An Indexing Method for Supporting Structure Queries on XML Documents

Y.R. Jean
yrjean@pu.edu.tw

H.W. Hsiao
g8911116@pu.edu.tw

Department of Computer Science and Information Management
Providence University
200 Chung-chi Rd.,Shalu Taichung County, TAIWAN 433

Abstract

Much research has been carried out to manage structured documents such as SGML (Standard Generalized Markup Language) or XML (eXtensible Markup Language) documents and to provide powerful query facilities exploiting document content, structure, and attributes. In order to perform structure queries efficiently on structured documents, a good indexing method for structured documents is important and necessary. In this paper, we present an indexing method which is utilized for supporting efficient query on content, structure and attributes on XML documents. Furthermore, we propose some strategies of update method to maintain the integration of changed index information and original XML documents.

Keywords: structure query, index method, DOM parser, structured document, and XML

1 Introduction

Structured documents are the documents that embed the document structures into the texts. Recently, many documents tend to be produced as structured ones using markup languages like SGML or XML since they make it possible to handle the texts in piece by piece in browsing or retrieval. Assume that the documents have been originally supplied with tagged in SGML. Furthermore, World Wide Web is likely to step toward XML from HTML (HyperText Markup Language) soon in creating Web pages. SGML and XML provide full-fledged features in making documents structured as they are, whereas HTML has only limited functions in structuring.

This tendency calls for the emergency of a new information retrieval system that enables to retrieve and access arbitrary parts of documents easily. It raises a difficult problem that the system should be able to figure out relevant elements to users queries issued at any level of the structure, which have not been tackled seriously in the conventional information retrieval system. And most of the structuring techniques proposed so far did not handle the problem efficiently.

In this paper, we propose an indexing method for supporting query on content, structure and attributes on XML documents that minimizes the indexing overhead and guarantees fast query response time. The main idea is that indexing is performed by assign an index interval to every node while traversing the tree in preordering way. We build inverted lists for content, structure and attributes query on XML documents.

This paper is organized as followings. In Section 2 we describe some related works about indexing methods on structured documents, in Section 3 we discuss how to build the index information, modify the index interval method, and reduce the space overhead on index information to improve query efficiency and more query types. Section 4 is describing the update method to maintain the integration of index information and original XML documents after the structured documents had be inserted, deleted, or modified. In section 5, we discuss the works we are doing follows up this paper was published and some future works that can improve in the information retrieval system. Section 6 presents concluding remarks with the proposed concepts of this paper.

2 Related works

For years, there has been growing interest in handling structured documents well in terms of indexing and retrieval. In this section, we discuss some indexing methods for structured documents e.g. SGML or XML documents. First one is a kind of indexing methods on structured documents by using TEI (Text Encoding for Interchange) guidelines – an encoding standard adopted by the humanities desciplines. Second one is uses an m-ary complete tree structure to create a document tree and assign a UID for each node of tree structure according to the order of the level-order tree traversal in subsection 2.2. In subsection 2.3, a method uses the index interval concept to assign each node the step numbers for reaching and leaving the node while traversing by the preorder way.

2.1 Indexing with TEI guidelines

Tuong Dao [6] encodes the structured document according to the TEI guidelines, and uses the extension of SCL (Simple Concordance Lists) model to support content, structure, and attribute queries on structured document. But, this method does not use the concept of tree structure to handle the index information. It uses containment relationships rather than hierarchical relationships to support queries on document structure. The document in Figure 1 is the collection of English classic texts available for public use at the Oxford Text Archive which contains three divisions, the table of contents and two chapters. In Figure 1, structural elements are marked up with XML tags and encoded by TEI guidelines.

```
     100.1          100.2    101    102  103    104    104.1
<doc n="1975"> <dtitle> Tarzan  of  the  Apes </dtitle>
     104.2          105         105.1
<div type="toc"> Contents … </div>
          105.2                 106    107   108      109
<div type="chapter" id="C7"> The  Light  of  Knowledge …
109.1    110    111    112      113     114 115  116    117  118
<p> …  Let  all  respect  Tarzan  of  the  Apes  and  Kala,
    119    120     120.1    120.2
    his  mother … </p> … </div>
          120.3                 121    122   123     124
<div type="chapter" id="C11"> King  of  the  Apes …
124.1   125    126    127    128    129      130     131       132  133
<p>  And  thus  came  the  young  Lord Greystoke  into  the
     134       135   136   137    137.1
    kingship  of  the  Apes … </p>
   137.2      137.3
… </div> … </doc>
```

