TR-91-015

AN EXPERIMENTAL MODEL OF CHINESE TEXTUAL DATABASE

C.

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

摘 要

全文資料庫的主要功能在檢索與管理文獻。它允許使用者線上查詢完整的或部分的文獻，而不只是查詢文獻的屬性資料。如同表格式資料庫以資料模式爲其內部結構，全文資料庫的開發亦需以文獻模式爲基礎。本論文將提出一套文獻模式（稱爲 ECHO 模式）和一個以 ECHO 模式爲基礎的全文資料庫架構。ECHO 模式包括文獻表達法（即 ECHO 結構）和以此表達法爲基礎的運算，這些運算可用以描述文獻的查詢和管理工作。ECHO 模式具有表達多重文獻結構、提供彈性檢索單位以及查詢部分全文資料庫的能力。此外，ECHO 結構也比較容易維護。爲了增進查詢的效率，本論文還提出一套改進的字符反列法（稱爲 ARCIM）做爲中文全文資料庫的檢索機制。ARCIM 比起普通的字符反列法來，檢索速度較快且儲存空間較省。

INTRODUCTION

Formatted databases and textual databases are two important categories of databases. A *formatted database* deals with retrieval and manipulation of formatted records while a *textual database* with documents. The main differences between the formatted records and the documents are the data structure, the query language, and operational requirements such as update frequency and size of database [6]. A formatted database uses a data model as its underlying structure. The three most important data models are relational, hierarchical and network [24]. They use similar ways to express the data. That is, the data of a formatted database are grouped into several data sets in which each one is associated with a record format consisting of a number of attributes. Adopted from [4], a data set has four important properties. First, there are no duplicate data in the set. Second, data are unordered in the set. Third, attributes are unordered in the record format of the set. Fourth, all simple attribute values are atomic. It will be mentioned in Section 2 that the properties of documents conflict with these properties. Hence data models are not suitable for developing textual databases.

A textual database allows a user to search online complete documents or parts of documents rather than attributes of documents as in a bibliographic system. The documents may be articles, newspaper stories, legal documents, dictionaries, books, etc. The parts of documents may be entries, paragraphs, sections, etc. In some publications dealing with text-access methods, e.g., [6,16], a document is considered as non-structured data consisting of an arbitrary number of words. This argument is not valid. Documents are also structured data though the structure is not similar to that of formatted records. Actually, a document is a text associated with one or more document structures [17,23]. 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 general, a *document model* consists of two components: a document representation of expressing documents and operations based upon the representation that serve to express queries and manipulations of documents. The *document representation* is defined as a data structure which represents the constituents of documents, including texts and document structures. From the viewpoint of formalization, a document structure can be considered as a context structure [23]. A document model based on this view, namely the ECHO model, will be introduced in Section 4.

An architecture of a textual database based upon the ECHO model is shown in Fig. 1. It consists of five modules: a user interface, a query processor, a text retrieval subsystem, an ECHO subsystem and a maintenance subsystem. The major function of the user interface is to accept and to recognize requests from users. A user request may be a query expression or a maintenance command. When a query expression is given, it must be passed to the query processor. If a maintenance command is given, then it has to be passed to the maintenance subsystem. By means of the text retrieval subsystem and the ECHO subsystem, the contexts that satisfy the query expression can be found by the query

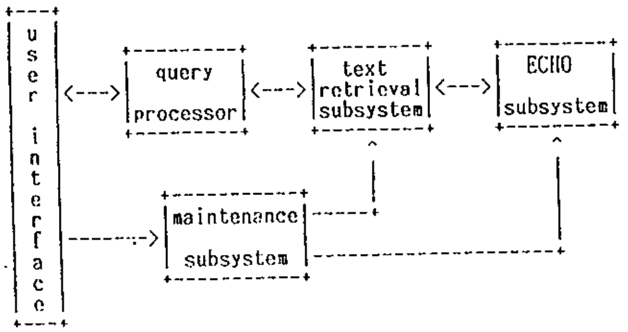


Fig. 1. A proposed architecture for textual database.

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

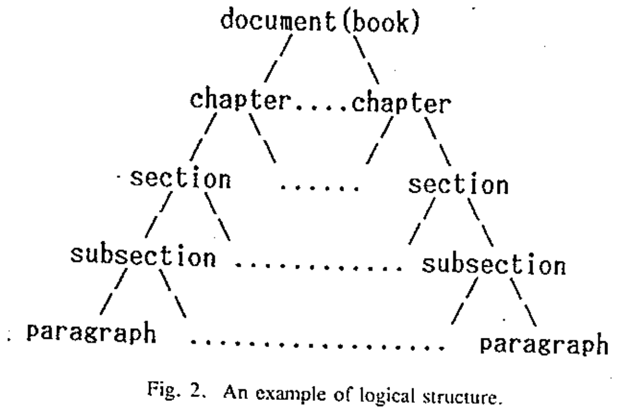

document(book)
chapter....chapter
section ...... section
subsection .... subsection
paragraph .... paragraph

Fig. 2. An example of logical structure.

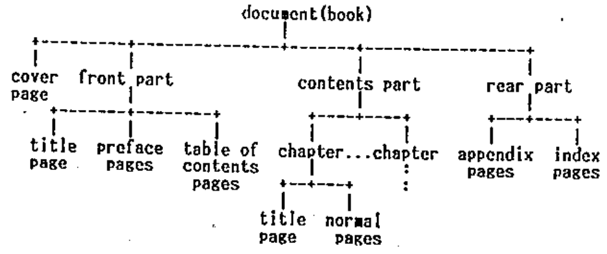


Fig. 3. An example of layout structure.

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 context, i=1, is partitioned into a series of level i+1 contexts or not, there are two cases to be discussed.

Case 1: If the level i context is not partitioned, then it forms a special tree structure having only the root.

