B) 45

AN EXPERIMENTAL MODEL OF CHINESE TEXTUAL DATABASE

Shih-Shyeng Tseng

Computing Center, Academia Sinica, Taipei, Taiwan 11529, R.O.C.

Chen-Chau Yang

Department of Electronic Engineering, National Taiwan Institute of Technology, Taipei, Taiwan 10772, R.O.C.

Ching-Chun Hsieh

Institute of Information Science, Academia Sinica, Taipei, Taiwan 11529, R.O.C.

Key Words: textual database, document model, odcument retrieval.

ABSTRACT

A textual database deals with retrieval and manipulation of documents. It allows a user to search on-line complete documents or parts of documents rather than attributes of documents. Resembling a formatted database which uses a data model as its underlying structure, a textual database has to base its development upon a document model. In this paper, a document model, called the ECHO model, is proposed. The ECHO model provides a document representation, called the ECHO structure, for expressing documents and operations on the representation that serve to express queries and manipulations on documents. It has the ability to provide multiple document structures for a document, a flexible search unit for retrieving textual information, and a subrange search on a textual database. In addition, the ECHO structure is relatively easy to maintain. An architecture of a textual database based on the ECHO model is also proposed. In order to improve the query performance, a refined character inversion method, called ARCIM, is proposed as the text-access method of the Chinese textual database. The ARCIM can retrieve texts faster than a simple inversion method and requires less space overhead.

中文全文資料庫之實驗模型

曾士熊

中央研究院計算中心

楊鍵樵 *

國立台灣工業技術學院電子工程技術系

謝清俊

中央研究院資訊科學研究所

*Correspondence addressee

processor. The query processing will be discussed in Section 6. The text retrieval subsystem plays the role of improving the retrieval performance. An ARCIM structure which is provided as the underlying mechanism of the text retrieval subsystem will be introduced in Section 5. The ECHO subsystem is an ECHO structure which provides a mechanism of storing documents and their context structures. The ECHO subsystem also provides the search operations on the ECHO structure. These operations will be mentioned in Section 4.1. The maintenance subsystem provides the necessary operations for maintaining the text retrieval subsystem and the ECHO subsystem. The maintenance operations on the ECHO structure and on the ARCIM structure will be mentioned in Sections 4.3 and 5.2, respectively.

DOCUMENT STRUCTURES

Definition 1: Document. A *document* can be defined in two ways: in terms of the author's thoughts and in terms of its constituents. According to the former, adopted from [17], a document is defined as a material reproduction of the author's thoughts and its prime objective is to transmit, communicate and store these thoughts as accurately as possible, regardless of the medium used for these thoughts. The later simply defines a document as a text associated with one or more document structures.

In Definition 1, the *text* is defined as a heterogeneous data string consisting of a sequence of text components. The *text components* may be symbols, words, phrases, or sentences in natural or artificial languages, figures, formulas, or tables. In addition, a *text element* is defined as a text forming a meaningful unit of a document, which may be the whole document or a part of the document, e.g., a paragraph, a section or a chapter. A text element which does not contain any subordinate text elements is called a *basic text element.*

When an author writes a document, the text of the document has to be organized into a logical structure in order to reflect the conceptual skeleton of the author's thoughts. The logical structure is determined by the author and is unique and unchangeable. In practice, the author

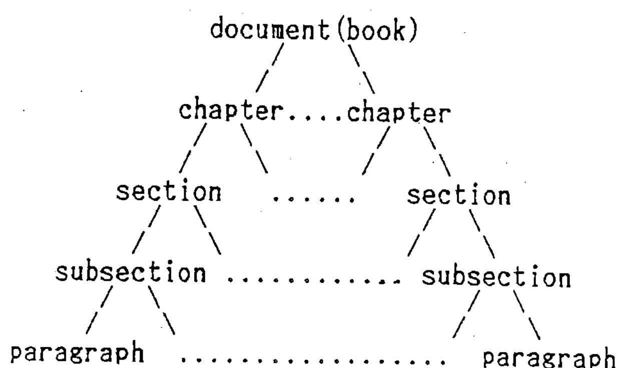


Fig. 2. An example of logical structure.

