the advantages of mature technologies of the commercial databases, especially use relational databases to store, query, and update XML data. For relational databases approach, although it can take advantage of mature technologies of relational databases, it needs to map XML data to relational data. In this paper, we research the problem of how to store XML data so that storing and updating of original XML data can be efficient than relational approach. We proposed a method to store XML data into linked lists with inverted index, in which the relationships between nodes of XML data tree are preserved by the links in linked lists. Inverted index are created for linked lists for efficiently querying and updating XML data tree. Two kinds of updates are considered including inserting a new node in or deleted an existed old node from XML data tree. Theoretical analysis of our algorithms shows that the methods proposed in the paper are efficient.

Keywords: *XML, store XML, update XML, insert XML, delete XML*

1. Introduction

XML (Extensible Markup Language) has become the de facto standard for data exchange and transformation on the World Wide Web and is widely used in many applications of various fields. With more and more XML data used in different applications, it is a natural problem of how to efficiently manage, store, search, and query XML data. Academic researchers and industry engineers have used two main methods of managing, storing, searching, and querying XML data: the first way uses native XML databases to store and manage XML data directly, such as Refs.[13,14], and the second way[4,5,12,15] used other mature commercial databases to store and manage XML data considering the advantages of mature technologies of the commercial databases, such as query optimization, normalization theories, index technologies, etc. Although the mature databases approach has the above advantages, it has to map XML data to the model in respect databases in order to store it in the mature databases[9]. Furthermore, to search or query XML data, the original XML queries such as XPath or XQurey have to be transformed to SQL(Structure Query Languages) or OQL(Object Query Languages) queries, and the results of SQL or OQL queries have to be transformed to XML formats again, which increase of query cost and leads to low query efficiency. For native

* To whom correspondence should be addressed. E-mail: want2fly2002@163.com, Tel:+86-551-15155139852.

ISSN: 2005-4270 IJDTA

approach, there is no such problems because it stores and queries XML data directly but with low query and update efficiency.

In this paper, we research the problem of how to store XML data so that updating, storing, and querying XML data are efficient. We first store each non-leaf node of XML data file in a set of linked list, in which the relationships between parent and child nodes (and also ancestor and descendent nodes) are preserved. Then inverted indexed are designed for all the linked lists to gain high querying and updating efficiencies. After that, updating XML data file are researched and two algorithms are given for inserting new nodes and deleting existed nodes of XML data file, respectively.

Related Work. Several solutions for storing XML data have been proposed both in academic and commercial community. These storage approaches can be classified according to the type of system or model on which XML document representation rely.

The first type uses mature database technologies to store XML documents. The primary choice is to use relational databases[3,16], in which a collection of relational tables are used to represent both XML data and their relationships and XML documents should be partitioned into rows and columns of relational tables. However, this choice can cause performance overhead mainly due to translation of tree structure of XML to tables of relational databases and vice versa. Another choice is to use object-oriented databases[10] to store XML documents, in which XML data are stored as collections of object instances. This choice can take advantage of object-oriented merits, such as inheritance and capsulation, however, how to organize XML data into an object is not natural and in some cases it is not flexible and efficient. To combine advantages of relational and object-oriented databases, object-relational databases are used to store XML data[17], but with disadvantages of both relational choice and object-oriented choice as mentioned above.

The second type uses native XML databases to store XML documents[7] in which XML documents are stored and retrieved according to a native XML model which defines XML data as well as its structure and schema. But native XML approach usually has low query efficiency. To overcome above disadvantages, Refs.[1,2,6,11] store XML documents based on various partition method. They can be used to improve XML queries efficiency because only relevant XML data needed to be accessed when queries are simplified to specific paths. For XML node labeling methods, Ref.[8] proposed a labeling scheme based on the concept of the complete tree whose space requirement of labeling scheme is superior to others in most cases.

Our work is concentrated on storing XML data efficiently in order to support efficient querying, accessing, and updating XML data directly. In this paper, we use a simple path label for each node to represent its path in the XML data tree. To capture the overall structure of nodes in the XML data tree, a set of linked list for each node are constructed to achieve this goal. To support query, access, and update efficiently, inverted index are constructed for the set of linked list. The advantages of the method proposed in the paper can support update easily as the nodes are linked by their parent links and sibling links in the linked lists.