Case 2: The level i context is partitioned into a series of level i+1 contexts. The necessary condition for the level i context to form a tree structure is that each of the level i+1 contexts contained in the level i context forms a tree structure. In other words, the level i context is the root of the tree structure and each of the level i+1 contexts forms a subtree of the root. Thus the problem is to show that each of these level i+1 contexts also forms a tree structure. Each level i+1 context which is not partitioned, similar to the case 1, forms a special tree structure with only a root. For each level i+1 context which is partitioned into a series of level i+2 contexts, let i+1 be i and recursively apply case 2 until a leaf context is reached. It is clear that each level i context forms a tree structure. From both cases, the theorem is proved. ■

It is obvious that each text element of a document is hierarchically partitioned from the whole text of the document. From Definition 2 and Theorem 1, it is easy to show that each text element of the document is a context and the logical structure of the document forms a context structure. On the other hand, if a page is seen as a basic layout element, then the layout structure of the document is also a context structure. Thus from the viewpoint of formalization, both the logical structure and the layout structure are context structures. The differences among text elements, layout elements and contexts are: the text elements are relevant to the author's thoughts, the layout elements are relevant to the formatting of a document in a representation medium, while the contexts are irrelevant to both. Hence a text element or a layout element has to be a context but a context may or may not be a text element or a layout element.

Because the context structure of a document is really a tree structure, it can be explicitly represented by a tree, called a *context tree.* In a context tree, each node denotes a context of the document. From Definition 2, Theorem 1, the relevant definitions of a tree mentioned in [15], and the nature of a document, it is easy to find that a context tree has the following properties:

Property 1: Given any two nodes X and Y of the context tree, node Y is a *descendant* of node X, or reversely node X is an *ancestor* of node Y, if and only if the context denoted by node Y is contained in the context denoted by node X.

Property 2: For any node of the context tree, the number of its children cannot be prespecified by a constant. That is, any context of a document contains a non-predictable number of subordinate contexts.

Property 3: For any node of the context tree, the order of its children is unchangeable. That is, the order of the contexts of a document cannot be changed at all.

Property 4: If nodes $Y_1, \ldots, Y_m$ are all children of node X in the context tree, then the length of the context denoted by node X is the sum of the lengths of the contexts denoted by nodes $Y_1, \ldots, Y_m$. In addition, for any context, its length cannot be prespecified by a constant.

By comparing these properties to the properties of formatted records, it is easy to conclude that the data model is not suitable to develop the textual database. Hence a textual database has to base its development upon a document model rather than a data model. There are some conventions used in this paper. First, the context denoted by node X of the context tree is simply called *context X.* Second, a context tree or a sub-context tree is named by its root, i.e., *context tree X* means that it has the root X. Third, the *height* (or *depth*) of a context tree is referred to as the maximal number of levels of the tree from the root to leaf nodes. Because a context structure is denoted by a context tree, the height (or depth) of the context tree is also referred as the height (or depth) of the context structure.

From Definition 1 and the above arguments, a document can be represented in terms of its text and its context structures. Two documents are said to be of the same class if they have similar length of texts and similar depth of context structures. A textual database is always composed of documents of the same class. Consider two different classes of docments: short-shallow documents and long-deep documents. A short-shallow document is short in the length of its text and shallow in the depth of its context structure, such as an abstract or a letter. In a textual database consisting of short-shallow documents, e.g., the UMI's Dissertation Abstracts Ondisc system, an entire document always serves as the search unit when retrieving information from it. In contrast, a long-deep document is long in the length of its text and deep in the depth of its context structure, such as a book. In a textual database consisting of long-deep documents, e.g., the Chinese History Documents Database [12], a whole document is too large to serve as the search unit. Instead of a whole document, a paragraph, a section or a page is suitable to serve as the search unit. The entire documents, the paragraphs, the sections and the pages are the contexts of documents. Hence the contexts of documents are able to serve as the search units when retrieving information from a textual database. A hypothesis is then proposed to conclude this section.

Hypothesis: A document can be completely represented in terms of its text and its context structures. The contexts of documents are able to serve as the search units when retrieving information from a textual database.

DOCUMENT REPRESENTATION

A document representation is either implicit or explicit. In an *implicit representation*, a context structure is embedded in the text and is recognized by a program.

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

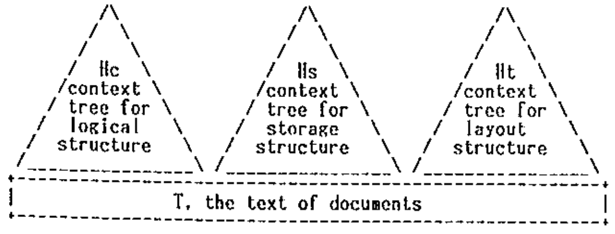


Fig. 4. An example of explicit representation.

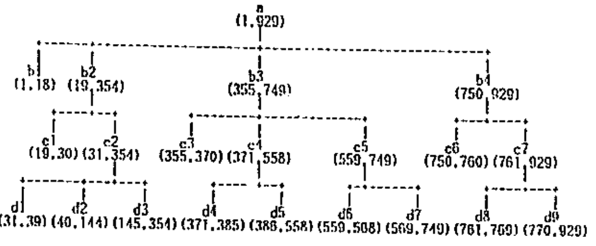


Fig. 5. An example of context tree.

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 context X is maintained. For any context tree, it is very difficult to accurately calculate the maintenance cost of a context. But for a nearly balanced context tree, the maintenance cost can be roughly calculated. A context tree is said to be *nearly balanced* if it satisfies the following properties. First, each non-leaf node of the tree has a nearly equivalent number of children. Second, the search path of each leaf node of the tree is nearly equivalent in length. Theorem 2 proves that the average maintenance cost of a leaf context on a nearly balanced context tree is about N/2, where N is the number of nodes of the tree (also the number of contexts of the context structure denoted by the tree). To maintain a context always invokes the maintenance of some leaf contexts of it. Hence it is reasonable to assert that the maintenance cost of a context is O(N). For a large textual database, because the number N is very large, the maintenance cost becomes a heavy burden when updating a context. The addressing mechanism for an explicit context tree is known as *absolute addressing*. In order to reduce the maintenance cost, a refined explicit representation using a relative addressing mechanism, called Explicit Context-Hierarchical Organization (abbr. ECHO), is proposed in Section 4. It can reduce the maintenance cost from O(N) to O(log N).

