A119

中華民國行政院文化建設委員會公佈
中文資訊交換碼(CCCII)維護技術資料

對CCCII控制碼的說明

謝清俊　張仲陶　楊鍵樵
黃克東　曾士熊

（資訊應用國字整理小組）

民國七十五年三月十三日

資訊應用國字整理小組

1．沿革

⑴民國六十九年四月，國字整理小組發表CCCII 第一冊〔1,2〕，當中對控制碼的規定包括：

①第一層（Plane）的第1、2段（Section）保留做為使用者增添專業用「標點符號」與「特別符號」的編碼之用．「特別符號」涵蓋中文資訊處理用控制字符（Control characters）．

②３位元組（Byte）延伸編碼的移轉程序（Escape Sequence）的對應控制碼規定為ESC 2/4 4/1.

⑵民國七十年二月，國字整理小組同時發表 CCCII的編碼結構、控制碼規則〔3〕以及CCCII第二冊第一版〔4〕．CCCII 第二冊第二版〔5〕（含編碼結構、控制碼規則等等），於民國七十年八月提出申請獲ISO認可，登記號碼為ISO/TC46/SC4/WG1/N.107. CCCII控制碼的規定：

①３位元組延伸編碼的移轉程序碼為ESC 2/4 4/1.

②CCCII 編碼結構中每層的第一段皆保留為中文資訊處理用控制碼，因此CCCII 雖是３位元組，但CCCII 控制碼卻是２位元組（詳見後文）．

③如有必要，３位元組 CCCII得由 CCCII控制碼指定預設層（即B3 default），而只儲存或傳輸B2與B1（表現段、編碼位置之位元組）．

⑶民國七十二年七月，國字整理小組所支援的中文文書編輯系統引用了一部份 CCCII控制碼〔6〕．

2．CCCII控制碼

⑴控制碼格式：

Ｂ３	Ｂ２	Ｂ１
☒	３３	３３－１２６

〔註1〕使用３位元組 CCCII時，B3無作用；使用２位元組 CCCII時，B3省略之（僅用B2，B1）．

〔註2〕本節中所用數字皆為十進制．

⑵控制碼處理程序：

```
Loop：Accept a CCCII；
      if B2=33 then
      call control-code-subroutine (parameter：B1)
      else begin
           if 33≦B3≦126 and 34≦B2≦126 and 33≦B1≦126
              then CCCII is a graphic code
              else report error messages
      end；
```

⑶CCCII控制碼的分類（暫訂）：

①本文分隔用控制碼（Text Separators）：用以將中文圖書內容劃分為篇、章、節、小節、段、句及詞的階層式組織；或者將中文圖書內文分為部、卷、冊、頁、行的階層式結構．

②版面效果用控制碼（Typesetting Effectors）：用於排版時的版面效果處理．

③編碼移轉用控制碼（Code-format Switchers）：用以將 CCCII的３位元組碼縮減為２位元組或單位元組．

④其它控制碼：尚待編訂．

⑷本文分隔用控制碼：

①用於將圖書內容劃分為篇、章、節、小節、段、句及詞的控制碼包括－

- DPA (Delimiter between Parts) ：用以劃分圖書內容為「篇」.
- DCH (Delimiter between Chapters) ：用以將一篇分隔為數「章」.
- DSE (Delimiter between Sections) ：用以將一章分隔為數「節」.
- DBL (Delimiter between Blocks/Subsections) ：用以將一節分隔為數「小節」.
- DPR (Delimiter between Paragraphs) ：用以分隔各小節中的「段落」.
- DST (Delimiter between Sentences) ：用以分隔段落中的「文句」.
- DWO (Delimiter between Words) ：用以分隔文句中的「詞」.

②用於將圖書內容劃分為部、卷、冊、頁及行的控制碼包括－－

- DTI (Delimiter between Titles) ：用以將成庫 (Library) 的圖書劃分為「部」.
- DCU (Delimiter between Chuns) ：用以將一部圖書分隔為「卷」.
- DVO (Delimiter between Volumes) ：用以將一卷圖書分隔為「冊」.

- DPG（Delimiter between Pages）：用以將一冊圖書分隔為「頁」.

- DLN（Delimiter between Lines）：用以將一頁的內容分隔成「行」.

〔註3〕卷譯為Chun，其義與Roll同.

(5)版面效果用控制碼：

- SCH（Skip one Character）：跳過一字，其作用形同ASCII的SP.

- SLN（Skip one Line）：跳過一行，其作用形同ASCII的CR/LF.

- CHS（Character Scale）：改變現有字體的級數，可用以放大或縮小後續的字；在CHS之後必須附加３或２位元組碼，其B3及B2無作用，僅以B1表現級數：級數的定義隨系統而異，但B1數值限制於33－126之間.

- CSR（Character Scale Recovery）：字體級數復原.

- CFS（Character-font Switch）：變換後續字的字體（如明體、宋體、楷書等），在CFS之後必須附加３或２位元組碼，其B3及B2無作用，僅以B1表現字體種類（值33-126）；B1數值所代表的字體隨系統而訂.

- CFR（Character-font Recovery）：字體復原.

- CDR（Character-display Reverse）：將印出字體由白底黑字反轉為黑底白字，或黑底白字反轉為白底黑字.

- CHL（Character High-light）：由終端機顯示中文字時，加強顯示亮度.

- HLR (High-light Recovery) ：字體顯示亮度復原為正常亮度.