Main Contributions. In this paper, the problem of storing and updating XML data are researched whose main contributions are followings: (1) We give an algorithm of storing XML data tree in linked lists with inverted index. (2) Two algorithms are given to update linked lists with inverted index when insert a new node or delete an existed node in the XML data tree.

Organization. The rest of the paper is organized as follows. Section 2 gives the algorithm to store XML data tree into a set of linked lists with inverted index. Section 3 gives the algorithms to update linked lists with inverted index when insert

a new node or delete an existed node in the XML data tree. And finally, Section 4 concludes the paper and gives the future direction of the work.

2. Storing XML Data File in Linked Lists with Inverted Index

An XML data file is usually represented as a labeled XML tree. Considering the following XML data file:

```
<A>
    <B>
      <C> vc1</C>
      <C>vc2</C>
      <D>
        <F>vf1</F>
      </D>
      <D>vd2 </D>
      <E> ve1</E>
    </B>
    <B>
      <D>
        <F>vf2</F>
      </D>
    </B>
    <B>
      <D>vd4 </D>
      <E>ve2 </E>
    </B>
</A>
```

Figure 1. An XML Data File

We can depict the above XML data file as an XML data tree such as Figure 2. In Figure 2, an uppercase letter is used to denote an element type, and an uppercase letter followed by a number to indicate a specific element node of this element type. In Figure 2, A, B, C, D, E, and F are all element types, A1 is a specific element node of type A, which is the root node of the XML tree, B1, B2, and B3 are three specific element nodes of type B, which are non-leaf nodes of the XML tree. D1 and D3 are two specific element node of type D, which are non-leaf nodes of the XML tree. For leaf nodes of XML data tree, C1 and C2 are two specific element nodes of type C, D2 and D4 are two specific element nodes of type D, E1 and E2 are two specific element nodes of type E, and F1 and F2 are two specific element node of type F. For each leaf node, there is a value assign to it, e.g. “vc1” is the value of leaf node “C1”. For the path of each node, we labeled them as following: For the root node, it is always labeled as “/1”. For other nodes, it is labeled orderly as its occurrence order under its parent node prefixed by its parent’s label recursively. For example, node B1 is the first child node of root node A1, so it is labeled as “/1/1” because its parent node A1’s label is “/1” and it is the first child node of A1; and D3 is the first child node of B2, so it is labeled as “/1/2/1” as its parent node B2 is labeled as “/1/2” and it is the first child node of B2.

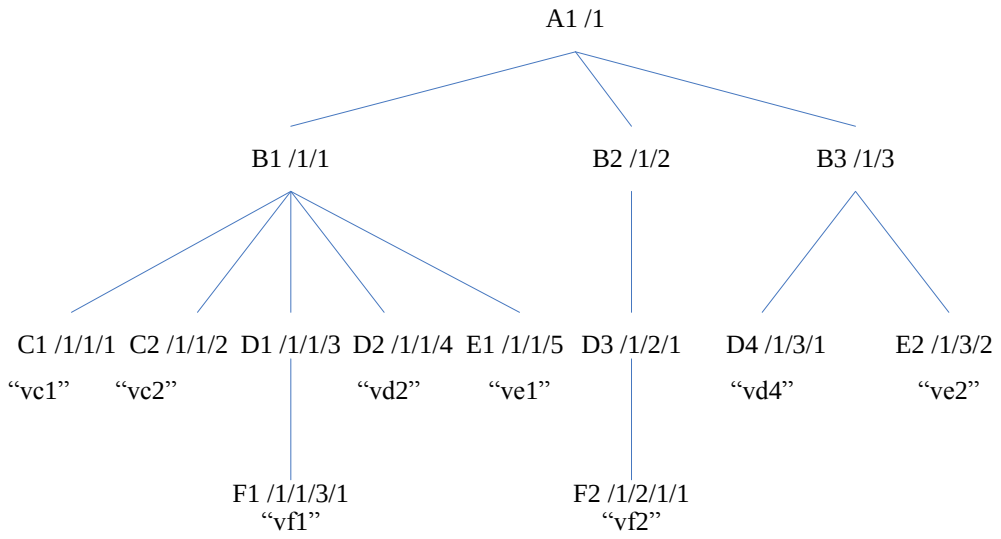

Figure 2. An XML Tree of Figure

For an XML tree such as Figure 2, we can store it as linked lists for efficiently updated, accessed and queried. For each non-leaf node, we construct a linked list, in which each node has the following form:

```
NODE
{
    NODE *Parent;
    String ElementType;
    String Path;
    NODE *Next;
    String Value;
};
```