Theorem 2: Given a text and a context tree which represents the context structure of the text, if the context tree is nearly balanced, then the average maintenance cost of a leaf context is about N/2, where N denotes the number of nodes on the context tree.

Proof: From the properties of a nearly balanced context tree mentioned above, the following two assumptions are reasonably induced:

(1) For each node of the context tree, except for leaf nodes, the average number of its children is m.

(2) For each leaf node of the context tree, the average length of its search path is n. The number n also roughly denotes the number of levels on the context tree. As a convention, these levels, from the level of the root to the level of leaf nodes, are named level 1, ..., level n, respectively. It is obvious that level i, $1 \leq i \leq n$, has m^{i-1} nodes. The total number of nodes on the context tree is

$$N = \Sigma_{1\leq i\leq n}\, m^{i-1} = (m^n - 1)/(m-1). \qquad (1)$$

Suppose that a leaf context X is maintained and the maintenance causes a change in the length of the context X. It is obvious that the length of each higher-level context cntaining the context X is then changed and the location of each context behind the context X is also moved. Hence each node on the search path of the node X or on a search path in the right side of the node X must be updated. In other words, a node has to be updated if any of its descendants is updated or any node on a search path on the left side of it is updated. In order to calculate the average maintenance cost of a leaf context, the total maintenance cost of all leaf contexts, called TC, must be computed first. Based on the assumption that each leaf context is maintained with equivalent probability, TC is given by

$$\begin{aligned} TC &= \Sigma_{1\leq j\leq M}\, C_j = \Sigma_{1\leq j\leq M}\, (\Sigma_{1\leq i\leq n}\, c_{ij}) \\ &= \Sigma_{1\leq i\leq n}\, (\Sigma_{1\leq j\leq M}\, c_{ij}). \qquad (2) \end{aligned}$$

In Eq. (2), $M = m^{n-1}$ denotes the total number of leaf contexts (also leaf nodes), C_j denotes the maintenance cost of an individual leaf context, and c_{ij} denotes the number of level i nodes which must be updated to maintain an individual leaf context. For each level i, $1 \leq i \leq n$, there are $L = m^{i-1}$ nodes on this level and each of these nodes has m^{n-i} leaf elements. A node X is a leaf element of a node Y if the node X is a leaf node and either the node X is a descendant of the node Y or the node X is the same as the node Y. As a convention, all nodes on level i are named node i1, ..., node iL, from left to right. For level i, from node ik, $1 \leq k \leq L$, to node iL must be updated to maintain any leaf element of the node ik. That is, all the leaf elements of node ik have the same value of c_{ij}, which is L-k+1. Thus the value of $\Sigma_{1\leq j\leq M}\, c_{ij}$ can be calculated as follows:

$$\begin{aligned} \Sigma_{1\leq j\leq M}\, c_{ij} &= \Sigma_{1\leq k\leq L}\, m^{n-i}(L-k+1) \\ &= (m^{n-i})L(L+1)/2 \qquad (3) \end{aligned}$$

From (1), (2), and (3), the value of TC is then given by

$$\begin{aligned} TC &= \Sigma_{1\leq i\leq n}\, [m^{n-i}L(L+1)/2] \\ &= M(N+n)/2 \fallingdotseq MN/2,\ N >> n. \qquad (4) \end{aligned}$$

Thus, the average maintenance cost of a leaf context is TC/M = N/2. ■ ■

THE ECHO MODEL

Definition 6: ECHO tree. An *ECHO tree* is a context tree representing a context structure of a given text. In the ECHO tree, each node uniquely denotes a context of the context structure and carries a local name and a vector (D,L). For the root, the D denotes the beginning position of the whole text. For each of the other nodes, the D denotes the distance between the beginning positions of the contexts denoted by the node and by the parent of the node. And for any node, the L denotes the length of the context denoted by it.

Given a node X of the ECHO tree, assume that its path name is $N_1 \ldots N_m$. As a convention, the D-value and L-value of the node X are denoted by $D(X)$ and $L(X)$, or by $D(N_1 \ldots N_m)$ and $L(N_1 \ldots N_m)$, respectively. And the beginning and the end positions of the context X are denoted by $BP(X)$ and $EP(X)$, or by $BP(N_1 \ldots N_m)$ and $EP(N_1 \ldots N_m)$, respectively.

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

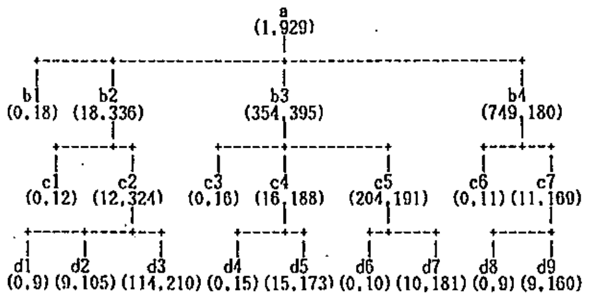
$$BP(N_1 \ldots N_m)=\Sigma_{1\leq i<m} D(N_1 \ldots N_i), \text{ and} \qquad (6)$$
$$EP(N_1 \ldots N_m)=BP(N_1 \ldots N_m)+L(N_1 \ldots N_m)-1 \qquad (7)$$

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

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

The result (569,749) is the same as (BP,EP) vector of node d_7 in Fig. 5. The search operation *get-text* is performed as follows. Apply first the operation *get-ptrs* to find the positions $BP(N_1...N_m)$ and $EP(N_1...N_m)$. Then read a subtext between the positions $BP(N_1...N_m)$ and $EP(N_1...N_m)$ from the text T. The subtext is the result.

Definition 9: *get-id, get-leafs* and *get-ids*. Assume that the root N_1 of a singular ECHO structure (T,E), the beginning and the end positions of a subtext S, and an integer m (only for the operation *get-ids*) are given.

(1) The search operation *get-id* finds the smallest context containing the subtext S from the singular ECHO structure (T,E).

(2) The search operation *get-leafs* finds a series of leaf contexts from the singular ECHO structure (T,E) such that the subtext S is contained in these leaf contexts.