Figure 1:Document in the text collection

2.2 Indexing with UID

Lee et al. [7] proposed an indexing structure that is able to reduce the storage overhead taken to indexing at all levels of document structure. They first represented a document as an m-ary complete tree where m is the largest number of child elements of an element in the structure. The result of the mapping is called 'document tree'. Secondly they assigned each element a UID (Unique element IDentifier) according to the order of the level-order tree traversal. For example, the document tree in

Figure 2, we assign UID's as shown in the Table 1 assuming a 3-ary tree. Here, we can consider this document tree as following: the node *a* is a *book*, *b* a *chapter*, *d* a *section*, and *h* is a s*ubsection*. Because they use the concept of complete tree, there are some virtual nodes which do not exist.

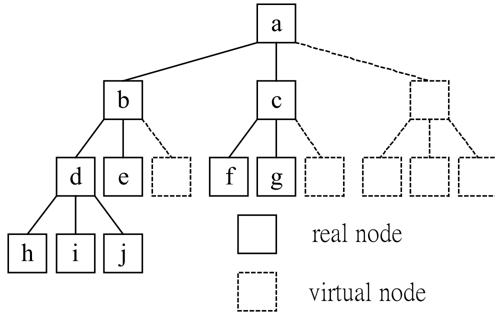


Figure 2:Example of document tree

Table 1:Unique element identifiers

Element	UID	Element	UID
a	1	f	8
b	2	g	9
c	3	h	14
d	5	i	15
e	6	j	16

The UID's of the parent and *j*-th child of a node whose UID is *i* can be obtained by the following two functions. Using these functions we can only decide the parent-child relationship between two nodes on condition that the difference of their level numbers is just 1. If the condition doesn't hold, the parent-child relationship has no choice to be decided by these two functions in recursive way.

$$parent(i) = \left\lfloor \frac{(i-2)}{k} + 1 \right\rfloor \quad (1)$$

$$child(i, j) = k(i-1) + j + 1 \quad (2)$$

2.3 Indexing with index interval

Jyh-Hong Tsay et al. [3] proposed an index method by using the index interval concept which assigns each node a pair of interval numbers - <*1st_index, 2nd_index*> according to the step numbers for reaching and leaving the node while traversing through the DOM tree in the preorder way. From this index interval concept, we can infer the relationship of parent-child in any level and the relationship of sibling in the same level by comparing the index interval numbers of specified nodes. They also proposed a document updating method – *Lazy update* by using the index update table to record the index update and index transformation information. This method lets the system does not need to update the index information immediately and could select suitable time to clean up the index table after the document is inserted, deleted, or modified. However, this method does not provide complete solution for inserting elements to the structured document and deleting elements from the structured document. Furthermore, they do not support the attribute query type for structured documents.

3 The indexing method

Indexing on structured documents for content, structure and attribute queries can be performed in various ways. In traditional text information retrieval systems, only the queries on content are supported. The rich structural information is contained in documents and the attributes of document components are not captured in these systems, and the queries on structure and attributes are not supported. For improving the index interval method [3] and adding the extra index information about the elements in XML documents, we use a kind of XML parser – DOM (Document Object Model) parser to build the index information and

propose five basic query functions to carry out the structure queries, provide information retrieval efficiently, and support more structure query types on structure documents.