where *Parent points to its parent node (except for the head node of XML tree), ElementType is the element type of the node, Path indicates the node's path which starts from the root node to the node itself, *Next points to the next child node of the head node of the linked list (except for the tail node, which points to "NULL" indicating that there is no more child node of the head node). Value of a node is used to store a value for each leaf node. But for clarity and without loss of generality, we do not consider the value of a node. Path of a node is a string separated by "/". For example, root node A1 of Figure 1 has the following linked list which says that A1 has 3 child nodes: B1, B2, and B3, respectively, each child node points to its parent node by a pre-link *Parent and linked together by post-link *Next. For leaf nodes, it is unnecessary to create a separated linked list as non-leaf nodes do, because leaf nodes are already directly linked by their parent nodes in the set of linked list and their structure relationships have been captured and stored in linked lists of their corresponding parent nodes.

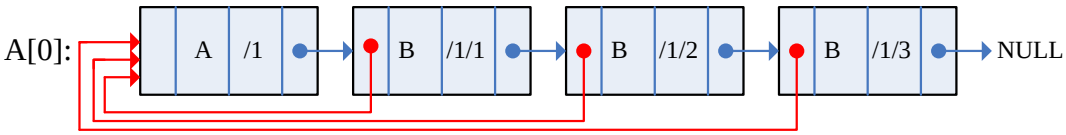

Figure 3. A Linked List of Figure 2

The overall linked lists are given in Figure 4 (values of leaf nodes are omitted here).

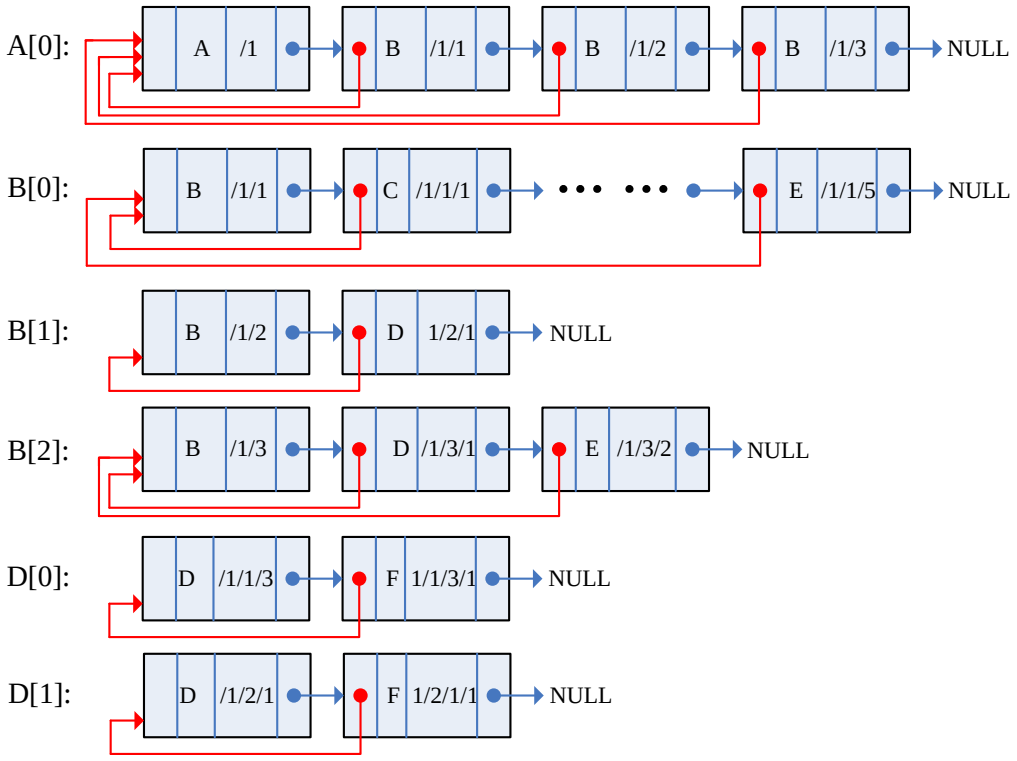

Figure 4. The Linked Lists for Figure 1

To improve update, access, and query efficiency, we can create inverted index on head nodes of the set of linked list based on what child nodes are contained in each head node. From Figure 5, we can easily accessed element type C just go through index "C:{B[0], B[1]}", i.e. the 2nd and 3rd linked list B[0] and B[1] of Figure 4 without traversing other linked lists.