(3) The search operation *get-ids* finds a series of contexts $(X_1, ..., x_n)$ from the singular ECHO structure (T,E) such that these contexts satisfy the following conditions. First, each context X_i, $1 \le i \le n$, is either a leaf context having the length of context-id no more than the integer m or a non-leaf context having the length m of context-id. And second, the subtext S is contained in these contexts.

Algorithm 1: Perform the search operation *get-id.*
Input: The root N_1 of the singular ECHO structure (T,E), and the beginning and the end positions of a subtext S, say BP and EP, respectively.
Output: A context-id $N_1...N_i$ of the tree E, which denotes the smallest context cntaining S.
Procedure:

```
if $BP \ge D(N_1)$ and $EP \le D(N_1)+L(N_1)-1$ then
  begin
    i: = 0;
    repeat
      i:=i+1; smallestflag:=true;
      $BP:=BP-D(N_1...N_i)$; $EP:=EP-D(N_1...N_i)$;
      if there exists a child of the node $N_1...N_i$, denoted
        $N_1...N_iN_{i+1}$, such that $BP \ge D(N_1...N_iN_{i+1})$
        and $EP \le D(N_1...N_iN_{i+1})+L(N_1...N_iN_{i+1})-1$
        then smallestflag:=false;
      until smallestflag=true;
      return the context-id $N_1...N_i$ as the result;
    end;
```

The search operation *get-leafs* is performed as follows. A method similar to Algorithm 1 is applied first to find two leaf context-ids C_1 and C_2 from the ECHO tree E, scuh that the BP(S) is located in C_1 and the EP(S) is located in C_2, i.e.,

$$BP(C_1) \le BP(S) \le EP(C_1) \text{ and}$$
$$BP(C_2) \le EP(S) \le EP(c_2). \qquad (9)$$

Then, all the leaf context-ids from C_1 to C_2 are searched from the ECHO tree E and are returned as the result.

The search operation *get-ids* can be performed in a way similar to the one mentioned above.

2. Constructing operations on ECHO structures

There are two operations with respect to constructing an ECHO structure: tree-constructing operation and ECHO-combining operation. The *tree-constructing operation* constructs an ECHO structure from a marked-up text. The *ECHO-combining operation* combines a number of ECHO structures into a larger one.

Definition 10: Tree-constructing operation. Given a text T associated with a number of k context structures, the *tree-constructing operation* constructs k ECHO trees representing the k context structures from the text T.

In order to construct k ECHO trees representing the k context structures from the text T, these context structures must be previously marked up. That is, for each of the context structures, an individual set of markup tokens must be previously inserted into T to identify each context of the context structure [12,22,23]. The markups have been discussed in [3,12,13,17]. When a marked-up text is available, the k ECHO trees can be constructed using Algorithm 2.

Algorithm 2: Constructs ECHO trees from a marked-up text.
Input: A marked-up text T.
Output: k ECHO trees each of which represents a context structure of T.
Procedure:
Step 1: *A context parser scans the text T to recognize each markup token and to construct k intermediate ECHO trees each of which reflects a context structure of T. In an intermediate ECHO tree, only the L-value of each leaf node is given while the other L-values and all D-values are not defined.*
Step 2: *For each intermediate ECHO tree, let D(root) of the tree be the BP(T). And then apply Theorem 3 to compute the other D-values and L-values of the tree.*

Definition 11: ECHO-combining operation. Given a number of j ECHO structures $(T_1,E_{11},...,E_{k1})$, ..., $(T_j,E_{1j},..., E_{kj})$, the *ECHO-combining operation* combines these ECHO structures to form a larger ECHO structure $(T,E_1, ..., E_k)$. That is, each given text T_i and ECHO tree E_{ni}, $1 \le n \le k$, $1 \le i \le j$, becomes a subtext of the text T and a subtree of the ECHO tree E_n, respectively.

From Definition 11, it is easy to infer that the ECHO-combining operation has to invoke one concatenation of texts and a number k of combinations of ECHO trees. The following are necessary constraints to the ECHO-combining operation. First, each given ECHO structure has exactly k ECHO trees each of which represents a kind of context structure. That is, there exist k different kinds of context structures. Second, these ECHO structures must be previously ordered by a correct sequence. Assume that the nth ECHO tree of each given ECHO structure, denoted E_{ni}, $1 \leq n \leq k, 1 \leq i \leq j$, 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

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

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

$$m \le MC_1 \le 1+K(n-1)<K(\log N). \quad (11)$$

It is clear that $MC_1 = O(\log N)$. Then consider the case where a leaf context $N_1 \ldots N_n$ is modified. From the discussion following Definition 14, the maintenance cost of this case, MC_2, is given by

$$MC_2 = n+\Sigma_{2\le i\le n} K_i. \quad (12)$$

From the assumptions given above, we have

$$n=\log N\le MC_2\le 1+K(n-1)<K(\log N) \quad (13)$$

Thus the MC_2 is also O(log N). From these cases, the theorem is proved. ■

From the discussion following Theorem 4, an explicit context tree H and an ECHO tree E have the same number of nodes if they represent the same context strucutre. Hence from Theorem 5 and the discussion before Theorem 2, it is obvious that the maintenance cost of a context can be reduced from O(N) to O(log N) if the explicit context tree is replaced by the equivalent ECHO tree.

4. Set operations on ECHO structures

Definition 15: Context set. A *context set*, denoted $C=(X_1, \ldots, X_m)$, is defined as a set of some contexts of an ECHO strucutre, which satisfies the following properties.

(1) In the set, there does not exist any repeated context, i.e., $X_i \ne X_j$ for $i\ne j$, $1\le i\le m$ and $1\le j\le m$.

(2) In the set, for each context x_i, $1\le i\le m$, there does not exist any context X_j, $1\le j\le m$ and $i\ne j$, such that X_i is contained in X_j.

In addition, each X_i, $1\le i\le m$, is a *member* of the set C.

A traditional set is defined as a collection of entities. A context set differs from a traditional set in members and operations. In general, a member (entity) of the set is not divisible. For example, given a set S={(a,b),c,d}, (a,b) is a member of S but neither (a) nor (b) is a member of S. It is clear that if a context X is in a context set C, then each subtext of X is also in C. That is, if the context X consists of a number of smaller contexts $Y_1, \ldots, Y_k$, then each Y_i, $1\le i\le k$, has to be a member of C. Hence a member of the context set, except for a leaf context, is divisible. The traditional set operations are union, intersection, difference and the Cartesian product. It is obvious that the Cartesian product is not applicable to context sets. In addition, there are some differences between the union, intersection and difference of context sets and those of traditional sets.