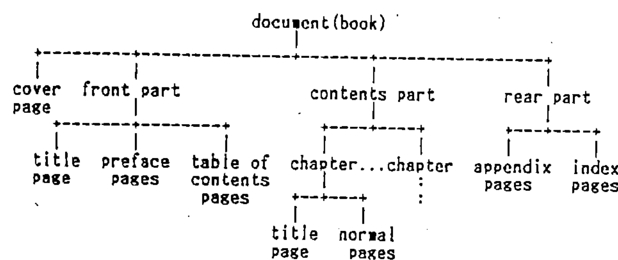


Fig. 3. An example of layout structure.

first organizes a number of text components to form a basic text element, e.g., a paragraph. Then he organizes a number of basic text elements to form a larger one, e.g., a section, and so on, until the document is formed. These text elements form a hierarchical structure in which each text element, except for a basic text element, is a composite of subordinate text elements, as shown in Fig. 2. The hierarchy of text elements of a document is referred to as the *logical structure* of the document. The logical structure is always presented in a human readable form, namely layout structure. For a document, the *layout structure* reflects the formatting of the text and the logical structure of it in a representation medium such as paper or screen. The layout structure, similar to the logical structure which is the hierarchy of text elements, is the hierarchy of layout elements. A *layout element* may be a page, a set of pages, or a subordinate element of a page, e.g., a line, a block or a frame. In general, 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, etc., as shown in Fig. 3.

Definition 2: Context. Given a text, a *context* is defined as follows:

(1) The whole text is a context.

(2) If a context is partitioned into a series of nonoverlapped but concatenated subtexts, then each subtext is a context.

It is clear that Definition 2 is recursive. In Definition 2, the whole text specified in (1) is referred to as the *root context* or the level 1 context. The partitioning operation specified in (2) is referred to as *hierarchical partitioning*. That is, a level i context may be partitioned into a series of level i+1 contexts. A context Y is said to be contained in a context X, or reversely, the context X is said to contain the context Y, if and only if the context Y is directly or indirectly partitioned from the context X. A *leaf context* is defined as one that doesn't contain any lower-level context. Because the number of text components of a text is finite, it is trivial that the number of levels from the root context to any leaf context is also finite.

Theorem 1: Each context of a given text forms a tree structure, called the *context structure.*

Proof: Adopted from the definition of a tree proposed in [15], the proof is given as follows. The given text is the level 1 context. Depending upon whether the level i con-

That is, a set of context delimiters has to be inserted into the text in order to identify each context. Every context is then surrounded by a pair of beginning and ending delimiters. A text scanner can then be used to find the beginnings and the ends of contexts. A higher-level context containing the current context or a lower-level context contained in the current context can also be searched by scanning the text forward and backward. The advantages of an implicit representation are: the data structure is simple, it requires only space overhead for context delimiters, and it is relatively easy to maintain. However, its disadvantage is that one has to retrieve contexts by means of full text scanning. The time required for scanning a context from the whole database is O(L), where L denotes the length of the whole text. Even though a refined string pattern matching method can improve the search speed [1,2,14], fulll text scanning still consumes too much time retrieving contexts. Hence it is not reasonable to develop a large textual database using the implicit representation. Recall from Section 2 that a context structure of a given text can be explicitly represented by a context tree with each node of the tree denoting a context of the context structure. Hence instead of inserting a set of context delimiters into the text, an *explicit representation* uses an explicit context tree to reflect each context structure of the text.

Definition 3: Explicit context tree. An *explicit context tree* is a context tree which represents a context structure of a given text. In the explicit context tree, each node uniquely denotes a context of the context structure and carries a local name and a vector consisting of a pair of pointers (BP,EP). The pointers BP and EP point to the beginning and the end positions of the context denoted by the node, respectively.

In a context tree H, it is clear that for each node X, there exists a unique search path from the root to node X. The *path name* of node X is then defined as the list consisting of the local names of the nodes along the search path of node X. The number of nodes (or local names) in a path name is referred to as the *length* of the path name.

Defintion 4: Explicit representation. The *explicit representation* of the context structures of a text T is defined as a finite set, $(T,H_1, \ldots, H_k)$, consisting of the text T and k explicit context trees.

A textual database is basically an instance of the explicit representation, as shown in Fig. 4. In Fig. 4, the

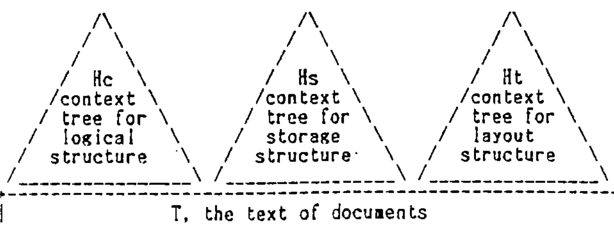


Fig. 4. An example of explicit representation.