$$"B" \Rightarrow \{A[0]\}$$
$$"C" \Rightarrow \{B[0]\}$$
$$"D" \Rightarrow \{B[0], B[1]\}$$
$$"E" \Rightarrow \{B[0], B[2]\}$$
$$"F" \Rightarrow \{D[0], D[1]\}$$

Figure 5. The Inverted Indexed Linked Lists for Figure 4

Now, we give the algorithm to create linked lists with inverted index to store XML data file as following:

Algorithm 1. Create linked lists with inverted index for an XML data tree T:

```
CreateLists(T)
   {
   For each node h of T While (NotLeaf(h)) //leaf nodes has already been included in their
parent nodes
   {
   h[k++]=CreateList(T, h);
   }
}
```

where CreateList( ) is a function to create a linked list for a non-leaf node h for a given XML data tree as follows:

```
CreateList(T, h)
{
New NODE Lh;  //First create a head node
Lh->Parent=NULL;
Lh->ElementType=h;
If (h is not visited) Then //it is a head node of the tree
Lh->Path=”/1”;
Else //it already has a path
  Lh->Path=Lh->Parent->Path+”/”+Order(h);  //Order returns h’s order as a child of its parent node
Lh->Next=NULL;
AddNodes(T, h, Lh);
}
```

where AddNodes( ) is a function to add child nodes for a linked list for a given head node Lh and XML data tree as follows:

```
AddNodes(T, h, Lh)
{
If (m=GetFirstChild(T, h)) Then  //Insert its first child node
{
New NODE Lm;  //Create a child NODE Lm
Lm->Parent=Lh;
Lm->ElementType=ElementType(m);
Lm->Path= Lm->Parent->Path+”/1”;
Lm->Next=NULL;
Lh->Next=Lm;
AddLists(Lh, Lm);
}
Else Break;
While(k=GetNextChild(T, h)) Do  //Insert other child nodes if existed
{
Lm=GetCurrentChild(T, Lh);
New NODE Lk;  //Create a child NODE Lk
Lk->Parent=Lh;
Lk->ElementType=ElementType(k);
Lk->Path= Lk->Parent->Path+”/”+Order(k);  //Order returns k’s order as a child of its parent node
Lk->Next=NULL};
Lm ->Next=Lk};
      AddLists(Lh, Lk);
```

where AddLists( ) is a function to add a linked list Lh into the inverted indexed linked list according to its child element. A list is created to store such information:

```
AddLists(Lh, Lm)
{
If Lh is not in Lm list Then  //If not existed then create it
New List Lm;
Else //If existed then just insert
Lm.Add(Lh);
}
```


Complexity Analysis. Suppose there are *n* non-leaf nodes and each has *c* child nodes on average in the XML tree. CreateLists( ), CreateList( ), AddNodes( ), and AddLists( ) all can be executed in constant time $O(1)$, and CreateLists( ) executes *n* times and AddNodes executes *c* times for each non-leaf nodes, so the overall complexity of Algorithm is $O(c+n)$. Suppose the nodes and edges are N and E for the given XML tree, as n and *c* is not greater than N and E, respectively, so the final complexity of Algorithm is $O(N+E)$.

3. Updating XML Data File in Linked Lists with Inverted Index

When an XML data file is updated, the linked lists with inverted index should be updated accordingly. The first kind of update is to insert a new node in XML data file. We should know the new inserted element type, its parent element, its order as child node of its parent node, etc. The following algorithm gives the details of how to update the corresponding linked lists Lh[k] when a new node is inserted in a given XML data tree.