3.1 Building the index information

The main idea of building the index information is that we use the DOM parser to parse the whole XML document into a tree structure, assign associated information (i.e. *index information = <document number, 1st_index, 2nd_index, level number>*) to each node in preordering way, and save the index information into three kinds of index information lists according to the node types – the element node, the attribute node and the text node. The index information of all element nodes is recorded into the ELEMENT list, the index information and attribute values of all element attribute nodes are recorded into the ATTRIBUTE list and the index information and content of all text node are recorded into the TEXT list. The concept of this processing is shown in Figure 3.

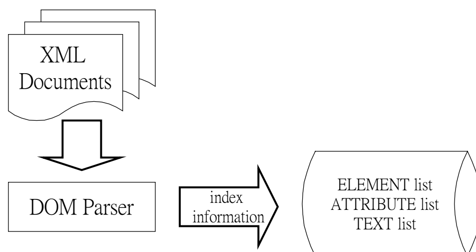


Figure 3:Building index information

The two interval numbers (*1st_index* and *2nd_index*) of index interval are the step numbers of reaching and leaving the node of the document tree while traversing in preorder way. This pair of index interval numbers could be used to determine what kind of relationships do specify nodes have by comparing the index interval numbers. We assume the two index interval numbers of nodes - U and V are U_{1st}, U_{2nd} and V_{1st}, V_{2nd} respectively. If $V_{1st} < U_{1st}$, and $U_{2nd} < V_{2nd}$, then the node U is a descendent of the node V. If $U_{1st} = V_{2nd} + 1$ or $V_{1st} = U_{2nd} + 1$, then the node U is a sibling of the node V. The document number tells which document the element belongs to, and the level number informs which level the element belongs to. An example of an XML document and its tree structure are shown in Figure 4 and Figure 5 respectively.

```
<?xml version="1.0"?>
<!DOCTYPE bib [
    <!ELEMENT bib (book+) >
    <!ELEMENT book (title, author+, publisher) >
    <!ATTLIST book year CDATA #REQUIRED >
    <!ELEMENT title (#PCDATA) >
    <!ELEMENT author (lastname) >
    <!ELEMENT lastname (#PCDATA) >
    <!ELEMENT publisher (name) >
    <!ELEMENT name (#PCDATA) >
]>
<bib>
  <book year="1995">
      <title> An Introduction to Database Systems </title>
      <author> <lastname> Date </lastname> </author>
      <publisher> <name> Addison-Wesley </name >   </publisher>
  </book>
  <book year="1998">
      <title> Foundation for Object/Relational Databases: The Third
              Manifesto </title>
      <author> <lastname> Date </lastname> </author>
      <author> <lastname> Darwen </lastname> </author>
      <publisher> <name> Addison-Wesley </name >   </publisher>
  </book>
</bib>
```

Figure 4:Example document

After building the index information, according to the node type number we separate the index information into three lists – ELEMENT, ATTRIBUTE, and TEXT. In this processing phase, the system does not store the *2nd_index* and *level number* into ATTRIBUTE and TEXT lists. Moreover, the system adds the attribute values after the corresponding index

information of the attribute. The content of these lists are shown in Figure 6. There is an additional temporary list – Q-TEXT that is not stored in the storage. We can use TEXT list to figure out Q-TEXT list for answering query requirements after loading these three index information lists into the memory.