Theorem 6: A context set $C=(X_1, \ldots, X_m)$ is uniquely denoted by a context-id set $C'=(X_1', \ldots, X_m')$ in which each context X_i is uniquely denoted by the context-id X_i', $1\le i\le m$. In addition, the context-id set satisfies the following properties:

(1) In the set, no repeated context-id exists, i.e., $X_i' \ne X_j'$ for $i\ne j$, $1\le i\le m$ and $1\le j\le m$.

(2) In the set, for each context-id X_i', $1\le i\le m$, no context-id X_j', exists $1\le j\le m$, such that X_j' is a proper prefix of X_i'.

Proof: From Definitions 6 and 15, it is easy to prove this theorem. ■

Definition 16: A context X is said to be in a context set C (or X is a member of C) if and only if either the context X or a context containing X is a member of the set C.

It is clear that Definition 16 also holds for the context-id set. Assume that the context X and the context set C are denoted by the context-id X' and the context-id set C', respectively. Definition 16 can be interpreted as that a context-id X' is said to be in a context-id set C' (or X' is a member of C') if and only if either the context-id X' or a proper prefix of the context-id X' is a member of the set C'.

Definition 17: The union, intersection and difference of context sets are defined as follows.

(1) Union. The *union* of two context sets A and B, denoted $A\cup B$, is a context set in which each context is in A or B or both.

(2) Intersection. The *intersection* of two context sets A and B, denoted $A\cap B$, is a context set in which each context is in both A and B.

(3) Set difference. The *difference* of two context sets A and B, denoted $A-B$, is a context set in which each context is in A but not in B.

Because a context set is uniquely denoted by a context-id set, the union, intersection and difference of context sets can be actually represented by these operations of context-id sets. Given two context sets A and B, assume that they are denoted by the context-id sets $A'=(X_1, \ldots X_m)$ and $B'=(Y_1, \ldots, Y_n)$, respectively. $A\cup B$, $A\cap B$ and $A-B$ can be respresented by $A'\cup B'$, $A'\cap B'$ and $A'-B'$, respectively. In addition, the operations performing the union and the intersection of two sets are named by *OR-merge* and *AND-merge*, respectively.

Union of context-id sets. $A'\cup B'$ is performed by OR-merging the sets A' and B'. First, for each context-id X_i, $1\le i\le m$, if there does not exist any context-id Y_j, $1\le j\le n$, such that Y_j is a proper prefix of X_i, then X_i appears in $A'\cup B'$. Second, for each context-id Y_j, $1\le j\le n$, if there does not exist any context-id X_i, $1\le i\le m$, such that either $Y_j = X_i$ or X_i is a proper prefix of Y_j, then Y_j appears in $A'\cup B'$. And third, nothing else appears in $A'\cup B'$.

Intersection of context-id sets. The $A'\cap B'$ is performed by AND-merging the sets A' and B'. First, for each context-id X_i, $1\le i\le m$, if there exists a context-id Y_j, $1\le j\le n$, such that either $X_i = Y_j$ or Y_j is a proper prefix of X_i, then X_i appears in $A'\cap B'$. Second, for each context-id Y_j, $1\le j\le n$, if there exists a context-id X_i, $1\le i\le m$, such that X_i is a proper prefix of Y_j, then Y_j appears in $A'\cap B'$. And third, nothing else appears in

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.

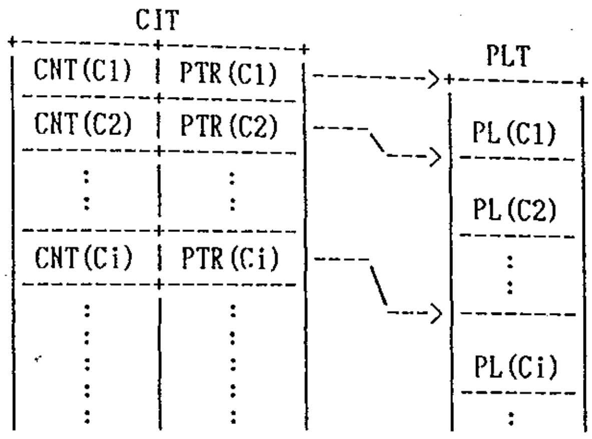


Fig. 7. The index structure of ARCIM.

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 the ECHO structure (T,E).

Algorithm 4: Perform the ARCIM-creating operation.

Input: A singular ECHO structure (T,E).

Output: An ARCIM structure (CIT,PLT) for the ECHO structure (T,E).

Procedure:

Step 1: *Create a new CIT in which the number of entries is equivalent to the number of Chinese characters used in the computer system. For each entry of the CIT, let $CNT(C_i)=0$ and let $PTR(C_i)=nil$. Then create a new PLT which contains nothing.*

Step 2: *For each leaf context X of the ECHO structure (T,E), perform the following operations.*

(1) *Sequentially scan the context X. If the scanned character is not a Chinese character, then remove it from the context X. The result is a string consisting of only Chinese characters, called string X'.*

(2) *Sort the string X' by a nondecreasing order of character codes.*

(3) *Sequentially scan the sorted string X'. If the scanned character is the same as the preceding one, then remove it from the string X'. The result is the string without any repeated character, called CLIST(X).*

(4) *Append the leaf context-id X to each character C_i of the CLIST(X). The result is called CIDLIST(X). In the CIDLIST(X), each entry is a pair of a character Ci and the leaf context-id X, denoted CIDPAIR (C_i,X).*

Step 3: *According to the sequence from the leftmost leaf context to the rightmost leaf context of T, concatenate all the CIDLISTs to form a CIDTABLE.*

Step 4: *Sort the CIDTABLE in nondecreasing order of the character codes. The result is an initial PLT. In the initial PLT, it is clear that all the CIDPAIRs having the same character C_i are grouped together to form a segment, i.e., an initial $PL(C_i)$. And in an initial $PL(C_i)$, it is clear that all the CIDPAIRs are ordered in ascending order of the leaf context-ids.*