Algorithm 2. Update inverted indexed linked list when insert a new node in XML data tree T:

```
InsertNode(T, p, e, o, Lh[k]) //Parent node p, new node e, order o of e as child nodes of
p
{
For i=1 to k  //Insert a new node e in  linked list Lh[k]
{
 If (Lh[k]->ElementType= p) Then  //Find the proper parent node
 {
   For i=1 to o-1  //Find the proper position to be inserted
  {
   If Not(m=Lh[k]->Child) Break;  //Exception handle
{
n=m->Next;  //insert the new NODE e
m->Next=e;
e->Parent=p;
e->Next=n;
e->ElementType=e;
e->Path=p->Path+(Postfix(Path(m))+Postfix(Path(n))/2;   //Reassign a pathBreak;
}
}
}
}
AddLists(Lh[k], e);  // Update inverted index of  linked lists
}
```

In Algorithm 2, we reassign a new path for the inserted new node e as following: its last symbol is the average of the last symbol of its previous and next sibling node, e.g. if m->Path=”/1/2/3” and n->Path=”/1/1/4”, then the last symbol of the path for node e is (3+4)/2=3.5, so e->Path=”/1/1/3.5”.

Complexity analysis. From Algorithm 2, we can see that insert a new node only require $O(ko)$, which is a constant time as k and o are two constant number, so the complexity of Algorithm 2 is $O(1)$.

The second kind of update is to delete an existed node in an XML data file. When the node is not a leaf node, the delete process can no be finished because it contains other nodes. To delete a non-leaf node and their descendants, we can delete its leaf nodes iteratively. The following algorithm gives the details of how to delete the corresponding node in the linked lists Lh[k] when an existed non-leaf node is deleted from a given XML

data tree. We should know the related information such as deleted element type and its parent element, etc.

Algorithm 3. Delete an existed node from inverted linked list:

```
DeleteNode(T, p, e, Lh[k])
   {
   If NotLeaf(e) Then Break;  //node e is not a leaf-node so cannot be deleted.
   For i=1 to k  //Delete a NODE e from  linked list Lh[k]
   {
    If (Lh[k]->ElementType= p) Then  //Find the proper parent NODE
    {
     While(m=Lh[k]->Next)  // Find the NODE e to be deleted
     {
      If (e==m) Then
   {
                  n=m->Next;  //Delete NODE e
   s=m->Parent;
   s->Next=n;
   DeleteLists(Lh[k], e); //Update inverted  linked list
   Break;
   }
       Else k++;
   }
   }
   }
}
```

where DeleteLists( ) is a function to update inverted linked list if necessary as follows:

```
DeleteLists(Lh[k], e)
{
   While(m=Lh[k]->Child)
   {
    If m= e Then
        Break;  //Same element types of e in the list, so do not change the lists
  }
    For i=k+1 to n
    {
     e[i-1]=ElementType[i];  //No same elements types in lists so delete corresponding
array of element e
    }
}
```

Complexity Analysis. From Algorithm 3, we can see that delete an existed node only require $O(k(n-k)o)$, which is a constant time as k, n, and o are all constant numbers(o is the order of the deleted element in the XML data tree), so the complexity of Algorithm 3 is $O(1)$.

4. Conclusions and Future Work

This paper proposed a new storing method to store XML data in linked lists with inverted index which can support update of original XML data efficiently. Two kinds of update are considered in the paper: the first update is to insert a new node in the XML data file, and the second one is to delete an existed node from the XML data file. From theoretical analysis, all of the two update jut need a constant time to achieve their

respective goals, which is very efficient and can be used in real applications such as there is a large number of nodes in an XML data file.

How to label nodes in linked lists is an interesting future work. This paper just uses a simple path to label a node in linked lists whose efficiency may be improved by methods such as path compression. The key challenge is how to label and compress paths without losing structure information between nodes of XML data file.

Acknowledgement

The work is supported by Introduction of Talents Foundation and Academic Leader Foundation of Anhui Xinhua Unversity (2014XXK06), National Natural Science Foundation of China (No. 11201002) and Natural Science Foundation of Anhui Province (No.1208085MF110).