⑹編碼轉移用控制碼：

①將CCCII由３位元組碼縮減為２位元組碼－－

- PSL (Plane-Shift, Locked) ：用以設定層 (B3) 的固有值 (Dfault Value) ，因而以後的CCCII皆不必再包含B3，而減為２位元組碼 (B1和B2) ；PSL之後必須附加２位元組碼 (B2和B1) ，其中B2無作用，而以B1指示新的層固有值．例如JIS6226被安置於CCCII的第92層 (B3=124) ，因此$\underbrace{[X]\ [33]\ [112]}_{\text{PSL}}$ $\underbrace{[Y]\ [124]}_{\text{參數}}$即可將CCCII轉為JISC6226碼，式中[ ]表示１位元組，X為33-126的任意數，Y為34-126的任意數．

- PSR (Plane-Shift Rcovery) 將Plane (B3) 的固有值設定取消，使CCCII由２位元組碼回復為３位元組碼．

- PSU (Plane-Shift, Unlocked) 與PSL功能相似，其差別在PSL可永久設定B3固定值，而PSU僅對跟隨其後的一字有效．
例如：$\underbrace{[X]\ [33]\ [114]}_{\text{PSU}}$ $\underbrace{[Y]\ [124]}_{\text{參數}}$ $\underbrace{[34]\ [58]}_{\text{JISC6226}}$ $\underbrace{[33]\ [70]\ [112]}_{\text{復原為CCCII}}$

[註4] 因CCCII的控制碼為２位元組,故CCCII由３位元組縮減為２位元組時所有控制碼功能不變．

[註5] PSL碼可應用於PC上，只２位元組即能處理CCCII．

②將CCCII由３位元組碼縮減為單位元組碼－－

- SSL (Section/Plane-shift, Locked) ：用以設定B3及B2的固有值，因而以後的CCCII只需B1即可；SSL之後須附加２位元組碼（B2和B1），其B2表示段的固有值，B1表示層的固有值．例如ASCII被安置於CCCII的第一層第三段（B3=33, B2=35），因此
$\underbrace{[X]\ [33]\ [115]}_{\text{SSL}}\ \underbrace{[35]\ [33]}_{\text{參數}}\ \underbrace{[65]}_{\text{"A"}}$表示ASCII的"A"，當然往後每個位元組皆視為ASCII碼．

[註6]一旦CCCII由３位元組碼縮減為單位元組碼之後、必須以ESC 2/4 4/1移轉程序才能使其復原為３位元組碼．

- SSU (Section/Plane-shift, Unlocked) ：與SSL相似，但為暫時性縮減；例如：

$$\underbrace{[X]\ [33]\ [115]}_{\text{SSU}}\ \underbrace{[35]\ [33]}_{\text{參數}}\ \underbrace{[65]}_{\text{"A"}}\ \underbrace{[33]\ [70]\ [112]}_{\text{復原為CCCII}}$$

③移面控制碼（Layer-shift Codes）：在CCCII已被縮減為２位元組碼的情況下，用以變換正、異體字，包括：

- LSL (Layer-shift, locked) ：用以改變B3固有值，LSL之後必須附加２位元組碼（B2和B1），其中B2無效，B3=(B3 mod 6)+(B1-33)*6，即B1=33表示移到第１面，B1=34表示移到第２面，.......

- LSU (Layer-shift, Unlocked) ：其功能與LSL相同，但為暫時性移面．

⑺CCCII控制碼表（暫訂）：

R\C	2	3	4	5	6	7
0			DWO	SCH		PSL
1			DST	SLN		PSR
2			DPR	CHS		PSU
3			DBL	CSR		SSL
4			DSE	CFS		SSU
5			DCH	CFR		LSL
6			DPA	CDR		LSU
7				CHL		
8			DLN	HLR		
9			DPG			
10			DVO			
11			DCU			
12			DTI			
13						
14						
15						

［說明］

B3：若CCCII為3位元組，B3值為33-126，但無作用．若CCCII為2位元組，B3省略之．

B2=33

B1=C*16+R

※參考文獻

[1] 國字整理小組，中文資訊交換碼，第一冊，中國國書館學會印行，民國六十九年四月，台北．

[2] Chinese Character Analysis Group,Chinese Character Code for Information Interchange, Vol. 1, Published by Library Association of China, April 1980, Taipei.

[3] Hsieh, Ching-Chun, et. al., "The Design and Application of the Chinese Character Code for Information Interchange (CCCII) ", International Workshop on Chinese Library Automation, Feb. 14-19, 1981, Taipei.

[4] Chinese character Analysis Group, Chniese Character Code for Information Interchange, Vol. II, 1st ed., Feb. 1981, Taipei.

[5] Chinese character Analysis Group, Chniese Character Code for Information Interchange, Vol. II, 2nd ed., Nov. 1982, Taipei.

[6] 王義科，中文文書編輯系統之研製，技術學院碩士論文，民國七十二年七月．

THE DOCUMENT REPRESENTATION AND A REFINED CHARACTER INVERSION METHOD FOR CHINESE TEXTUAL DATABASE

Shih-shyeng Tseng*, Chen-chau Yang** and Ching-Chun Hsieh***

* Computing Center, Academia Sinica,
Nankang, Taipei 11529, Taiwan, ROC
** Dept. of Electronic Eng., NTIT,
No. 43, Keelung Rd., Sec. 4, Taipei, Taiwan, ROC
*** Institute of Information Science, Academia Sinica,
Nankang, Taipei 11529, Taiwan, ROC

ABSTRACT

In this paper, the explicitly context-hierarchical organization (abbr. ECHO) and a refined character inversion method (abbr. ARCIM) will be introduced. The ECHO is a useful model to construct the text organization within a textual databae. It has the abilities to provide multiple context hierarchies for a document, a flexible search unit for retrieving textual information, and a subrange search on a textual database. In addition, the ECHO tree is relatively easy to maintain. The word inversion is the retrieval method that has been most commonly used in English textual databases. Its advantages are that it is relatively easy to implement and is fast. However, the word inversion is not suitable for accessing Chinese texts; instead, the character inversion is possible. ARCIM can retrieve texts faster and use smaller storage overhead than the original character inversion method.

1. THE CHARACTERISTICS OF TEXTUAL DATA

The differences between a conventional database and a textual database, which discussed in [1] and [2], can be summarized as follows. The conventional database and the textual database deal with retrieval and manipulation of formatted data (records) and textual data (documents), respectively. And the formatted records differ from the text environment in the structure of data, the query language, and operational requirements such as update frequency and size of database. In [1] and [2], a document is considered as a non-structured data consisting of an arbitrary number of words. This discussion is not true at all. The document is also a structured data though its structure is not similar to the sturcture of formatted records.