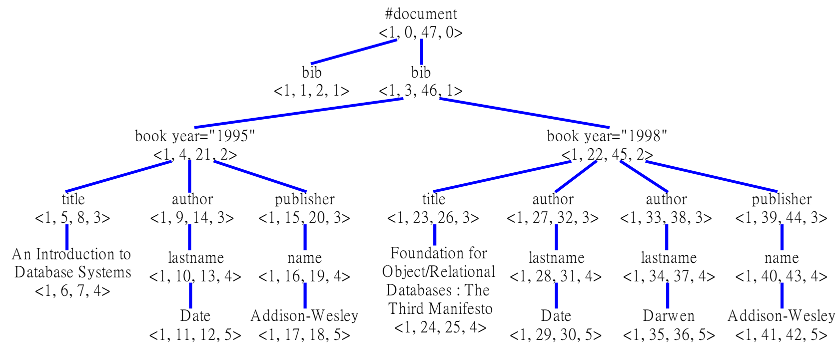


Figure 5:Tree structure of example document

ELEMENT index information:<Doc_No, 1st_index, 2nd_index, level_No>
bib > 1 3 46 1
book > 1 4 21 2 > 1 22 45 2
title > 1 5 8 3 > 1 23 26 3
author/lastname > 1 10 13 4 > 1 28 31 4 > 1 34 37 4
publisher/name > 1 16 19 4 > 1 40 43 4

ATTRIBUTE index information:<Doc_No, 1st_index, Att_Value>
year > 1 4 1995 > 1 22 1998

TEXT index information:<Doc_No, 1st_index>
An Introduction to Database Systems > 1 6
Date > 1 11 > 1 29
Addison-Wesley > 1 17 > 1 41
Foundation for Object/Relational Databases: The Third Manifesto > 1 24
Darwen > 1 35

Q-TEXT index information:<Doc_No, 1st interval>
1 6 > An Introduction to Database Systems
1 11 > Date
1 17 > Addison-Wesley
1 24 > Foundation for Object/Relational Databases: The Third Manifesto
1 29 > Date
1 35 > Darwen
1 41 > Addison-Wesley

Figure 6: The content of listings.

We will discuss the contents of these lists in following. In ELEMENT list, it contains the name of elements and its index information. If an element includes only one element, it could combine with its child element and keep the index information of its child. In this way, we can not only store the parent-child relationship of these two nodes but also save some storage spaces. In our example document, *author* and *lastname* elements are combined into one element – *author/lastname* and stored into ELEMENT list using the index information of lastname. The symbol "/" between *author* and *lastname* is used to show their parent-child relationship. The index information of TEXT list and Q-TEXT list just includes content of nodes, its document number and first index interval number. Because the second index interval number of TEXT nodes always equals to its first index interval number adds one. In this way, we not only reduce some space overhead but also infer the second index interval number of specific node from one's first index interval number. In order to keep the attribute value of the attribute node, the ATTRIBUTE list includes the name of attribute and its index information – document number, first index interval number, and attribute value. In the ATTRIBUTE list, we just keep the first index interval number, because we can only use the first index interval number to decide which element node includes this attribute node.

3.2 Five basic query functions

The XML documents are marked up with start-tags and end-tags, these markup tags are used to delimit the boundaries of structural elements or specify the values of attributes associated with structural elements. Based on these properties of XML documents, we propose five basic query functions for finding out the index information and inferring relationships of specified elements, attributes, or keywords. The description of these basic query functions and the corresponding examples based on the previous example document are shown in Table 2.

Table 2: The description and examples of basic query functions

Function	Description	Example
Match_Text	Return the index information of the specified word or the word of the specified index information in a document. Return NULL, if not matched.	1) *Match_Text ("Darwen")* Return: 1 35 2) *Match_Text (1, 11)* Return: Date
Match_Att	Return the index information of the specified attribute and its value in a document. Return NULL, if not matched.	*Match_Att ("year", "1998")* Return: 1 22 1998
Match_Ele	Return the index information of the specified element in a document. Return NULL, if not matched.	*Match_Ele ("title")* Return: 1 5 8 3 and 1 23 26 3
Include	Return index information of the specified element which includes the specified index information. Return NULL, if not found.	1) *Include ("publisher/name", 1, 41)* Return: 1 40 43 4 2) *Include ("book", 1, 16, 19, 4)* Return: 1 4 21 2
Included	Return index information of the specified element which is included in the specified index information. Return NULL, if not found.	*Included ("title", 1, 4, 21, 2)* Return: 1 5 8 3

3.3 Query on content, structure and attributes

So far, we have discussed the index method for extracting the index information of elements, attributes, and texts in a document. Moreover, we have proposed five basic query functions for finding out the index information and inferring the including relationship. Now we will describe how to use the index method and the five basic query functions for processing the higher-level user queries. Based on the previous example document, we take three examples to explain how to find out the results of query on content, structure and attribute as following. The first and second examples are combinations of content and structure query. The third one is a combination of structure and attribute query.