References

[1] Sonic Software Corporation. http:// www.sonicsoftware.com/products/sonic_xml_server/ index.ssp.
[2] Software AG. http://www1.softwareag.com/ corporate/products/tamino/default.asp.
[3] L. J. Chen, P. A. Bernstein, P. Carlin, D. Filipovic, M. Rys, N. Shamgunov, J. F. Terwilliger, M. Todic, S. Tomasevic and D. Tomic, "Mapping XML to a wide sparse table", Proceeding of IEEE 28th International Conference on Data Engineering, **(2012)**; Arlington, Virginia, USA.
[4] A. Deutsch, M. Fernandez and D. Suciu, "Storing semi structured data with STORED", Proceedings of ACM SIGMOD International Conference on Management of Data, **(1999)**; Philadelphia, Pennsylvania, USA.
[5] A. Schmidt, M. L. Keysten, M. Windhouwer and F. Wass, "Efficient relational storage and retrieval of XML documents", Proceedings of the Third International Workshop on the Web and Databases, **(2000)**; Dallas, Texas, USA.
[6] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. J. Shekita and C. Zhang, "Storing and querying ordered XML using a relational database system", Proceedings of the ACM SIGMOD international conference on Management of Data, **(2002)**; Madison, Wisconsin, USA.
[7] T. Lv and P. Yan, Information and Software Technology, vol.4, no.48, **(2006)**.
[8] P. Bohannon, J. Freire, P. Roy and J. Simeon, "From XML schema to relations: A cost-based approach to XML storage", Proceedings of the 18th International Conference on Data Engineering, **(2002)**; San Jose, California, USA.
[9] F. Tian, D. J. DeWitt, J. Chen and C. Zhang, ACM SIGMOD Record, vol.1, no.31, **(2002)**.
[10] J. McHugh, S. Abiteboul, R. Goldman, D. Quass and J. Widom, ACM SIGMOD Record, vol.3, no.26, **(1997)**.
[11] N. D. Widjaya, D. Taniar and J. W. Rahayu, "Inheritance Transformation of XML schema to object-relational database", Proceedings of 4th International Conference on Intelligent Data Engineering and Automated Learning, **(2003)**; Hong Kong, China.
[12] C. Kanne and G. Moerkotte, "Efficient storage of XML data", Proceedings of the 16th International Conference on Data Engineering, **(2000)**; San Diego, California, USA.
[13] A. Arion, A. Bonifati, I. Manolescu and A. Pugliese, World Wide Web, vol.1, no.11, **(2008)**.
[14] K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, R. Lyle, F. Ö zcan, H. Pirahesh, N. Seemann, T. C. Truong, B. V. der Linden, B. Vickery and C. Zhang, "System RX: one part relational, one part XML", Proceedings of the ACM SIGMOD International Conference on Management of Data, **(2005)**; Baltimore, Maryland, USA.
[15] H. Georgiadis and V. Vassalos, "Xpath on steroids: exploiting relational engines for xpath performance", Proceedings of the ACM SIGMOD International Conference on Management of Data, **(2007)**; Beijing, China .
[16] R. Murthy, Z. Liu, M. Krishnaprasad, S. Chandrasekar, A. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora and V. Krishnamurthy, "Towards an enterprise XML architecture", Proceedings of the ACM SIGMOD International Conference on Management of Data, **(2005)**; Baltimore, Maryland, USA.
[17] R. Lin, Y. Chang and K. Chao, "A Compact and Efficient Labeling Scheme for XML Documents", Proceedings of 18th International Conference on Database Systems for Advanced Applications, **(2013)**; Wuhan China.

Authors

Teng Lv, born on April 1975, Datong, Shanxi Province, China
Current position, grades: an associate professor in Anhui Xinhua University, PhD.
University studies: BSc degree in Computer Science from Artillery Academy (1997), MSc degree in Computer Science from Artillery Academy (2000), Ph.D degree in Computer Science from Fudan University (2003)
Scientific interest: His research interest fields include Data management
Publications: more than 70 papers published in various journals and referenced conferences
Experience: He has teaching experience of 10 years, has completed 4 scientific research projects

Ping Yan, born on December 1972, Urumqi, Xinjiang Uygur Autonomous Region, China
Current position, grades: a professor in Anhui Agricultural University, PhD.
University studies: BSc degree in Applied Mathematics from Xinjiang University (1994), MSc degree in Applied Mathematics from Xinjiang University (1999), Ph.D degree in Applied Mathematics from Fudan University (2002)
Scientific interest: Her research interest fields include neural networks and data management
Publications: more than 50 papers published in various journals and referenced conferences
Experience: She has teaching experience of 20 years, has completed 6 scientific research projects