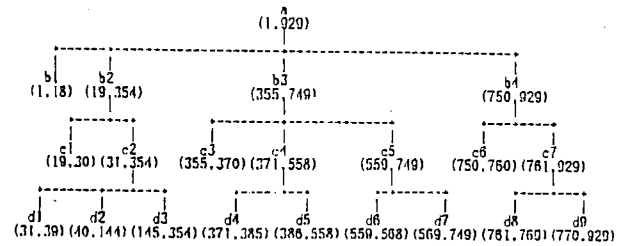


Fig. 5. An example of context tree.

textual database is simply represented as (T,Hc,Ht,Hs), where T denotes the whole text of all documents of the database, and Hc, Ht and Hs respectively represent the logical structure, the layout structure, and the storage structure of the text T. Both the logical and the layout structures have been mentioned, in Section 2. The storage structure represents how the text T is stored in the secondary storage of a particular computer system. In addition, an example of the context tree is shown in Fig. 5.

Definition 5: Context-id. Given a text T and a context tree H denoting a context structure of the text T, assume that each context X of the context structure is denoted by a node X' of the tree H. The *context-id* of the context X is defined as the path name of the node X'.

In an explicit context structure $(T,H_1, \ldots H_k)$, the beginning and the end positions of any context can be easily searched if its context-id is given. A higher-level context containing the current context or a lower-level context contained in the current context is also easily searched via the links on the context trees. The time required to search any context by means of a context tree H_i, $1 \le i \le k$, is O(n), where n denotes the height of the tree H_i. The number n is roughly O(log N) and is much smaller than L, where N denotes the number of nodes of the tree H_i and L denotes the length of the text T. Hence from the viewpoint of search speed, the explicit representation is much better than the implicit one.

An explicit context tree has an additional property as follows:

Property 5: For each node X of an explicit context tree, if the length of its path name is m, then $m \ge 1$ and the path name of the node X can be formally presented as a list $N_1 \ldots N_m$. It is clear that the node X has $m-1$ ancestors and each of them is denoted by a proper prefix of the list $N_1 \ldots N_m$, where a proper prefix of the list $N_1 \ldots N_m$ is defined as a sublist of the form $N_1 \ldots N_j$, $1 \le j < m$. It is obvious that this property also holds for the context-id of the context denoted by the node X.

The advantages of the explicit representation will be mentioned in Section 7. Its disadvantages are: it requires space overhead of storing the context trees, and both of the text and the context trees have to be updated in order to maintain a context. In a context structure denoted by a context tree H, the *maintenance cost* of any context X is defined as the number of the nodes of the tree H whose vector (BP,EP) must be updated if the con-

Theorem 3: In an ECHO tree, the D-value and the L-value of any node X, i.e., D(X) and L(X), except for D(root), can be computed as follows.

(1) If the node X is a leaf node, then L(X) is actually the length of the leaf context X; otherwise, $L(X)=L(Y_1)+\ldots+L(Y_k)$, where $Y_1, \ldots, Y_k$ are all children of the node X.

(2) If the node Y is the leftmost child of its parent, then $D(Y)=0$; otherwise, $D(Y)=D(Z)+L(Z)$, where Z is the nearest left sibling of the node Y, i.e., the node Z has the same parent of the node Y and is in the position immediately preceding the node Y.

Proof: From Definition 6 and Property 5, it is easy to show part (1). Part (2) is then proved as follows. For each non-leaf node X, assume that its children are nodes $Y_1, \ldots, Y_k$, from left to right. From Definition 2, it is obvious that the contexts Y_1 and X have the same beginning pposition. Hence $D(Y_1)=0$. In addition, it is trivial that

$$D(Y_i) = D(Y_{i-1}) + L(Y_{i-1}),\ 1<i\leq k, \qquad (5)$$

where node Y_{i-1} is the nearest left sibling of node Y_i. From the above, this theorem has been proved. ■

Theorem 4: $BP(N_1\ldots N_m)$ and $EP(N_1\ldots N_m)$, i.e., the beginning and the end positions of the context denoted by a node $N_1\ldots N_m$ of an ECHO tree, can be computed by

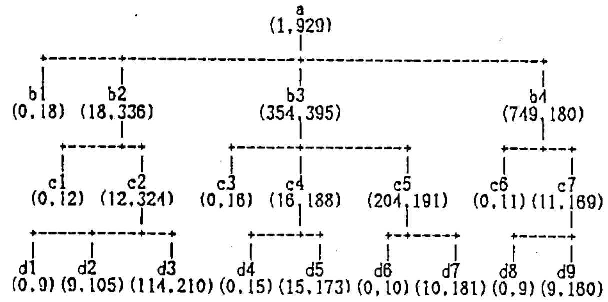
$$BP(N_1\ldots N_m)=\Sigma_{1\leq i<m} D(N_1\ldots N_i), \text{ and} \qquad (6)$$
$$EP(N_1\ldots N_m)=BP(N_1\ldots N_m)+L(N_1\ldots N_m)-1 \qquad (7)$$