The structure of document is hierarchical [3]. In general, there are a unique content structure and at least one typesetting structure embedded in a document. The content structure, an important constituent to represent the conceptual skeleton of document, is conceived as a tree-like structure having distinct levels of text elements such as document, chapter, section, subsection, paragraph, etc. An example of content structure is shown in Figure 1. The typesetting structure of a text reflects its positioning in a representation medium. For example, a page forms the representation unit of the document contents. A number of pages constitute a set which may be a chapter, a preface, or a table of contents, as shown in Figure 2. A document may have more than one typesetting structures, in which each structure represents the prositioning in an individual medium and for an individual version.

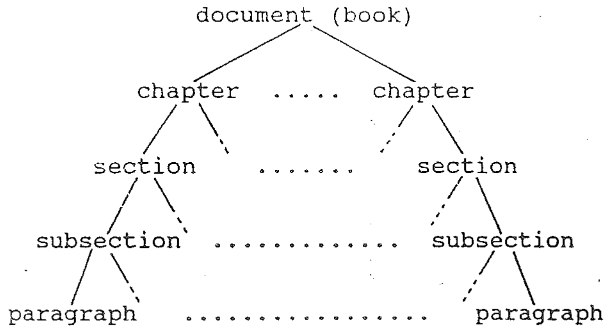

Figure 1 An example of content structure

From the viewpoint of formalism, both the content structure and the typesetting structure are hierararchies of contexts. A context is defined as a piece of text or the whole text within a document, with the property of hierarchy. That is, any context must be fully contained in a higher-level context, except the root context which denotes the whole document. It is apparent that a leaf context does not contain any lower-level context. The context hierarchy of a document can be realized as a tree structure, called context tree, in which each node denotes a context of the document. The context tree has the following properties.

(1) *The node B is a descendant of the node A if and only if the context B is wholly contained in the context A.*

(2) *For any node, the number of children is not constant. That is, a context contains a variable number of lower-level contexts.*

(3) *For any context, its length is not constant.*

(4) *For any node, the order of its children cannot be changed.*

(5) *If node B_1 , B_2 , ..., B_m are children of the node A, then the length of context A is the sum of the lengths of contexts B_1 , B_2 , ..., B_m .*

In addition, the context trees in a textual database, in which each context tree represents a document, can be grouped into a larger context tree by adding a dummy root or a dummy higher context tree beyond all the original context trees. The original context trees then become the subtrees of the new larger context tree. In this case, the properties of the context tree, discussed above, also hold.

2. THE ECHO MODEL

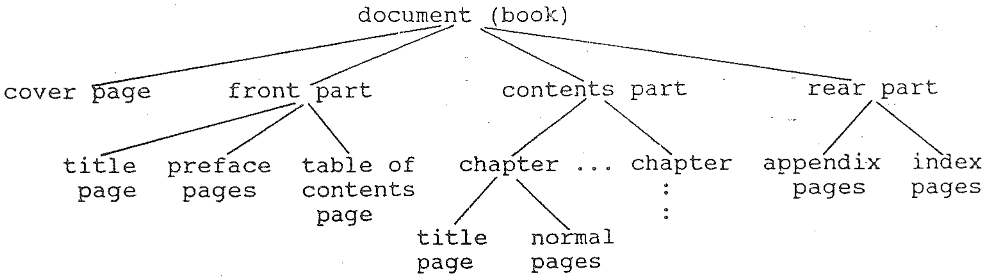

Figure 2 An example of typesetting structure

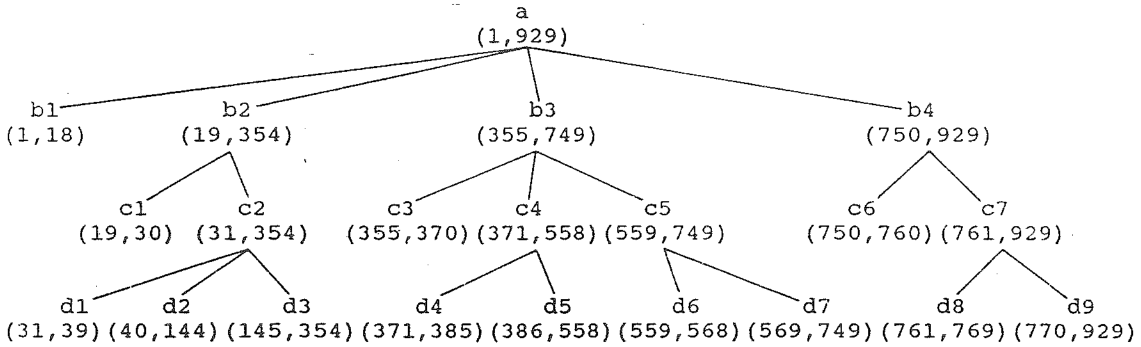

Figure 3 An example of context tree

The contexts are the constituents for retrieving textual information. For example, we always inquire which documents, which paragraphs, or which pages contain the terms that we concerned. In order to provide a necessary mechanism, the texts in a textual database have to be organized as a context hierarchy. There are two possible methods to implement the context organization : implicit representation and explicit representation. They will be discussed in the following subsection.

2.1 THE ECHO TREE

In the implicit representation, a set of specific context delimiters such as described in [4], must be inserted into the text in order to identify each context. A context is then surrounded by a pair of beginning and ending delimiters. The text scanner can thus be used to find the beginings and ends of contexts. In addition, a higher-level context which contains the current context or a lower-level context which is contained in the current context can also be searched by scanning the text forward and backward. The advantages of the implicit representation are : the context organization is simple, requires no space overhead, and is easy to maintain. However, the price paid is the bad response time.