Example 1: Find out the title and author's name of every book whose publisher's name is "Addison-Wesley".

Step 1:

Call Match_Text ("Addison-Wesley")
Return: 1 17 and 1 41
Call Include ("publisher/name", 1, 17)
Return: 1 16 19 4
Call Include ("publisher/name", 1, 41)
Return: 1 40 43 4

Step 2:

Call Include ("book", 1, 16, 19, 4)
Return: 1 4 21 2
Call Include ("book", 1, 40, 43, 4)
Return: 1 22 45 2

Step 3:

Call Included (“title”, 1, 4, 21, 2)
Return: 1 5 8 3
Call Included (“author/lastname”, 1, 4, 21, 2)
Return: 1 10 13 4
Call Included (“title”, 1, 22, 45, 2)
Return: 1 23 26 3
Call Included (“author/lastname”, 1, 22, 45, 2)
Return: 1 28 31 4 and 1 34 37 4

Step 4:

Call Match_Text (1, 6)
Return: An Introduction to Database Systems
Call Match_Text (1, 11)
Return: Date
Call Match_Text (1, 24)
Return: Foundation for Object/Relational Databases: The Third Manifesto
Call Match_Text (1, 29)
Return: Date
Call Match_Text (1, 35)
Return: Darwen

In *Step 1*, after calling *Match_Text(“Addison -Wesley”)* function to get index information of keyword *“Addison-Wesley”*, the system checks to see if the keyword “*Addison-Wesley*” is a publisher’s name by call *Include(“publisher/n-me”, 1, 17)* and *Include (“publisher/name”, 1, 41)* functions. The *Step 2* is finding out which *book* includes the index information of results – 1 16 19 4 and 1 40 43 4 in Step 1 by calling *Include(“book”, 1, 16, 19, 4)* and *Include (“book”, 1, 40, 43, 4)* functions. The *Step 3* looks for what index information of “title” and “author/lastname” are included in index information of element *book* - 1 4 21 2 and 1 22 45 2 by calling *Included(“title”, 1, 4, 21, 2), Included(“author/lastname”, 1, 4, 21, 2)*, *Included(“title”, 1, 4, 21, 2)*, and *Included (“author/lastname”, 1, 4, 21, 2)* functions. The last *Step*, the system uses the document number and the first index interval number of returned index information of *Included()* functions in *Step 3* to extract the content of the *title* and *author’s name* for answering the user’s query by call *Match_Text()* function sequentially. Before the system calls the *Match_Text()* function, the system has to add 1 to the each first index interval number of returned index information in *Step 3*. These index information can be used to extract the corresponding content. For instance, the content of element *title>1 5 8 3* is *1 6>An Introduction to Database Systems*.

Example 2: Find out the title and publisher of every book whose author’s name is “Darwen”.

Step 1:

Call Match_Text (“Darwen”)
Return: 1 35
Call Include (“author/lastname”, 1, 35)
Return: 1 24 27 4

Step 2:

Call Include (“book”, 1, 24, 27, 4)
Return: 1 22 45 2

Step 3:

Call Included (“title”, 1, 22, 45, 2)
Return: 1 23 26 3
Call Included (“publisher/lastname”, 1, 22, 45, 2)
Return: 1 40 43 4

Step 4:

Call Match_Text (1, 24)
Return: Foundation for Object/Relational Databases: The Third Manifesto
Call Match_Text (1, 41)
Return: Addison-Wesley

The *Step 1* checks to see if the keyword *“Darwen”* is a author’s name and the *Step 2* checks which book includes the author name is *“Darwen”*. *Step 3* finds out the index information of book’s *title* and *publisher’ name* and uses these information for answering the user’s query in *Step 4*.

