COMPUTER PROCESSING OF CHINESE & ORIENTAL LANGUAGES
Vol. 4, Nos. 2 & 3, July 1989

On the Design of Chinese Textual Database

Shih-shyeng Tseng, Chen-chau Yang and Ching-chun Hsieh

ABSTRACT

In this paper, an architecture of textual database based upon ECHO model is proposed. The ECHO model provides a representation for expressing documents and a set of operations on the representation that serves to express queries and other manipulations on documents. It has the abilities to provide multiple context structures of documents, a flexible search unit for retrieving textual information, and a subrange search on a textual database. In addition, the ECHO tree is relatively easy to maintain.

The inversion of terms is the most-commonly-used retrieval method for textual database. Its advantages are that it is relatively easy to implement and is fast. In order to improve the query performance, a refined character inversion method for Chinese textual database, called ARCIM, is also proposed in the paper. The ARCIM can retrieve texts faster than a simple character inversion method.

Keywords: Textual database, document model, document retrieval.

1. INTRODUCTION

The differences between a conventional database and a textual database, which are discussed in [Falo85] and [Ozka86], can be summarized as follows. First, the conventional database deals with retrieval and manipulation of formatted records but the textual database with documents. Second, formatted records differ from documents in the data structure, the query language, and operational requirements such as update frequency and size of database. Both in [Falo85] and [Ozka86], a document is considered as a non-structured data consisting of an arbitrary number of words. This argument is not true at all. On the contrary, the document is also a structured data only its structure is not similar to the structure of formatted records.

Final manuscript received on July 10, 1989. S.-s. Tseng is with the Computing Center, Academia Sinica, Nankang, Taipei, Taiwan 11529; C.-c. Yang is with the Department of Electronic Engineering, National Taiwan Institute of Technology, No. 43, Keelung Rd., Sec. 4, Taipei, Taiwan 10772; and C.-c. Hsieh is with the Institute of Information Science, Academia Sinica, Nankang, Taipei, Taiwan 11529.

Since documents are the constituents of a textual database, the development of textual database has to be based upon an underlying model for documents. The model for documents, similar to conventional data models [Ullm82], consists of two components:
(1) a representation for expressing the documents, and
(2) operations on the representation that serves to express queries and other manipulations on documents.

A document can be defined in two ways: from the viewpoint of its functions and from the viewpoint of its constituents. The former, adapted from [PeJN85], 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 latter simply defines a document as a text associated with one or more structures as discussed in [BeRG88], [Hora85], [Hsie88], [ISO8613], [PeJN85], and [TsYH88]. In this paper, a document is regarded as the both. The structures of documents will be discussed in detail in sections 2 and 3, and a representation for expressing the structures of documents, called ECHO, will be introduced in section 4.

An architecture of textual database based upon the ECHO model is shown in figure 1. It is composed of five modules: an ECHO subsystem, a text retrieval subsystem, a user interface, a query processor, and a maintenance subsystem. The ECHO subsystem provides a mechanism for storing the documents and their structures. It will be introduced in section 4.4. The text retrieval subsystem provides an important mechanism to improve the retrieval performance, which will be discussed in section 5. The user interface accepts and recognizes the requests given from users. A user request may be a query expression or a maintenance command. If a query expression is given, it will be passed to the query processor. By means of the text retrieval subsystem and the ECHO subsystem, a set of contexts satisfying the query expression can be found by the query processor. The query processing will be discussed in section 6. The maintenance subsystem provides the necessary operations for maintaining the ECHO subsystem and the text retrieval subsystem. It will be mentioned in detail in section 7.

2. DOCUMENT REPRESENTATION

Definition-1: Text. A *text* is a heterogeneous data string consisting of a sequence of text components of various types. *Text components* may be symbols, words, phrases, or sentences in natural or artificial languages, figures, formulas, or tables. A set of text components are said to be in the same type if and only if they are represented by the same set of notations and manipulated by the same set of operations.