Instead of using the embedded context delimiters, the explicit representation uses an additional context tree to reflect the context organization of a document [5]. In the context tree, each node which denotes one context of the document has a name (or number) and a pair of pointers which point to the beginning and ending positions of the context, respectively. The pathname of a node, which is the list of node-names from the root to this node, is referred as the context-id of the context that it deontes. Suppose that the context-id of the context N_k is $N_1\ N_2\ ....N_k$. A sublist of this context-id, say $N_1\ N_2\ ....N_i$, $i<k$, is called a proper prefix of the context-id $N_1\ N_2\ ....N_k$. If a context N_i with the context-id $N_1\ N_2\ ....N_i$ which is a proper prefix of the context-id of the context N_k , then the context N_i is one of the ancestors of the context N_k . An example of the context tree is shown in Figure 3. By means of the context tree, the beginning and end of a context are easily located when the context-id is given. The higher-level and lower-level contexts of the current context are also easy to search. In addition, some sohpisticated retrieval methods can be applied to impove the retrieval performance. The disadvantages of the explicit representation are that it requires space overhead for storing the context trees, and the number of O(n) nodes must be maintained when a context is updated, where n is the number of nodes in a context tree.

The ECHO (explicitly context-hierarchical organization) is an modified explicit representation. It is proposed in order to reduce the maintenance cost. Instead of the absolute addressing scheme used in the original explicit representation, the ECHO uses a relative addressing scheme. In the ECHO tree, each node has a name (or number) and a vector (D,L) which replaces the pair of pointers discussed above. The D denotes the distance between the beginning position of the current context and the beginning position of the parent context of this context. For the root, D is set to value "1". If a node is the leftmost child of its parent, then the D-value of this node has to be set to value "0" [from the property(5) of the cnotext tree, which was discussed in Section 1]. The L denotes the length of the current context. Except the leaf contexts, the L-value of a given context is the sum of all the L-values of its children, i.e., the property(5) of the context tree. An example of ECHO tree which is equivalent to the context tree shown in Figure 3, is shown in Figure 4.

2.2 THE OPERATIONS ON ECHO TREES

The operations on ECHO trees can be classified into three types : (1) the search functions, (2) the tree-constructing operations, and (3) the tree-updating operations.

2.2.1 The Search Functions

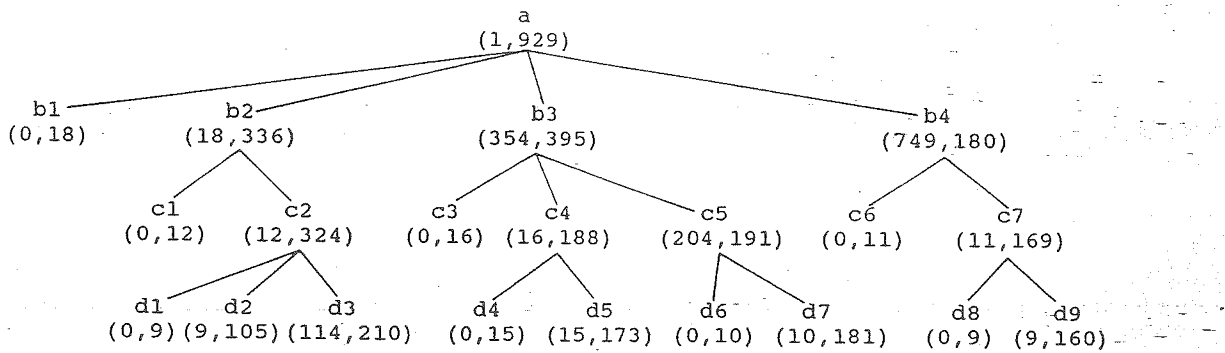

Figure 4 The ECHO tree corresponding to the context tree shown in Figure 3

The ECHO model provides two search functions : (1) find a context by giving its context-id, and (2) find the contexts which contain a text by giving its beginning and ending pointers. For the first search function, the context can be accessed by means of its beginning and eding positions. Let the context-id of the context N_k be $N_1 N_2 ....N_k$. The beginning position, denoted BP(x), and the ending position, denoted EP(x), of the context N_k can be derived by the Formula-1 and Formula-2.

[Formula-1] $BP(N_k)=D(N_1)+D(N_2)+....+D(N_k)$

[Formula-2] $EP(N_k)=BP(N_k)+L(N_k)-1$

Where D(x) and L(x) denote the D-value and L-value of the context, respectively. Let's take the context d_7 shown in Figure 4 as an example. The context-id of d_7 is $ab_3 c_5 d_7$, thus

$BP(d_7)=D(a)+D(b_3)+D(c_5)+D(d_7)=1+354+204+10= 569,$
and
$EP(d_7)=BP(d_7)+L(d_7)-1=569+181-1 =749.$

The vector (569,749) is the same as the pointers-pair of the context d_7, shown in Figure 3.

For the second search function, suppose a text T with the beginning position, BP(T), and the ending position, EP(T), are given. This search function can be considered as to find two contexts, denoted C1 and C2, such that C1 "contains" BP(T) and C2 "contains" EP(T); that is,

$BP(C1)\leq BP(T)\leq EP(C1)$ *and*
$BP(C2)\leq EP(T)\leq EP(C2)$.

And then all the contexts from C1 to C2 are retured as the result.

2.2.2 The Tree-contstructing Operations

The tree-constructuing operations inculde (1) to construct the ECHO tree from a given document, and (2) to combine a number of ECHO trees to form a larger ECHO tree. In order to construct the context trees from a given document, a set of markup symbols must be inserted into the document. And then a context parser scans this markupped document to recognize each markup symbol and construct the context tree [5]. The details of the markup symblos and markup rules can be found in [3], [5] and [6]. The context parser is too complicated to discuss in this paper. Thus we assume that the context tree of a document is constructed by the context parser, and the length of each leaf context is known. The Algorithm-1 can be used to compute the vector (D,L) of each context in the context tree and then tranfer the context tree into an ECHO tree.

```
Algorithm-1: Compute the vector (D,L) of each context.
Given: A context tree and the length of each leaf context.
Procedure: Let $D(root)=1$ and then traverse the context tree in post
           order: (1) If the context visited X is the leftmost child of
           its parent, then let $D(X)=0$; else let $D(X)= D(Y)+ L(Y)$,
           where Y denotes the nearest left sibling of X. (2) If the
           context visited X is not a leaf context, then let
           $L(X)=D(Z)+L(Z)$, where Z denotes the context last
           visited (i.e., the rightmost child of X).
```