Step 5: *Scan the initial PLT to find each initial $PL(C_i)$. At the same time, for each initial $PL(C_i)$, perform the following operations.*

(1) *Compute both the starting address (in number of CIDPAIRS) and the number of CIDPAIRs of the initial $PL(C_1)$, which are the $PTR(C_i)$ and the $CNT(C_i)$, respectively.*

(2) *Write both the $CNT(C_i)$ and the $PTR(C_i)$ into a proper entry of the CIT and append the context-ids of the initial $PL(C_i)$, i.e., the $PL(c_i)$, to the PLT.*

Algorithm 4 has been successfully applied to create an ARCIM structure of the CED, which is an experimental Chinese electronic dictionary [22]. In the CED, the space overhead required to store the ARCIM structure, 1.9 Mbytes, is about 30% of the size of the text files, 6.2 Mbytes. The work required to create ARCIM structures has to be done only once. Even though the ARCIM-creating operation consumes a lot of computing hours, it is endurable.

2. The maintenance of ARCIM structures

For a singular ECHO structure (T,E) and an attached ARCIM structure (CIT, PLT), it is clear that the ARCIM structure has to be maintained if the ECHO structure is updated. According to the updating operations *echoinsert*, *echodelete* and *leafmodify*, there are three corresponding updating operations of ARCIM structure proposed in Algorithms 5, 6 and 7, respectively.

Algorithm 5: Update the ARCIM structure (CIT,PLT) when a leaf context X is inserted into the ECHO structure (T,E).

Procedure:

Step 1: *Construct the CLIST(X) by using step 2 of Algorithm 4. Then sort the CLIST(X) in ascending order of the character codes. Assume that $CLIST(X)=C_1\ldots C_j$.*

Step 2: *According to each character C_i of CLIST(X), $1\le i\le j$, insert the context-id X into the $PL(C_i)$, keeping the ascending order of context-ids.*

Step 3: *Scan the CIT from the entry for C_1 to the end. Assume that the entry currently scanned is of a character A.*

(1) *If $A=C_i$, $1\le i\le j$, then $CNT(A):=CNT(A)+1$.*

(2) *If $C_i<A\le C_{i+1}$, $1<j$, then $PTR(A):=PTR(A)+i$.*

(3) *For each A, $A>C_j$, do $PTR(A):=PTR(A)+j$.*

Algorithm 6: Update the ARCIM structure (CIT,PLT) when a leaf context X is deleted from the ECHO structure (T,E).

Procedure:

Step 1: *Construct the CLIST(X) by using step 2 of Algorithm 4. Then sort the CLIST(X) in ascending order of the character codes. Assume that $CLIST(X)=C_1\ldots C_j$.*

Step 2: *According to each character C_i of CLIST(X), $1\le i\le j$, remove the context-id X from the $PL(C_i)$.*

Step 3: *Scan the CIT from the entry for C_1 to the end. Assume that the entry currently scanned is of a character A.*

(1) *If $A=C_i$, $1\le i\le j$, then $CNT(A):=CNT(A)-1$.*

(2) *If $C_i<A\le C_{i+1}$, $1<j$, then $PTR(A):=PTR(A)-i$.*

(3) *For each A, $A>C_j$, do $PTR(A):=PTR(A)-j$.*

Algorithm 7: Update the ARCIM structure (CIT,PLT) when a leaf context X of the ECHO structure (T,E) is modified. Assume that the modified version of the context X is named context X'.

Procedure:

Step: *Construct the CLIST(X) and the CLIST(X') by using step 2 of Algorithm 4. Then sort the CLIST(X) and the CLIST(X') in ascending order of the character codes.*

Step 2: *Compare the CLIST(X) with the CLIST(X') to obtain two strings $B_1\ldots B_n=CLIST(X')-CLIST(X)$ and $D_1\ldots D_m=CLIST(X)-CLIST(X')$, where the sym-*

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 consistence check, the query processor divides the search-clause into a number of search phrases. The query processor evaluates each search phrase to obtain a phrase-level posting list in which each leaf context-id probably satisfies the search phrase, i.e., the *search process.* Then the query processor processes and OR-merges all phrase-level posting lists to obtain a clause-level posting list as the result, i.e., the *post process.* In order to provide better search performance, a bottom-up and greedy method is used in both the search process and the post process.

Algorithm 8: Evaluate a given search phrase to obtain a phrase-level posting list.

Procedure:

Step 1: *Reconstruct each search phrase using the following operations.*

(1) *Eliminate each term following an AND NOT operator and all operators from the search phrase. The result is a string containing only Chinese characters.*

(2) *Eliminate all repeated characters from the string obtained in* (1).

(3) *Using the CIT specified by the scope clause, sort the remaining characters in a nondescreasing order of CNTs.*

Step 2: *For each reconstructed search phrase, say $B_1 \dots B_k$, perform the following operations to obtain a phrase-level posting list. If $CNT(B_1) = 0$, then return an empty posting list as the result. Otherwise, AND-merge $PL(B_1)$ and $PL(B_2)$ to form $PL(B_1B_2)$, then AND-merge $PL(B_1B_2)$ and $PL(B_3)$ to form $PL(B_1B_2B_3)$, and so on, until either an empty posting list is obtained or $PL(B_1B_2 \dots B_k)$ is formed. The $PL(B_1 \dots B_i)$, $1 \le i \le k$, denotes a posting list in which each leaf context contains all the characters B_1, ..., and B_i.*

It is clear that the reconstructed search phrase is a string consisting of a number of non-repeated characters, say $B_1 \dots B_k$. The string $B_1 \dots B_k$ has the following properties:

Property 6: $B_i \ne B_j$ for $i \ne j$.

Property 7: $CNT(B_i) \le CNT(B_j)$ for $i < j$.

Property 8: $PL(B_1 \dots B_i) = PL(B_1) \cap \dots \cap PL(B_i)$ for $1 \le i \le k$.