From definition-1, it is clear that the concatenation of a number of texts forms a larger text and the segmentation of a text forms a number of smaller texts. The text components can be roughly classified into two types: character type and non-character type. A text component of the character type is one that consists of only characters, such as a symbol, a word, a phrases, a sentence, and sometimes a formula or a table. The figures such as

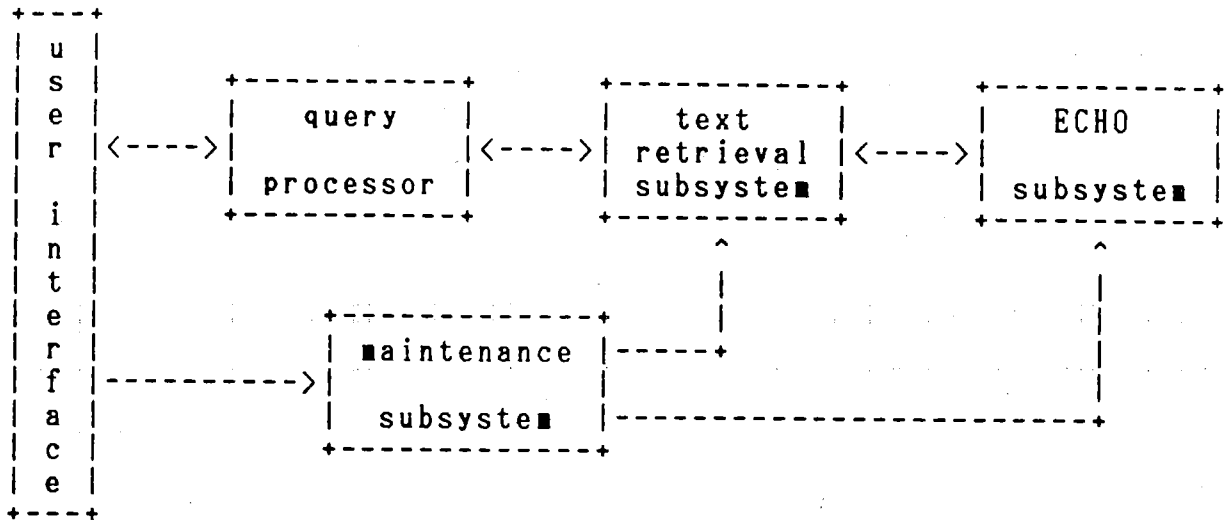

Figure 1 A proposed architecture for textual database

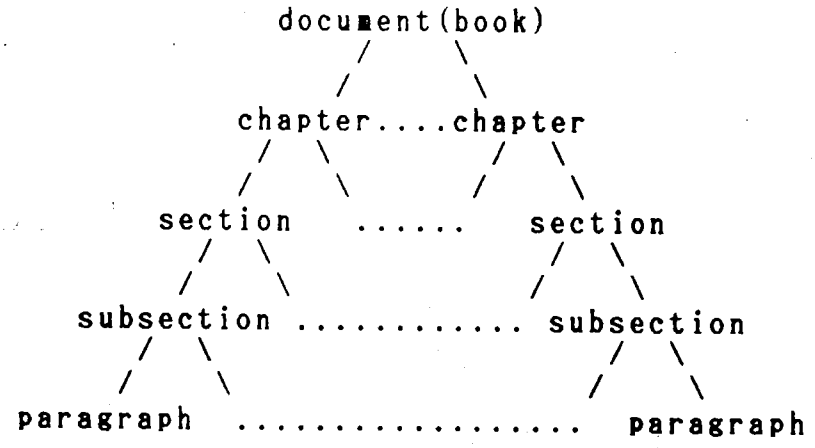

Figure 2 An example of logical structure

pictures, diagrams, drawings, paintings, or images, by contrast, are the text components of non-character type. In this paper, only the retrieval on the character type is regarded. The text components of the non-character type are excluded from the retrieval operation at current state.

From the definition of document which mentioned in section 1, the necessary condition that a text forms a document is that the text implicitly contains a complete set of the author's thoughts on a specific topic. In practice, when an author writes a document, he first organizes a number of text components to form a basic text element, e.g., a paragraph. Then he organizes a number of the basic text elements to form a larger text element, e.g., a section, and so on, until the document is formed. That is, in order to reflect the conceptual skeleton of the author's thoughts, the text of a document must be organized into a logical structure, e.g., as shown in figure 2. The *logical structure* is defined as a hierarchy of text elements, in which each text element, except for a basic text element, is a composite of a set of smaller text elements. Where a *text element* is a part of the text or the whole text that forms a meaningful unit of a document, e.g., a paragraph, a section, a chapter, a document, etc. A text element which contains no any smaller text element is called a *basic text element*. A text element, by contrast, is a *structured text element* if it contains some subordinate text elements.