The combining of a number of ECHO threes can be considered as the concatinating of texts. The internal structure and the length of each text are still the same, but the positions of those texts except the first text are changed. Suppose there are a number of k ECHO trees combined into a larger ECHO tree, the root of each original ECHO tree is denoted H_i , $1\leq i\leq k$, corresponding with the sequence in the combination. The combining operation is then provided by the Alogrithm-2.

```
Alogrithm-2: Combine k individual ECHO trees into a larger one.
Step-1: Create a new root context R, and then link the root of each
        original ECHO tree to R.
Step-2: Let $D(R)=1$ and let $L(R)=L(H_1)+L(H_2) +..... + L(H_k)$.
Step-3: For i=2 to k do $D(H_i)=D(H_{i-1})+ L(H_{i-1})$.
Step-4: Let the first text be the result text.
Step-5: For i=2 to k do append the i-th text to the result text and form
        a new result text.
```

2.2.3 The Tree Updating Operation

The maintenance of a database involves insertion, deletion, and modification. The ECHO tree of a document (or textual database) has to be updated when any of the following situations occurred : (1) a new context (or document) is inserted into the document (or database), (2) an old context (or document) is deleted from the document (or database), and (3) a context of the document is modified. In addition, the retrieval organization such as the character index table and the posting lists table, which will be discussed in Section 3, may also be updated when the contents of document (or textual database) is changed.

Before the insertion is made, a new context (or document), say X, must be prepared with that all the necessary markups are available. The insertion operation is then executed, consisting of two phases :

(1) Parsing phase : The ECHO tree of X is built by means of the constructing operation as discussed in Subsection 2.2.1.

(2) Insertion phase : The context (or document) X is then inserted into the designated position on the document (or database) H. At the same time, the ECHO tree X, as a subtree, also has to be inserted into the corresponding location on the ECHO tree H and forms an extended tree H´. At last, some of (D,L) vectors on H´ must be updated, discussed as follows.

After the text X is inserted into the H, the following situations occur : (1) the positions of all the right siblings of X are moved right L(X) from the original positions, (2) the lengths of all the ancestors of X, say Y for any, are increased by L(X), and (3) the positions of all the right siblings of Y are moved right L(X) from the original positions. These situations must be reflected when the subtree X is inserted into the ECHO tree H. That is, the (D,L) vectors of the corresponding nodes must be then updated.

When a context (or document) X is deleted from the document (or database) H the following situations will occur : (1) the positions of all the right siblings of X are moved left L(X) from the original positions, (2) the lengths of all the ancestors of X, say Y for any, are decreased by L(X), and (3) the positions of all the right siblings of Y are moved left L(X) form the original positions.

The modification of a context is actualy realized by modifying some leaf contexts of it, say X for any. We call the modified leaf context X'. The modification of X may cause the change of L(X), i.e., L(X')≠L(X). The ECHO tree of the original document (or database) need not to be updated if L(X')=L(X). But the ECHO tree must be updated when L(X')≠L(X). The Alogrithm-3 illustrates the modification operation on the ECHO trees.

Alogrithm-3: *Update the ECHO tree H when a leaf context X is modified and L(X)≠L(X').*

```
Step-1: Replace the L(X) by L(X').
Step-2: For each right sibling of X', called Z, let D(Z)=D(Z)-L(X)+
        L(X').
Step-3: For each parent of X', called Y, let L(Y)=L(Y)-L(X)+
        L(X'). And for each right sibling of Y, called W, let
        D(W)=D(W)-L(X) +L(X').
```

Suppose the number of contexts on an ECHO tree H is N, and the height of H is log N. The number of nodes from the root to any leaf context is then log N. Therefore for any leaf context X, the number of nodes inculding node X, all the right siblings of X, all the ancestors of X, and all the right sibilings of the ancestors of X, is O(log N). It is suitable to conclude that only O(log N) nodes of an ECHO tree have to be updated when a context is inserted, deleted, or modified.

3. A REFINED CHARACTER INVERSION METHOD FOR CHINESE TEXT

The retrieval methods for text can be classified into five categories : full text scanning, inversion of terms, multiattribute hashing, signature files, and clustering [1,7]. The following discussion for inversion method is summarized from [1], [7], and [8]. The inversion method for English text uses an index in which each entry conssists of a word along with a list of pointers, called posting list. These pointers point to the contexts that contain this word. The advantages of the word inversion are that it is relatively easy to implement, is fast, and provides synonyms easily. For these reasons, the word inversion has been adopted in most commercial systems such as BRS, DIALOG, MEDLARS, ORBIT and STAIRS [8]. But the disadvantages of the word inversion are : (1) the storage overhead (50-300% of the original file size [9]), (2) the cost of updating and reorganizing the index, if the environment is dynamic, and (3) the cost of merging the posting lists, if they are too long or too many.

However, the word inversion is not suitable for Chinese text because (1) a Chinese character string does not contain any natural delimiters, such as blanks in an English sentence, to separate Chinese words, and (2) an effective and automatic word segmentation method for Chinese sentences has not been derived. Thus we use character inversion instead of word inversion to access Chinese text. In the following, a refined character inversion method (abbr. ARCIM) will be introduced. It can provide a faster access speed and needs smaller storage overhead than the conventional character inversion method. Besides, it can reduce the cost of updating the index table and the posting lists table.

3.1 THE SEARCH OPERATION ON ARCIM

Similar to the word inversion used in English text, the ARCIM uses a character index table, called CIT, and a posting lists table, called PLT, as shown in Figure 5. An entry of the PLT, denoted PL(C_i), is a posting list consisting of a variable number of context-id's, in which these context-id's are always kept sorted. Each context-id of the PL(C_i), say X for any, means that the context X contains the character C_i . In the CIT, each entry consists of a count and a pointer, denoted count(C_i) and pointer(C_i), respectively. The pointer(C_i) points to the starting location of the corresponding PL(C_i) and the count(C_i) denotes the number of context-id's of this PL(C_i). If there exists a character C_j which does not appear in any context, the count(C_j) and the pointer(C_j) will be set to zero and nil, respectively, and no PL(C_j) will appears in the PLT. If there exists a character C_k which appear in most contexts, say more than a predefined threshold ratio t% to the total number of contexts in the database, then the count(C_k) and pointer(C_k) are set to -1 and nil, respectively. And the PL(C_k) must be then removed from the PLT. The size of the CIT is constant since the number of Chinese characters used in a computer system must be limited to a constant. Thus the CIT may be permanently located in the main memory as a system table. When a character C_i is given, the count(C_i) and pointer(C_i) can be accessed by means of a specific hash function which depends upon the coding scheme of the Chinese characters. An example of the hash function can be found in [10].