Example 3: Find out every book whose publishing year is “1995”.

Step 1:

Call Match_Att (“year”, “1995”)
Return: 1 4 1995
Call Include (“book”, 1, 4)
Return: 1 4 21 2

Step 2:

Call Included (“title”, 1, 4, 21, 2)
Return: 1 5 8 3
Call Included (“author/lastname”, 1, 4, 21, 2)
Return: 1 10 13 4
Call Included (“publisher/lastname”, 1, 4, 21, 2)
Return: 1 16 19 4

Step 3:
    Call Match_Text (1, 6)
        Return: An Introduction to Database Systems
    Call Match_Text (1, 11)
        Return: Date
    Call Match_Text (1, 17)
        Return: Addison-Wesley

In *Step 1*, the system will check the keyword "1995" is included in which book. The *Step 2* finds out the index information of book's *title*, *author*, and *publisher* and uses these index information for answering the user's query in *Step 3*.

4 Update method

After on user insert or remove some element of structured document, the document content and index information has to be changed. Once the structure changed, we have to do something to maintain the document content and index information integrations. Otherwise, we cannot retrieve the content correctly anymore. One of the update methods is to re-compute the index information of the document immediately. And then write the new index information back to the index information storage. This method is called *Immediate Update*, which can maintain the integration on index information of document and the content of document. But this way may waste too much overhead for frequent updating. Another update methods can avoid the expensive update cost and provide update index information more efficiently is called *Lazy update* [3]. *Lazy Update* use an index update table to record the index update and index transformation information, it let's the system does not need to update the index immediately. Then, the system can select suitable time to clean up the index information. However, Jyh-Hong Tsay et al. did not consider the validation of the structured document after inserting and removing the element of structured documents.

We take advantages of *Lazy Update* and modified the concept of *Lazy Update* to solve the problem on the validation of the structured documents and let the updating of index information more efficiently. The following are our different strategies on different cases.

(1) Modifying the specific content of some element

First of all, we have to find out the specific content position of the element in TEXT list and Q-TEXT list that we want to modify. Then we use the new data to replace the old data. In this case, we only need to do this modification immediately, because the structure of whole XML document does not be changed.

(2) Removing some element structure

In this case, we will use the update table to keep the index interval of the removed element. For example, we want to remove the first author of second book of example document in Figure 4 (i.e. Date). First, we have to find out which index interval of element includes the author's name "Date". Next, we record it's index information (i.e. <1, 27, 32, 3>) in the update table. Using the concept of update table a user still could issue the query after this operation, but the system needs to check that the element structure is removed from the index information lists or not before outputting the query result.

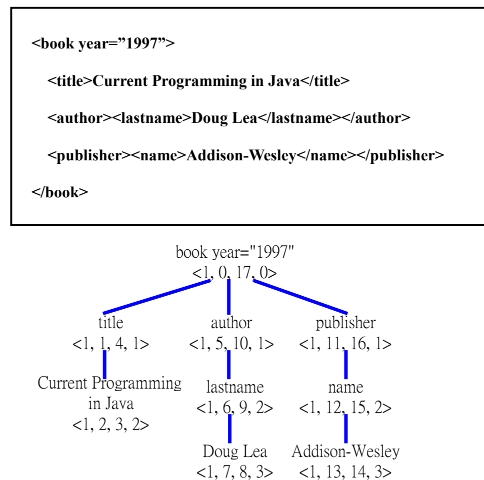
If we issue a query: *Find out the title and author's name of every book whose publisher's name is "Addison-Wesley"* (i.e. *Example 1* of section 3) after the above removing operation, the system will check

to see if the index interval of result data is included in the index interval which are recorded in the update table. So the result of *Example 1* doesn't contain *"Date"*.

(3) Inserting some element structure

In this case, the system will insert the element structure that we want to add into the XML document into the last of relative position of document tree. Because we have to consider the validation of XML document and insert the index information of the inserted element structure into the index information lists easily.