Property 9: $PL(B_1 \dots B_k) \subseteq PL(B_1 \dots B_{k-1}) \subseteq \dots \subseteq PL(B_1)$.

Property 10: $CNT(B_1) = 0$ implies $PL(B_1 \dots B_i) = null$, $1 \le i \le k$.

These properties provide an important basis for step 2 of Algorithm 8. First, by applying Property 10, if CNT(B1)=0, we immediately let the phrase-level posting list to null rather than apply AND-merges. Second, from Properties 7, 8 and 9, it is obvious that step 2 applies a shortest-first strategy to the AND-merges for a reconstructed search phrase. And it is easy to find the similarities between the shortest-first strategy applied to the AND-merges and the solution to the problem "optimal storage on tapes", i.e., a greedy method [10]. For a search process, the major factors of the performance are the number of AND-merges and the number of leaf context-ids invoked by the AND-merges. Evaluation of the reconstructed search phrase shows that its performance is better than that of its original form. The reasons are as follows. First, it is obvious that a reconstructed search phrase contains less characters than its original form. Hence the evaluation of a reconstructed search phrase needs less AND-merges and less leaf context-ids than that of its original form. Second, in Algorithm 8, a greedy manner is applied. Referring to [10], it is easy to prove that the AND-merges invoke the smallest total number of leaf context-ids.

However, the reconstruction of a search phrase is an information-lost operation. That is, some constraints provided by an original search phrase are ignored. The lost information includes the terms following AND NOT operators, the wild-card operators contained in terms, the relationships among characters within terms and the relationships among terms. Hence some "false-drops" will occur in a phrase-level posting list. A leaf context-id is a false drop if it satisfies the reconstructed search phrase but does not satisfy the original search phrase. In addition, if either the phrase "UNDER $N_1 \dots N_j$" or the phrase "FORM $A_1 \dots A_j$ TO $B_1 \dots B_k$" is given as the scope clause, then a leaf context-id which does not satisfy the scope clause is also a false-drop.

2. The post process

The post process consists of three operations: the elimination of false-drops, the adjustment of context-ids, and the merge operations.

Elimination of false-drops. In order to improve the precision of the query processing, the false-drops have to be eliminated from each phrase-level posting list. There are three cases that have to be considered. First, if the scope clause "UNDER $N_1 \dots N_j$" is given, then for each phrase-level posting list, all the leaf context-ids having no prefix $N_1 \dots N_j$ must be eliminated. Second, if the scope clause "FROM $A_1 \dots A_j$ TO $B_1 \dots B_k$" is given, then for each phrase-level posting list, all the leaf context-ids which are out of the range from $A_1 \dots A_j$ to $B_1 \dots B_k$ have to be ignored. And finally, for a given search phrase SP_i, assume the phrase-level posting list PPL_i is obtained by the search process. For each leaf context-id X contained in the PPLi, the post process must scan the leaf context X and performs the following actions.

(1) If the context X contains a term which is specified in the SP_i and follows an AND NOT operator, the context-id X has to be eliminated.

(2) If the context X does not contain all character strings specified in the SP_i, except for the strings following AND NOT operators, the context-id X must be ignored.

(3) If the context X does not satisfy all the constraints

sistence check, the query processor divides the search-clause into a number of search phrases. The query processor evaluates each search phrase to obtain a phrase-level posting list in which each leaf context-id probably satisfies the search phrase, i.e., the *search process.* Then the query processor processes and OR-merges all phrase-level posting lists to obtain a clause-level posting list as the result, i.e., the *post process.* In order to provide better search performance, a bottom-up and greedy method is used in both the search process and the post process.

Algorithm 8: Evaluate a given search phrase to obtain a phrase-level posting list.

Procedure:

Step 1: *Reconstruct each search phrase using the following operations.*

(1) *Eliminate each term following an AND NOT operator and all operators from the search phrase. The result is a string containing only Chinese characters.*

(2) *Eliminate all repeated characters from the string obtained in* (1).

(3) *Using the CIT specified by the scope clause, sort the remaining characters in a nondescreasing order of CNTs.*

Step 2: *For each reconstructed search phrase, say $B_1 \ldots B_k$, perform the following operations to obtain a phrase-level posting list. If $CNT(B_1)=0$, then return an empty posting list as the result. Otherwise, AND-merge $PL(B_1)$ and $PL(B_2)$ to form $PL(B_1B_2)$, then AND-merge $PL(B_1B_2)$ and $PL(B_3)$ to form $PL(B_1B_2B_3)$, and so on, until either an empty posting list is obtained or $PL(B_1B_2 \ldots B_k)$ is formed. The $PL(B_1 \ldots B_i)$, $1 \leq i \leq k$, denotes a posting list in which each leaf context contains all the characters B_1, ..., and B_i.*

It is clear that the reconstructed search phrase is a string consisting of a number of non-repeated characters, say $B_1 \ldots B_k$. The string $B_1 \ldots B_k$ has the following properties:

Property 6: $B_i \neq B_j$ for $i \neq j$.

Property 7: $CNT(B_i) \leq CNT(B_j)$ for $i < j$.

Property 8: $PL(B_1 \ldots B_i) = PL(B_1) \cap \ldots \cap PL(B_i)$ for $1 \leq i \leq k$.

Property 9: $PL(B_1 \ldots B_k) \subseteq PL(B_1 \ldots B_{k-1}) \subseteq \ldots \subseteq PL(B_1)$.

Property 10: $CNT(B_1)=0$ implies $PL(B_1 \ldots B_i)=null$, $1 \leq i \leq k$.

These properties provide an important basis for step 2 of Algorithm 8. First, by applying Property 10, if CNT(B1)=0, we immediately let the phrase-level posting list to null rather than apply AND-merges. Second, from Properties 7, 8 and 9, it is obvious that step 2 applies a shortest-first strategy to the AND-merges for a reconstructed search phrase. And it is easy to find the similarities between the shortest-first strategy applied to the AND-merges and the solution to the problem "optimal storage on tapes", i.e., a greedy method [10]. For a search process, the major factors of the performance are the number of AND-merges and the number of leaf context-ids invoked by the AND-merges. Evaluation of the reconstructed search phrase shows that its performance is better than that of its original form. The reasons are as follows. First, it is obvious that a reconstructed search phrase contains less characters than its original form. Hence the evaluation of a reconstructed search phrase needs less AND-merges and less leaf context-ids than that of its original form. Second, in Algorithm 8, a greedy manner is applied. Referring to [10], it is easy to prove that the AND-merges invoke the smallest total number of leaf context-ids.