The ARCIM can provide an ability of free term search. The followings are rules for search expressions.

<search-expression>::=<phrase> {OR <phrase>}
<phrase>::=<term> {AND [NOT] <term>}
<term>::=<string> | <wild-card-term> | <ordered -term>

Where <....> denotes a token, [....] denotes an optional item, {....} denotes a repetition of any times, and the verticle bar means "or". By applying the Alogrithm-4, a given search expression can be evaluated and then a set of contexts which satisfy this expression will be obtained.

Algorithm-4: *Evaluate a given search expression by means of a bottom-up and greedy manner and then obtain a set of contexts which satisfy this expression.*

```
Step-1: Partition the search expression into a set of phrases.
Step-2: Reconstruct each phrase which is obtained form Step-1.
Step-3: For each reconstructed phrase, perform the character-level
        search operations and AND-merge operations to obtain a
        posting list on the phrase level.
```

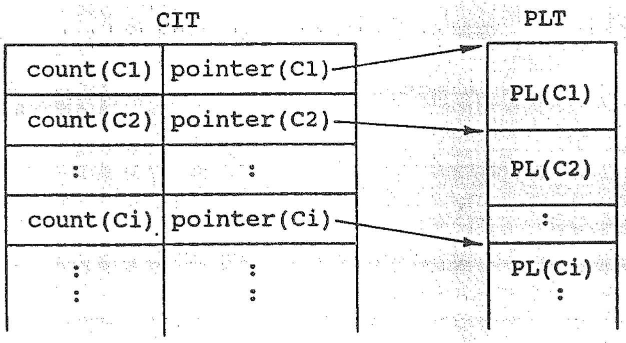

Figure 5 The index organization of ARCIM

Step-4: OR-merge all phrase-level posting lists to obtain a posting list as result.
Step-5: In posting list thus obtained, scan each context and remove the contexts which don't satify the original search expression.

The Step-2 is an important step. To evaluate a reconstructed phrase needs less posting lists and less AND-merge operations operations than the evaluation of the original phrase. Hence, this will reduce the time consummed in the Step-3. The algorithm for Step-2 is shown as follows.

Algorithm-5: *Reconstruct a given phrase.*
Step-1: Eliminate every term which follows the "NOT" operator and all operators from the given phrase.
Step-2: Eliminate repeated characters.
Step-3: If there is any character C_i with count(C_i)=-1, delete it.
Step-4: Sort the remaining characters in nondecreasing order of counts.

The reconstructed phrase, say b_1 b_2 ....b_k (each b_i, $1 \leq i \leq k$, denotes a character) has the following properties.

(1) $b_i \neq b_j$ *for* $i \neq j$.
(2) *count*$(b_i) \leq$ *count*(b_j) *for* $i<j$.
(3) $PL(b_1 b_2 ... b_i) = PL(b_1) \cap PL(b_2) \cap .. \cap PL(b_i)$ *for* $1 \leq i \leq k$.
(4) $PL(b_1 b_2 ... b_k) \subseteq PL(b_1 b_2 ... b_{k-1}) \subseteq ... \subseteq PL(b_1)$.
(5) *count*$(b_1)=0$ *implies* $PL(b_1)=\emptyset$ *imples* $PL(b_1 b_2 .... b_k)=\emptyset$.

The properties mentioned above are very important to the Step-3 of the Alogrithm-4. According to the property(5), if count(b_1)=0, we rather immediately let the phrase-level posting list be null than apply the Step-3 of the Alogrithm-4. According to the property(4), if count(b_1) count(b_1)≠0 we search PL(b_1) and PL(b_2) and AND-merge them first, then search PL(b_3).

3.2 CREATION AND MAINTENANCE ON PLT AND CIT

In order to provide the ARCIM as the underlying retrieval mechanism, the PLT and CIT must be created while a Chinese textual database is built. The Algorithm-6 can be used to create the PLT and CIT when the text files and the ECHO tree are available.

Algorithm-6: *Create the PLT and CIT from the text files and the ECHO tree.*
Step-1: Create an initial CIT in which the number of entries must be equal to the number of Chinese characters used in the computer system as well as for each entry let count(C_i)=0 and pointer(C_i)=nil.
Step-2: For each leaf countext X do
(1) eliminate all blanks, punctuation symbols, and other unnecessary symbols, the remainds are called X';
(2) eliminate all repeated characters from X', the remainder are called the character list of X.
Step-3: For character list of leaf context X, append the context-id X to each character of this list; the result is called character context-id list of X.
Step-4: According to the sequence from the smallest context-id to the largest context-id, concatenate all the character context-id lists. The result is called the character context-id lists table.
Step-5: Sort the character context-id lists table by a nondecreasing sequence of the character codes. The result is called the initial PLT. A segment of the initial PLT in which each entry has the same character C_i and different context-id is called the initial PL(C_i).
Step-6: Scan the initial PLT to find the starting address (in number of entries) of each initial PL(C_i) and compute the count(C_i).

If the value of count(C_i)/Q is more than a predefined threshold ratio t, then let count(C_i)=-1 and pointer(C_i)=nil, where Q denotes the total number of leaf contexts. In this case, remove the initial PL(C_i) from the initial PLT. Write the count(C_i) and pointer(C_i) into the corresponding entry of the initial CIT. During the scanning, delete the character field from each entry of the initial PLT.

The CIT and PLT must be updated when a context is inserted, deleted, or modified. The Algorithm-10, -11, and -12 describe the update operations for the case of insertion, deletion, and modification, respectively.