The proof of Theorem 4 is trivial and therefore omitted. ■

From Definitions 3 and 6, it is clear that an explicit context tree H and an ECHO tree E are equivalent if they represent the same context structure. That is, they are the same except that each vector (BP,EP) of the tree H is replaced by a vector (D,L) of the tree E. In addition, the trees H and E have the same number of nodes. The explicit context tree H can be transferred to the equivalent ECHO tree E by the following method. First, let the D(root) of the tree E be the BP(root) of the tree H. Second, for each leaf node X of the tree H, the L(X) is computed by

$$L(X) = EP(X)-BP(X)+1 \qquad (8)$$

Last, compute all the D- and L-values of the tree E by applying Theorem 3. For example, the ECHO tree which is equivalent to the context tree shown in Fig. 5 is shown in Fig. 6. Reversely, the ECHO tree E can be transferred to the equivalent explicit context tree H by applying Theorem 4.



Definition 7: ECHO structure. The *ECHO structure* of the context structures of a text T is defined as a finite set consisting of the text T and k ECHO trees $E_1, \ldots, E_k$, denoted $(T,E_1, \ldots, E_k)$. An ECHO structure (or sub-ECHO structure) consisting of a text and only one ECHO tree is called a *singular ECHO structure*.

From the above, it is clear that an explicit representation $(T,H_1, \ldots, H_k)$ can be transferred to an equivalent ECHO structure $(T,E_1, \ldots, E_k)$ by replacing each context tree H_i by an equivalent ECHO tree E_i, $1\leq i\leq k$. For example, Fig. 4 can also be used to depict a textual database based upon the ECHO structure if the context trees Hc,Ht and Hs are replaced by the equivalent ECHO trees Ec, Et and Es, respectively.

1. Search operations on ECHO structures

A *search operation* on an ECHO structure $(T,E_1,\ldots,E_k)$ is an operation that retrieves information from an ECHO structure. The information includes context-ids, beginning and end positions of contexts and texts of contexts. A search operation is always constrained to search one type of informaiton from a singular ECHO structure (T,E_i), $1\leq i\leq k$. The search operations are *get-ptrs, get-text, get-id, get-leafts* and *get-ids*.

Definition 8: *get-ptrs* and *get-text*. Assume that an ECHO structure $(T,E_1,\ldots E_k)$ and a context-id $N_1\ldots N_m$ are given.

(1) The search operation *get-ptrs* computes the beginning and the end positions of the context $N_1\ldots N_m$, i.e., $BP(N_1\ldots N_m)$ and $EP(N_1\ldots N_m)$.

(2) The search operation *get-text* reads the context $N_1\ldots N_m$, i.e., the subtext between the positions $BP(N_1\ldots N_m)$ and $EP(N_1\ldots N_m)$, from the text T.

The search operation *get-ptrs* is easily performed by applying Theorem 4. Let us take node d_7 in Fig. 6 as an example. The context-id of context d_7 is $ab_3\ c_5\ d_7$. From Theorem 4, we have

$$BP(ab_3c_5d_7)=D(a)+D(b_3)+D(c_5)+D(d_7)$$
$$=1+354+204+10 = 569, \text{ and}$$
$$EP(ab_3c_5d_7)=BP(ab_3c_5d_7)+L(d_7)-1$$
$$=569+181-1=749.$$

rect sequence. Assume that the nth ECHO tree of each given ECHO structure, denoted E_{ni}, $1 \le n \le k, 1 \le i \le j$, represents the n-th kind of context structure. And third, only the ECHO trees that represent the same kind of context structure are able to be combined.

Algorithm 3: Perform the ECHO-combining operation.
Input: j ECHO structures $(T_1, E_{11}, \ldots, E_{k1}), \ldots, (T_j, E_{1j}, \ldots, E_{kj})$.
Output: A combined ECHO structure $(T, E_1, \ldots, E_k)$.
Procedure:

T: = null;
for i: = 1 to j do append (T,T_i);
for n: = 1 to k do
  begin
    create a new node E_n;
    for i: = 1 to j
      do link the node E_{ni} to the node E_n such that E_{ni} becomes the rightmost child of E_n;
    $D(E_n)$: = *the beginning position of T;*
    $D(E_{n1}) := 0;$
    for i: = 2 to j do $D(E_{ni}) := D(E_{n,i-1}) + L(E_{n,i-1});$
    $L(E_n) := D(E_{nj}) + L(E_{nj});$
  end;

3. Basic updating operations on ECHO structures

A *basic updating operation* is an operation that maintains a singular ECHO structure. That is, for a singular ECHO structure (T,E), a basic updating operation can insert a new context into it, delete an old context from it, or modify an existing leaf context of it. These operations are named *echoinsert*, *echodelete* and *leafmodify*, respectively. In general, an updating operation on an ECHO structure $(T, E_1, \ldots, E_k)$ has to invoke a series of basic operations and some optional search operations.