The logical structure of a document is always presented in a readable manner. For example, a paragraph starts with a new line and the first word of it usually follows some leading blanks; sections are distinguished from each other by some space lines; a chapter begins from the top of a new page; and the titles of chapters or sections are printed in an individual line and by enlarged fonts. The one such as mentioned above that formats the text of a document in a representation medium, e.g., paper or screen, is called the layout structure of the document. In other words, the layout structure of a document explicitly reflects the formatting of the text and the logical structure of the document in a representation medium. The *layout structure* is defined as a 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 figure 3. Sometimes, a document may have more than one layout structure in which each represents the formatting of the text in an individual medium and for an individual version, e.g., a normal size version or a packet size version. For any document on a particular medium and on a particular version, the logical structure and the layout structure are mixed and they are embedded in the text. The logical structure and the layout structure of a document and the relationship between them have been discussed in detail in [Hora85], [ISO8613], and [PeJN85].

Definition-2: Context. Given a text, a *context* is defined as follows:
(1)The whole text is a context.

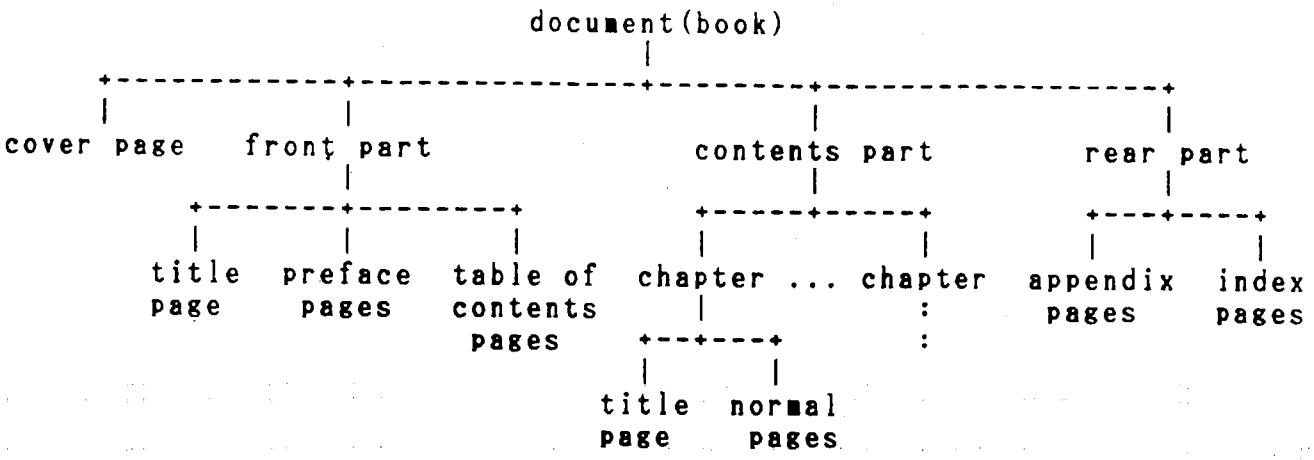

Figure 3 An example of layout structure

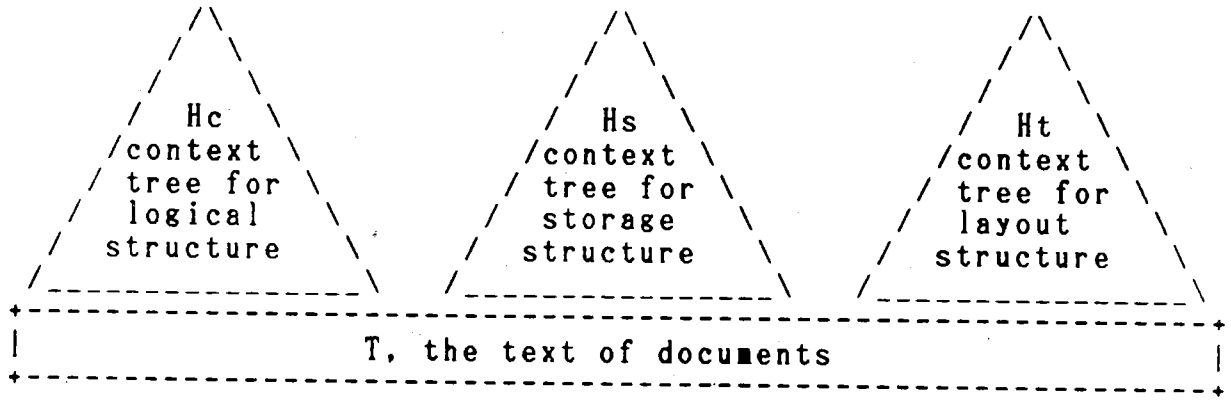

Figure 4 An example of explicit representation

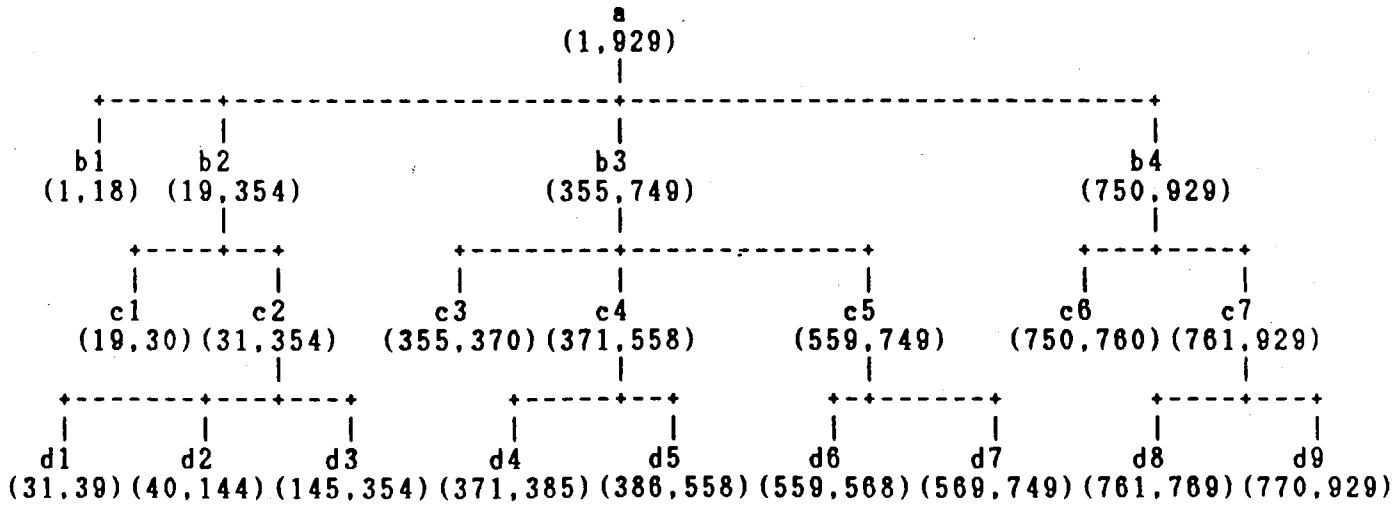

Figure 5 An example of context tree

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