Algorithm-7: *Update the CIT and PLT when a leaf context X is inserted into the database.*
Step-1: Construct the character list of X by means of Step-1 of the Algorithm-6. And then sort this list by an ascending order of character codes. Suppose the characters within the sorted character list are $b_1 b_2 .... b_k$.
Step-2: Insert the context-id X into each PL(b_i), $1 \leq i \leq k$, keeping the ascending order of context-id's on each PL(b_i).
Step-3: Scan the CIT from the character b_1 to the last. Suppose the character that currently be sacnned is a. If $a=b_i$, $1 \leq i \leq k$, then let count(a)= count(a)+1. If $b_i < a \leq b_{i+1}$, $1 \leq i < k$, then let pointer(a)=pointer(a)+i. For each $a > b_k$, let pointer(a)=pointer(a)+k.

Algorithm-8: *Update the CIT and PLT when a leaf context X is deleted from the database.*
Step-1: Same as the Step-1 of the Algorithm-7.
Step-2: Delete the context-id X from each PL(b_i), $1 \leq i \leq k$.
Step-3: Scan the CIT from character b_1 to the last. Suppose the character currently scaned is a. If $a=b_i$, $1 \leq i \leq k$, let count(a) = count(a)-1. If $b_i < a \leq b_{i+1}$, $1 \leq i < k$, let pointer(a) = pointer(a)-i. For each $a > b_k$, let pointer(a) = pointer(a)-k.

Algorithm-9: *Update the CIT and PLT when a leaf context X is modified.*
Step-1: Construct the character lists of the old context and of the new context, respectively. Then sort these two character lists by ascending order of character codes. As a convertion, we call the sorted character lists of old context and new context CL(X) and CL(X'), respectively.
Step-2: Compare CL(X) with CL(X') to obtain a string $b_1 b_2 .... b_k$ = CL(X')-CL(X) and a string $d_1 d_2 .... d_j$ =CL(X)-CL(X'). The string b_1 b_2 ....b_k here denotes the characters that are inserted into X, while the string d_1 d_2 ....d_j denotes the characters that are deleted from X.
Step-3: Apply the Step-2 and -3 of the Algorithm-7 for the string $b_1 b_2 ... b_k$.
Step-4: Apply the Step-2 and -3 of the Algorithm-8 for the string $d_1 d_2 ... d_j$.

4. CONCLUSION

The following are the advantages of the ECHO model:

(1) Multiple context hierarchies of a document can be provided by the ECHO model, for example, context structures and typesetting structures.
(2) Any level of contexts can be located by menas of the ECHO tree. Thus the ECHO has the ability to provide a flexible unit for retrieving the textual information.
(3) The ECHO can do subtree search to speed up the retrieval performance. In other words, users can specify a subset of the database as the search range.
(4) The ECHO tree is relatively easy to maintain. There are just O(log n) modes of an ECHO tree to be updated when a context is inserted, deleted, or modified, where n is the number of nodes in an ECHO tree.

Step-4: OR-merge all phrase-level posting lists to obtain a posting list as result.

Step-5: In posting list thus obtained, scan each context and remove the contexts which don't satify the original search expression.

The Step-2 is an important step. To evaluate a reconstructed phrase needs less posting lists and less AND-merge operations operations than the evaluation of the original phrase. Hence, this will reduce the time consummed in the Step-3. The algorithm for Step-2 is shown as follows.

Algorithm-5: *Reconstruct a given phrase.*

Step-1: Eliminate every term which follows the "NOT" operator and all operators from the given phrase.

Step-2: Eliminate repeated characters.

Step-3: If there is any character C_i with $count(C_i) = -1$, delete it.

Step-4: Sort the remaining characters in nondecreasing order of counts.

The reconstructed phrase, say $b_1\ b_2 \ldots b_k$ (each b_i, $1 \leq i \leq k$, denotes a character) has the following properties.

(1) $b_i \neq b_j$ *for* $i \neq j$.

(2) $count(b_i) \leq count(b_j)$ *for* $i < j$.

(3) $PL(b_1 b_2 \ldots b_i) = PL(b_1) \cap PL(b_2) \cap \ldots \cap PL(b_i)$ *for* $1 \leq i \leq k$.

(4) $PL(b_1 b_2 \ldots b_k) \subseteq PL(b_1 b_2 \ldots b_{k-1}) \subseteq \ldots \subseteq PL(b_1)$.

(5) $count(b_1) = 0$ *implies* $PL(b_1) = \emptyset$ *imples* $PL(b_1 b_2 \ldots b_k) = \emptyset$.

The properties mentioned above are very important to the Step-3 of the Alogrithm-4. According to the property(5), if $count(b_1) = 0$, we rather immediately let the phrase-level posting list be null than apply the Step-3 of the Alogrithm-4. According to the property(4), if $count(b_1)$ $count(b_1) \neq 0$ we search $PL(b_1)$ and $PL(b_2)$ and AND-merge them first, then search $PL(b_3)$.

3.2 CREATION AND MAINTENANCE ON PLT AND CIT

In order to provide the ARCIM as the underlying retrieval mechanism, the PLT and CIT must be created while a Chinese textual database is built. The Algorithm-6 can be used to create the PLT and CIT when the text files and the ECHO tree are available.

Algorithm-6: *Create the PLT and CIT from the text files and the ECHO tree.*

Step-1: Create an initial CIT in which the number of entries must be equal to the number of Chinese characters used in the computer system as well as for each entry let $count(C_i) = 0$ and $pointer(C_i) = nil$.

Step-2: For each leaf countext X do

(1) eliminate all blanks, punctuation symbols, and other unnecessary symbols, the remainds are called X';

(2) eliminate all repeated characters from X', the remainder are called the character list of X.

Step-3: For character list of leaf context X, append the context-id X to each character of this list; the result is called character context-id list of X.

Step-4: According to the sequence from the smallest context-id to the largest context-id, concatenate all the character context-id lists. The result is called the character context-id lists table.

Step-5: Sort the character context-id lists table by a nondecreasing sequence of the character codes. The result is called the initial PLT. A segment of the initial PLT in which each entry has the same character C_i and different context-id is called the initial $PL(C_i)$.

Step-6: Scan the initial PLT to find the starting address (in number of entries) of each initial $PL(C_i)$ and compute the $count(C_i)$.