Definition 12: *echoinsert*. Assume that an ECHO structure (T,E), an object of insertion (Tx,Ex), and a parameter either $N_1 \ldots N_m$:l or $N_1 \ldots N_m$:r are given. The updating operation *echoinsert* performs either of the following operations, depending upon which parameter is given.

(1) If a parameter $N_1 \ldots N_m$:l is given, then the text Tx is inserted into the text T in the position immediately preceding the context $N_1 \ldots N_m$ and the node Ex is linked to the node $N_1 \ldots N_{m-1}$ as the nearest left sibling of the node $N_1 \ldots N_m$.
(2) If a parameter $N_1 \ldots N_m$:r is given, then the text Tx is inserted into the text T in the position immediately following the context $N_1 \ldots N_m$ and the node Ex is linked to the node $N_1 \ldots N_{m-1}$ as the nearest right sibling of the node $N_1 \ldots N_m$.

It is obvious that after the text Tx is inserted into the text T, the following occur. First, the BP(Ex) is changed, i.e., D(ex) is changed. the new D(Ex) has to be computed by applying Theorem 3. Second, each context denoted by a right sibling W of the node Ex is moved right by L(Ex) from its original position, i.e., D(W) must be increased by L(Ex). Third, the length of each context U containing the text Tx is increased by L(Ex), i.e., L(U) has to be increased by L(Ex). And fourth, for each ancestor U of the node Ex, each context denoted by a right sibiling W of the node U is also moved right by L(Ex) from its original position, i.e., D(W) must be increased by L(Ex).

Definition 13: *echodelete*. Given an ECHO structure (T,E) and a context-id $N_1 \ldots N_m$ of the ECHO structure, the updating operation *echodelete* removes the context $N_1 \ldots N_m$ from the text T and removes the subtree $N_1 \ldots N_m$ from the ECHO tree E.

It is clear that after the context $N_1 \ldots N_m$ is removed, the following occur. First, each context denoted by a right sibling W of the node $N_1 \ldots N_m$ is moved left by $L(N_1 \ldots N_m)$ from its original position, i.e., D(W) has to be reduced by $L(N_1 \ldots N_m)$. Second, the length of each context U containing the context $N_1 \ldots N_m$ is reduced by $L(N_1 \ldots N_m)$, i.e., L(U) must be reduced by $L(N_1 \ldots N_m)$. And third, for each ancestor U of the node $N_1 \ldots N_m$, each context denoted by a right sibling W of the node U is also moved left by $L(N_1 \ldots N_m)$ from its original position, i.e., D(W) has to be reduced by $L(N_1 \ldots N_m)$.

Definition 14: *leafmodify*. Given a leaf context X of an ECHO structure (T,E) and the modified version of the context X, say context X', the updating operation *leafmodify* replaces the context X by the context X'. In addition, the operation *leafmodify* updates the relevant D-values and L-values of the ECHO tree E if $L(X') \neq L(X)$.

The modification of a context is actually realized by modifying some leaf contexts of the context. The modification of the leaf context X may cause a change in L(X), i.e., $L(X') \neq L(X)$. When L(X) is changed, adopted from the above, the following occur. First, for each right sibling W of either the node X or any ancestor of the node X, the D(W) has to be changed to $D(W) - L(X) + L(X')$. And second, for each ancestor U of the node X, the L(U) must to be changed to $L(U) - L(X) + L(X')$.

Theorem 5: For a nearly balanced singular ECHO structure (T,E), the maintenance cost of a context is O(log N), where N is the number of nodes of the ECHO tree E.
Proof: We first consider the cases of insertion and deletion. From the discussions following Definitions 12 and 13, the maintenance cost MC_1 of a context $N_1 \ldots N_m$ is given by

$$MC_1 = m + \Sigma_{2 \le i \le m} K_i \text{ for } m \le n, \qquad (10)$$

where K_i is the number of the right siblings of each node $N_1 \ldots N_i$ and n is the height of the ECHO tree E. Because the ECHO tree E is nearly balanced, it is reasonable to assume that the average number of children of each nonleaf node of E is K and n = log N. Thus we have

A'∩B'.