We will describe the concept of inserting an element structure by using the following element structure in Figure 7. Assume that we will insert the element structure into the example document in Figure 4. First of all, the system has to figure out the index information of tree structure of the inserted elements, assign the one to document number, and calculate the difference of index interval numbers of the root node (i.e. 17-0=17) in the inserted element structure. The concept of insertion operation and the result document tree of insertion operation are shown in Figure 8 and Figure 9 respectively. The inserted element structure is a sibling of second book in the example document tree. So the first index interval number of the root in inserted element structure is starting form 46 and it's level number is starting form 2. The system will add 46 to each index interval number, add 2 to each level number in the inserted element structure, add 18 (i.e. the difference of index interval numbers of root node in inserting element structure+1) to each index interval number great than 45 in the original index information lists, and then insert the index information of the inserted element structure into the last of the corresponding position of the ELEMENT, ATTRIBUTE, TEXT lists. The new content of all lists after insertion operation are shown in Figure 10.



```
<book year="1997">
  <title>Current Programming in Java</title>
  <author><lastname>Doug Lea</lastname></author>
  <publisher><name>Addison-Wesley</name></publisher>
</book>
```


Figure 7: Inserted element and structure

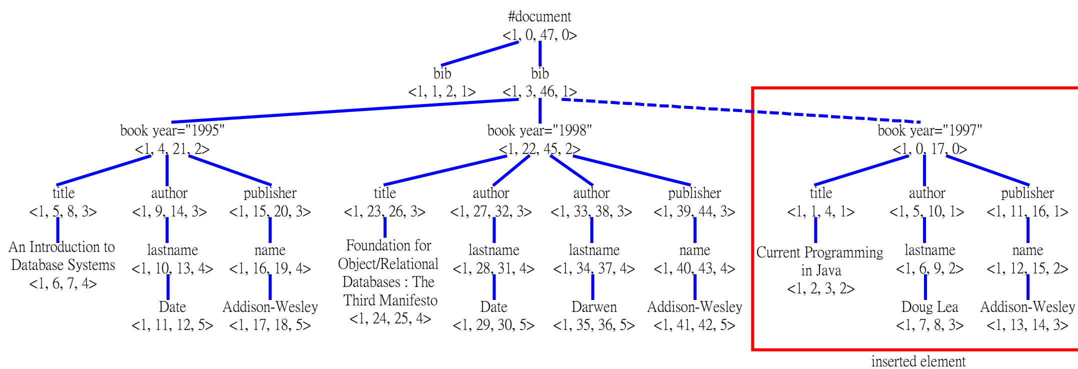


Figure 8: The concept of insertion operation

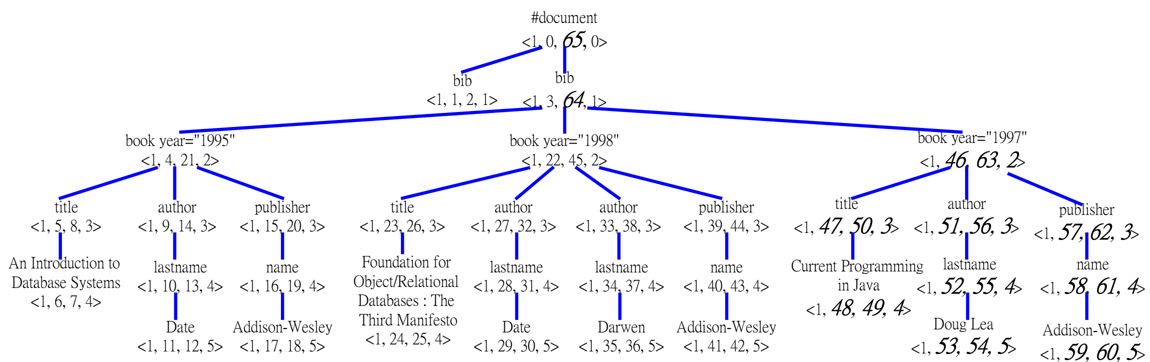


Figure 9: The result document tree of insertion operation

ELEMENT index information:<Doc_No, 1st_index, 2nd_index, level_No>
bib > ***1 3 64 1***
book > 1 4 21 2 > 1 22 45 2 > ***1 46 63 2***
title > 1 5 8 3 > 1 23 26 3 > ***1 47 50 3***
author/lastname > 1 10 13 4 > 1 28 31 4 > 1 34 37 4 > ***1 52 55 4***
publisher/name > 1 16 19 4 > 1 40 43 4 > ***1 58 61 4***