However, the reconstruction of a search phrase is an information-lost operation. That is, some constraints provided by an original search phrase are ignored. The lost information includes the terms following AND NOT operators, the wild-card operators contained in terms, the relationships among characters within terms and the relationships among terms. Hence some "false-drops" will occur in a phrase-level posting list. A leaf context-id is a false drop if it satisfies the reconstructed search phrase but does not satisfy the original search phrase. In addition, if either the phrase "UNDER $N_1 \ldots N_j$" or the phrase "FORM $A_1 \ldots A_j$ TO $B_1 \ldots B_k$" is given as the scope clause, then a leaf context-id which does not satisfy the scope clause is also a false-drop.

2. The post process

The post process consists of three operations: the elimination of false-drops, the adjustment of context-ids, and the merge operations.

Elimination of false-drops. In order to improve the precision of the query processing, the false-drops have to be eliminated from each phrase-level posting list. There are three cases that have to be considered. First, if the scope clause "UNDER $N_1 \ldots N_j$" is given, then for each phrase-level posting list, all the leaf context-ids having no prefix $N_1 \ldots N_j$ must be eliminated. Second, if the scope clause "FROM $A_1 \ldots A_j$ TO $B_1 \ldots B_k$" is given, then for each phrase-level posting list, all the leaf context-ids which are out of the range from $A_1 \ldots A_j$ to $B_1 \ldots B_k$ have to be ignored. And finally, for a given search phrase SP_i, assume the phrase-level posting list PPL_i is obtained by the search process. For each leaf context-id X contained in the PPLi, the post process must scan the leaf context X and performs the following actions.

(1) If the context X contains a term which is specified in the SP_i and follows an AND NOT operator, the context-id X has to be eliminated.

(2) If the context X does not contain all character strings specified in the SP_i, except for the strings following AND NOT operators, the context-id X must be ignored.

(3) If the context X does not satisfy all the constraints

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).

4. Date, C.J., *An Introduction to Database Systems*, 4th Ed., Addison-Wesley, Mass. (1986).
5. Emrath, P.A., "Page Indexing for Textual Information Retrieval Systems," Ph.D. Diss., Univ. of Illinois at Urbana-Champaign (1983).
6. Faloutsos, C., "Access Methods for Text," *Comput. Surv.*, Vol. 17, No. 1, pp. 49-74 (1985).
7. Fan, C.K. and W.H. Tsai, "Automatic Word Indentification in Chinese Sentences by the Relexation Technique," *Computer Processing of Chinese and Oriental Languages*, Vol. 4, No. 1, pp. 33-56 (1988).
8. Haskin, R.L., "Special-Purpose Processors for Text Retrieval," *Database Engineering*, Vol. 4, No. 1, pp. 16-29 (1981).
9. Hollaar, L.A., "Text Retrieval Computers," *Computer*, Vol. 12, No. 3, pp. 40-50 (1979).
10. Horowitz, E. and S. Sahni, *Fundamentals of Comptuer Algorithms*, Computer Science Press Inc., Potomac, Md. (1987).
11. Hou, W.H., "Automatic Recognition of Chiense Words," Master Thesis, National Taiwan Institute of Technology (NTIT), Taipei (1983) (in Chinese).
12. Hsieh, C.C., Z.K. Ding, Y.H. Wang, S.F. Tung, S. Lin and C.C. Shu, "Full-Text Database for Chinese History Documents," *Proc. of* 1988 *International Conference on Computer Processing of Chinese and Oriental Languages (ICCPCOL)*, Toronto, Canada, pp. 334-340 (1988).
13. International Organization for Standardization, *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, Draft International Standard ISO/DIS 8879 (1985).
14. Knuth, D.E., J.H. Morris and V.R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Comput.*, Vol. 6, No. 2, pp. 323-350 (1977).
15. Knuth, D.E., *The Art of Computer Programming: Vol. 1, Fundamental Algorithms*, 2nd Ed., Addison-Wesley, Mass. (1973).
16. Ozkarahan, E., *Database Machines and Database Management*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1986).
17. Peels, A.J.H.M., N.J.M. Janssen and W. Nawijn, "Document Architecture and Text Formatting," *ACM Trans. Off. Inf. Syst.*, Vol. 3, No. 4, pp. 347-369 (1985).
18. Salton, G. and M.J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York (1983).
19. Sproat, R. and C.L. Shih, "A Statistical Method for Finding Word Boundaries in Chiense Text," *Proc. of* 1989 *International Conference on Computer Processing of Chinese and Oriental Languages*, Changsha, China (1990).
20. Tseng, S.S., "The Design of Chinese Character Database," Master Thesis, National Taiwan Institute of Technology, Taipei (1982) (in Chinese).
21. Tseng, S.S., "CCCII: The Coding Scheme and the Applications," *Communications of Computing Center, Academia Sinica*, Vol. 2, Nos. 5-8 and 12-13 (1986) (in Chinese).
22. Tseng, S.S., M.Y. Chang, C.C. Hsieh and K.J. Chen, "Approaches on an Experimental Chinese Electronic Dictionary," *Proc. of* 1988 *International Conference on Computer Processing of Chinese and Oriental Languages*, pp. 371-374 (1988).
23. Tseng, S.S., C.C. Yang and C.C. Hsieh, "The Document Representation and a Refined Character Inversion Method for Chinese Textual Database," *Proc. of* 1988 *International Conference on Computer Processing of Chinese and Oriental Languages*, pp. 376-381 (1988).
24. Ullman, J.D., *Principles of Database Systems*, 2nd Ed., Pitman Publishing Ltd., London (1982).

Discussions of this paper may appear in the discussion section of a future issue. All discussions should be submitted to the Editor-in-Chief.

Manuscript Received: July 14, 1989
Revision Received: May 18, 1990
and Accepted: June 6, 1990