Difference of context-id sets. The A' − B' is performed as follows. First, for each context-id X_i, $1 \leq i \leq m$, if there does not exist any context-id Y_j, $1 \leq j \leq n$, such that $X_i = Y_j$ or Y_j is a proper prefix of X_i or X_i is a proper prefix of Y_j, then X_i appears in A' − B'. Second, for each context-id X_i, $1 \leq i \leq m$, if there exists a context-id Y_j, $1 \leq j \leq n$, such that X_i is a proper prefix of Y_j, then each context-id of the set difference $(X_{i1}, \ldots, X_{ih}) - (Y_j)$ appears in A' − B'. Each X_{ik}, $1 \leq k \leq h$, is a descendant of X_i, which is either a leaf context-id with $\text{length}(X_{ik}) \leq \text{length}(Y_j)$ or a non-leaf context-id with $\text{length}(X_{ik}) = \text{length}(Y_j)$. And third, nothing else appears in A' − B'.

The set operations of context sets (or context-id sets) provide an important basis for processing the query of textual databases. In general, a query consists of some predicates which are combined by Boolean operators. Given two context sets A and B as an illustration, suppose each member of the set A satisfies a predicate P and each member of the set B satisfies a pedicate Q. It is clear that each member of A∪B satisfies P or Q or both, each member of A∩B satisfies both P and Q, and each member of A − B satisfies P but not Q. Hence the context sets A∪B, A∩B and A − B are the results of the Boolean expression "P OR Q", "P AND Q" and "P AND NOT Q", respectively.

THE ARCIM

A *text-access method* is the method of identifying, retrieving, and/or ranking contexts in a collection of contexts, that might be relevant to a given query. The text-access methods, as discussed in [6,16], can be classified into four categories: full text scanning, inversion of terms, surrogates of contexts and clustering. The basic idea of an inversion method is that a context is considered as a list of terms, which describe the contents of the context for retrieval purposes. In an English text, a term is actually a word. A fast retrieval can be achieved if one inverts terms. An inversion method uses an index structure in which each entry consists of a term along with a posting list. The *posting list* is a list of pointers each of which points to a context containing the term. The disadvantages of the inversion method are: the storage overhead (50-300% of the size of the text files [8]); the cost of updating and reorganizing the index, if the environment is dynamic; and the cost of merging the posting lists, if they are too long or there are too many of them. In contrast, the advantages are that it is relatively easy to implement and is fast. Hence the word inversion has been adopted in most commercial systems, such as BRS, DIALOG, MEDIARS, ORBIT and STAIRS [18].

The word inversion is not applicable to Chinese text. The reasons are as follows. A Chinese sentence does not contain any natural delimiters, such as blanks in an English sentence, to separate Chinese words. While some automatic methods of identifying the words in a Chinese sentence have been proposed [7,11,19], it is still impossible for words in a Chinese sentence to be completely identified by any computer technology. Hence a character inversion isntead of the word inversion is used for retrieving Chinese text. A refined character inversion method (abbr. *ARCIM*) will be proposed in this paper.

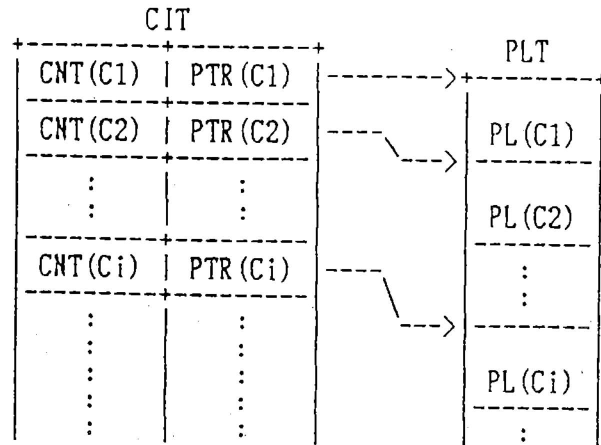


Fig. 7. The index structure of ARCIM.

The ARCIM uses an ARCIM structure as its index structure. The *ARCIM structure*, denoted (CIT,PLT), consists of a character index table CIT and a posting list table PLT, as shown in Fig. 7. An entry of the PLT, denoted *PL(C_i)*, is a posting list consisting of a variable number of ordered leaf context-ids. Each context-id X of the PL(C_i) stands for the condition that the context X contains the Chinese character C_i. In the CIT, each entry consists of a count CNT(C_i) and a pointer PTR(C_i). The *CNT(C_i)* denotes the number of leaf context-ids of the PL(C_i) and the *PTR(C_i)* points to the starting location of the PL(C_i). Because the coding space of Chinese characters of a computer system is fixed [21], the CIT can be organized with a constant size and may be permanently located in the main memory. If a Chinese character C_i is given, then the CNT(C_i) and the PTR(c_i) can be easily searched from the CIT by using a specific hash function. An example of the hash function has been introduced in [20]. If there exists a Chinese character C_j which does not appear in any context, then the CNT(C_j) and the PTR(C_j) must be set to zero and nil, respectively. And the PL(C_j) does not appear in the PLT. There are three types of operations provided for the ARCIM structure: the search operations, the creation operations and the maintenance operations. The search operations are discussed in Section 6 rather than in this section.