If the value of $count(C_i)/Q$ is more than a predefined threshold ratio t, then let $count(C_i) = -1$ and $pointer(C_i) = nil$, where Q denotes the total number of leaf contexts. In this case, remove the initial $PL(C_i)$ from the initial PLT. Write the $count(C_i)$ and $pointer(C_i)$ into the corresponding entry of the initial CIT. During the scanning, delete the character field from each entry of the initial PLT.

The CIT and PLT must be updated when a context is inserted, deleted, or modified. The Algorithm-10, -11, and -12 describe the update operations for the case of insertion, deletion, and modification, respectively.

Algorithm-7: *Update the CIT and PLT when a leaf context X is inserted into the database.*

Step-1: Construct the character list of X by means of Step-1 of the Algorithm-6. And then sort this list by an ascending order of character codes. Suppose the characters within the sorted character list are $b_1\ b_2 \ldots b_k$.

Step-2: Insert the context-id X into each $PL(b_i)$, $1 \leq i \leq k$, keeping the ascending order of context-id's on each $PL(b_i)$.

Step-3: Scan the CIT from the character b_1 to the last. Suppose the character that currently be sacnned is a. If $a = b_i$, $1 \leq i \leq k$, then let $count(a) = count(a) + 1$. If $b_i < a \leq b_{i+1}$, $1 \leq i < k$, then let $pointer(a) = pointer(a) + i$. For each $a > b_k$, let $pointer(a) = pointer(a) + k$.

Algorithm-8: *Update the CIT and PLT when a leaf context X is deleted from the database.*

Step-1: Same as the Step-1 of the Algorithm-7.

Step-2: Delete the context-id X from each $PL(b_i)$, $1 \leq i \leq k$.

Step-3: Scan the CIT from character b_1 to the last. Suppose the character currently scaned is a. If $a = b_i$, $1 \leq i \leq k$; let $count(a) = count(a) - 1$. If $b_i < a \leq b_{i+1}$, $1 \leq i < k$, let $pointer(a) = pointer(a) - i$. For each $a > b_k$, let $pointer(a) = pointer(a) - k$.

Algorithm-9: *Update the CIT and PLT when a leaf context X is modified.*

Step-1: Construct the character lists of the old context and of the new context, respectively. Then sort these two character lists by ascending order of character codes. As a convertion, we call the sorted character lists of old context and new context CL(X) and CL(X'), respectively.

Step-2: Compare CL(X) with CL(X') to obtain a string $b_1\ b_2 \ldots b_k = CL(X') - CL(X)$ and a string $d_1\ d_2 \ldots d_j = CL(X) - CL(X')$. The string $b_1\ b_2 \ldots b_k$ here denotes the characters that are inserted into X, while the string $d_1\ d_2 \ldots d_j$ denotes the characters that are deleted from X.

Step-3: Apply the Step-2 and -3 of the Algorithm-7 for the string $b_1\ b_2 \ldots b_k$.

Step-4: Apply the Step-2 and -3 of the Algorithm-8 for the string $d_1\ d_2 \ldots d_j$.

4. CONCLUSION

The following are the advantages of the ECHO model:

(1) Multiple context hierarchies of a document can be provided by the ECHO model, for example, context structures and typesetting structures.

(2) Any level of contexts can be located by menas of the ECHO tree. Thus the ECHO has the ability to provide a flexible unit for retrieving the textual information.

(3) The ECHO can do subtree search to speed up the retrieval performance. In other words, users can specify a subset of the database as the search range.

(4) The ECHO tree is relatively easy to maintain. There are just O(log n) modes of an ECHO tree to be updated when a context is inserted, deleted, or modified, where n is the number of nodes in an ECHO tree.

(5) A sophisticated retrieval method such as character inversion is easy to attachg to the ECHO model.

In addition, the ECHO model is language independent. That is, it is suitable for representing the context structures of documents in any language.

The major factors of the access performance of an inversion method are AND-merge and OR-merge operations. The ARCIM can reduce both the number of AND-merge operations and the total length of posting lists invoked by AND-merge operations. Thus, the ARCIM can improve the access performance by a conventional character inversion method. Besides, the ARCIM needs less storage overhead than a conventional character inversion method.

REFERENCE

1. Faloutsos, C., "Access Methods for Text", *ACM computing Surveys*, Vol. 17, No. 1, March 1985, p.p.49-74.
2. Ozkarahan, E., *Database Machines and Database Management*, Reading, ISBN 0-13-196031-8, Ch. 12, "Document Retrieval", Prenice-Hall, Inc., New Jersey.
3. Peels, A. J. H. M., Janssen, N. J. M., and Nawijn, W., "Document Architecture and Text Formatting", *ACM Transactions on Office Information Systems*, Vol. 3, No. 4, October 1985, p.p.347-369.
4. Emrath, P. A., *Page Indexing for Textual Information Retrieval Systems*, Ph. D. Thesis, Univ. of Illinois at Urbana-Champaign, 1983.
5. Hsieh, C. C. et.al., *The Design and Implementation on the Chinese Full-text Processing System*, Chinese Publication, Computing Center, Academia Sinica, Sept. 1986.
6. ISO, Draft International Standard ISO/DIS 8879, *Information Processing–Text and Office Systems–Standard Generalized Markup Language (SGML)*, International Organization for Standardization, Geneva, Switzerland, Oct. 1985.
7. Faloutsos, C., and Christodoulakis, S., "Signature Files : An Access Method for Documents and Its Analytical Performance Evalusion", *ACM Transactions on Office Information Systems*, Vol. 2, No. 4, Oct. 1984, p.p. 267-288.
8. Salton, G. and McGill, M. J., *Introduction to Modern Information Retrieval*, McGiraw-Hill, New York, 1983.
9. Haskin, R. L., "Special-purpose Processors for Text Retrieval", *Database Engineering*, Vol. 4, No. 1, Sept. 1981, P.P. 16-29.
10. Tseng, Shih-shyeng, *The Design and Implementation of the Chinese Character Characteristics Database*, Chinese Publication, Master Thesis, National Taiwan Institute of Technology, Taipei, Taiwan, June 1982.