ATTRIBUTE index information:<Doc_No, 1st_index, Att_Value>
year > 1 4 1995 > 1 22 1998 > ***1 46 1997***

TEXT index information:<Doc_No, 1st_index>
An Introduction to Database Systems > 1 6
Date > 1 11 > 1 29
Addison-Wesley > 1 17 > 1 41 > ***1 59***
Foundation for Object/Relational Databases: The Third Manifesto > 1 24
Darwen > 1 35
Current Programming in Java > 1 48
Doug Lea > 1 53

Q-TEXT index information:<Doc_No, 1st interval>
1 6 > An Introduction to Database Systems
1 11 > Date
1 17 > Addison-Wesley
1 24 > Foundation for Object/Relational Databases: The Third Manifesto
1 29 > Date
1 35 > Darwen
1 41 > Addison-Wesley
1 48 > Current Programming in Java
1 53 > Doug Lea
1 59 > Addison-Wesley

Figure 10: The content of lists after insertion operation

5 Future work

So for, we have discussed the update method on the index information lists, but did not discuss how to update the XML document correspond to the changed index information lists. The reason is that we can answer user's queries just by the index information lists, not original XML document. Furthermore, we will propose the write back strategy to handle the integration problem of the original XML document and it's index information lists. Moreover, we will use the XML scheme for setting more data types on content of XML document on the document validation, and take the XQL or XML-QL [1] grammar to implement the system query language.

6 Conclusions

In order to perform structure queries efficiently on structured documents, we have proposed the new index method that makes the users can issue the content, structure and attribute queries on XML documents. We have shown how to handle these kinds of queries and how to use the strategies of update method for solving the update problem on element's insertion, element's removing and content's modification situations.

References

[1] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, Dan Suciu. XML-QL: A Query Language for XML. March 2001 *http://www.w3.org/TR/1998/NOTE-xml-ql-19980819*

[2] Dongwook Shin, Hyuncheol Jang, and Honglan Jin. BUS: An Effective Indexing

and Retrieval Scheme in Structured Documents. Dongwook Shin, Hyuncheol Jang and Honglan Jin; Proceedings of the third ACM Conference on Digital libraries, Pages 235 - 243, 1998

[3] Jyh-Jong Tsay and Gang-Heng Lu. Dynamic Indexing Schemes for Structured Documents. Master Thesis, Department of Computer Science and Information Engineering National Chung Cheng University. August 31, 2000

[4] S. Park, and Hyoung-Joo Kim. A new query processing technique for XML based on signature. Database Systems for Advanced Applications, 2001. Proceedings. Seventh International Conference, Pages: 22 –29, 2001

[5] Takeyuki Shimura, Masatoshi Yoshikawa and Shunsuke Uemura. Storage and Retrieval of XML Documents using Object-Relational Databases. July 2001.
http://citeseer.nj.nec.com/shimura99storage.htm

[6] Tuong Dao. An Indexing Model for Structured Documents to Support Queries on Content, Structure and Attributes. Research and Technology Advances in Digital Libraries, 1998. ADL 98. Proceedings. IEEE International Forum on 1998 Pages:88-97

[7] Yong Kyu Lee, Seong-Joom Yoo, Kyoungro Yoon. Index Structures for Structured Documents. Proceedings of the 1st ACM international conference on Digital libraries, Pages 91-99, 1996

[8] W3C. Document Object Model (DOM). July 2001.
http://www.w3.org/DOM/