1. The creation of ARCIM structures

Definition 18: ARCIM-creating operation. Given a singular ECHO structure (T,E), the *ARCIM-creating operation* creates an ARCIM structure (CIT,PLT) from

bol — denotes a set difference operation. It is clear that the characters $B_1 \ldots B_n$ are added to the context X and the characters $D_1 \ldots D_n$ are removed from the context X.
Step 3: *Apply steps 2 and 3 of Algorithm 5 to the string $B_1 \ldots B_n$. And apply steps 2 and 3 of Algorithm 6 to the string $D_1 \ldots D_m$.*

QUERY PROCESSING

The steps involved when a user queries information from a textual database can be roughly summarized as follows. He first selects some terms (keywords) which stand for his topic of interest. These terms have to be conjoined by some operators, as mentioned in [9], to form a predicate. Then he queries the database to find the contexts that satisfy the predicate. In our system, a query expression has the following form:

FIND *context-clause*
CONTAIN *search-clause*
scope-clause; (14)

In a query expression, the *context-clause* specifies which type of context is retrieved. A context clause has the following form:

context-clause
:: = LEA CONTEXTS | CONTEXTS OF LENGTH k (15)

The symbol :: = means "is defined as" and the vertical bar | stands for "or". The phrase "LEAF CONTEXTS" denotes that each retrieved context must be a leaf context. The phrase "CONTEXTS OF LENGTH k" specifies that each retrieved context is either a leaf context having a context-id length no greater than k or a non-leaf context having a context-id length of k.

The *search-clause* specifies a condition the retrieved contexts must satisfy. The basic form of a search clause is shown below.

search-clause
:: = *search-phrase* {OR *search-phrase*} (16)
search-phrase
:: = *term* {AND [NOT] *term*} (17)
term
:: = *string* | *wild-card-term* | *ordered-term* (18)

The braces {...} denote a repetition and the brackets [...] denote an optional item. The operations for a search clause will be discussed later.

The *scope-clause* specifies the search space a query invokes. A scope-clause has the following form:

scope-clause
:: = UNDER *context-id* |
FROM *context-id*$_1$ TO *context-id*$_2$ |
FROM SETS *context-name* {,*context-set-name*} (19)

An ECHO structure may have more than one ECHO tree, as shown in Fig. 4. In this case, the phrase "UNDER *context-id*" is necessary in order to specify a singular ECHO structure in which the retrieved contexts are contained. For example, assume that the following query expression is given.

FIND LEAF CONTEXTS
CONTAIN "textual data?base" OR
"information retriev*"
UNDER Ec; (20)

A paragraph, i.e., a leaf context of the logical structure Ec, containing "textual database", "textual data base", "information retrieve", "information retrieval" or "inforamtion retrieving" will be retrieved. The phrase "FROM *context-id*$_1$ TO *context-id*$_2$" specifies a subset of contexts of a singular ECHO structure, from the context denoted by context-id$_1$ to the context denoted by context-id$_2$, as the search space. A constraint of this phrase is that the context denoted by context-id$_1$ must precede the context denoted by context-id$_2$. Sometimes, a user needs to search the contexts from the results of previous queries or from a specific context set such as the titles of the documents. The FROM SETS phrase is provided for these purposes. The constraint of the FROM SETS phrase is that all members of the given sets must belong to the same ECHO structure.

The query processor invokes three phases for evaluating a query expression, namely consistence check, search process and post process. When a query expression is given, the query processor first checks whether the scope clause is valid, i.e., the *consistence check*. There are three cases that must be cfonsidered.
Case 1: If a phrase "UNDER $N_1 \ldots N_k$" is given, the query processor applies the operation *get-ptrs* to confirm that $N_1 \ldots N_k$ is a valid context-id.
Case 2: If a phrase "FROM $A_1 \ldots A_j$ TO $B_1 \ldots B_k$" is given, the query processor confirms first that the context-ids $A_1 \ldots A_j$ and $B_1 \ldots B_k$ are of the same ECHO tree, i.e., $A_1 = B_1$. Then the operation *get-ptrs* is applied to confirm that $A_1 \ldots A_j$ and $B_1 \ldots B_k$ are valid context-ids and $A_1 \ldots A_j$ precedes $B_1 \ldots B_k$. The context $A_1 \ldots A_j$ is said to precede the context $B_1 \ldots B_k$ if $EP(A_1 \ldots A_j) < BP(B_1 \ldots B_k)$.
Case 3: Assume that a phrase "FROM SETS $S_1, \ldots, S_k$" is given. The query processor OR-merges these context sets to form a larger one and at the same time it checks that each context-id of the sets has the same root name. If the consistence check fails, then the query expression will be rejected. The search process and the post process are discussed in Section 6.1 and Section 6.2, respectively.

1. The search process

After the query expression is confirmed by the con-

specified by the wild-card terms and ordered terms, the context-id X must be deleted.

The phrase-level posting list PPL_i' which satisfies the original search phrase SP_i is then obtained.

Adjustment of context-ids. Two cases have to be considered. First, if the context clause "LEAF CONTEXTS" is given, the adjustment operation is not necessary. Second, assume that the context clause "CONTEXTS OF LENGTH k" is given. In the case, if a leaf context-id contained in the PPL_i' has a length equal to or less than the number k, it does not have to be adjusted. If a leaf context-id contained in the PPL_i' has a length greater than the number k, say $N_1 \ldots N_k N_{k+1} \ldots N_m$, then its postfix $N_{k+1} \ldots N_m$ must be truncated. In addition to the second case, if the scope clause "FROM SETS S_1, ..., S_k" is given, then each context-id of the resultant set of OR-merging sets S_1, also is adjusted. The adjusted resultant set is called set S.

Merge operation. If the scope clause is not of the form "FROM SETS S_1, ..., S_k", then the merge operation is simply OR-merge for all the phrase-level posting lists and forms the clause-level posting list. From Section 6.1, it is also easy to show that the OR-merges invoke the smallest total number of context-ids if a shortest-first strategy is applied. If the scope clause "FROM SETS S_1, ..., S_k" is given, the clause-level posting list and the S have to be AND-merged in order to obtain the final result.

CONCLUSIONS

The following are the advantages of the ECHO structure. First, the ECHO structure is applicable to a Chinese textual database or an English textual database, i.e., it is language-independent. Second, multiple context structures of documents can be represented by the ECHO structure, e.g., the logical structure and the layout structure. Third, any context of a document can be searched by means of the ECHO strucutre. Hence the ECHO structure has the ability to provide a flexible search unit of a query. Fourth, the ECHO strucutre can provide a subrange search to speed up the retrieval operation. That is, a user can specify a subset of the database as the search space. Fifth, the ECHO structure is relatively easy to maintain. Only O(log N) nodes of an ECHO tree have to be updated if a context is inserted, deleted or modified, where N denotes the number of nodes of the ECHO tree. And last, it is easy to attach a sophisticated retrieval method such as the ARCIM to the ECHO structure. These advantages, except for the fifth one, also hold for the explicit representation. In addition, for an inversion method, the major factors of the search performance are the AND-merges and OR-merges. Referred to in Section 6, the ARCIM can reduce the number of AND-merges, and the total length of posting lists invoked by the AND-merges and by the OR-merges. Hence the ARCIM improves the search performance.

Adopted from the definition of the text mentioned in Section 2, a text component can be roughly classified into either character type or non-character type. A text component of the character type is the one that consists of only characters, such as a symbol, a word, a phrase, and sometimes a formula or table. The figures such as pictures, diagrams, drawings, paintings, or images, by contrast, are text components of the non-character type. Text components of the non-character type are excluded from our system at present. In addition, the annexed attributes of contexts such as bibliographies of documents and the semantic information of contexts are also excluded. Representations and operations of these will be studied in the future.

NOMENCLATURE

ARCIM	a refined character inversion method
BP(X)	the beginning position (pointer) of the context X
CIDLIST(X)	the ordered set of CIDPAIRs of the context X
CIDPAIR(a,X)	a pair of Chinese character a and the context-id X, in which C is contained in the context C
CIT	character inversion table
CLIST(X)	the order set of different Chinese characters of the context X
CNT(a)	the count of the contexts containing the Chinese character a
D(X)	the distance between the beginning position of the context X and that of the context immediately containing X
ECHO	explicit context-hierarchical organization
EP(X)	the end position (pointer) of the context X
L(X)	the length of the context X
PL(a)	the order list of context-ids (i.e., posting list) denoting the contexts containing the Chinese character a
PLT	posting lists table
PPL	a phrase-level posting list
PTR(a)	the pointer that points to the PL(a)

REFERENCES

1. Aho, A.V. and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, Vol. 18, No. 6, pp. 333-340 (1975).
2. Boyer, R.S. and J.S. Moore, "A Fast String Searching Algorithm," *Commun. ACM*, Vol. 20, No. 10, pp. 762-772 (1977).
3. Coombs, J.H., A.H. Renear and S.J. Derose, "Markup Systems and the Future of Scholarly Text Processing," *Commun. ACM*, Vol. 30, No. 11, pp. 933-947 (1987).