In definition-2, the whole text specified in (1) is referred to as the root context or the $level_1$ context. The partitioning process specified in (2) is referred to as the *hierarchical partitioning*. That is, the root context can be partitioned into a set of $level_2$ contexts, then a $level_2$ context can be further partitioned into a set of $level_3$ contexts, and so on. In other words, formally, a $level_i$ context can be partitioned into a set of $level_{i+1}$ contexts. A context Y is said to be (entirely) contained in a context X, or reversely, the context X is said to (fully) 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 that it contains no any lower level context.

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

Proof: Referred to the definition of a tree proposed in [Knut73], the proof is given as follows.

From definition-2, the given text is the $level_1$ context. Depending upon whether the $level_i$ context, i=1, is partitioned into a set of $level_{i+1}$ contexts or not, there are two cases have 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 set of $level_{i+1}$ contexts. The necessary condition that the $level_i$ context forms a tree structure is that each of the $level_{i+1}$ contexts contained in the $level_i$ context forms a tree structure. Thus the problem is turned into that to show each of these $level_{i+1}$ contexts also forms a tree structure. For each $level_{i+1}$ context which is not partitioned, similar to case 1, it forms a special tree structure having only a root. For each $level_{i+1}$ context which is partitioned into a set of $level_{i+2}$ contexts, let i+1 be i and recursively apply the 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. □

Theorem-2: The contexts in a context structure satisfy the *property of hierarchical partitioning*. That is, given any two contexts of the context structure, say contexts A and B, only one of the followings holds:

(1) context A is entirely contained in context B,

(2) context B is entirely contained in context A, and

(3) context A and context B have no common subtext.

Proof. Referred to definition-2, it is obvious that only one of the three relationships exists between the contexts A and B:

(1)The context A is partitioned from the context B. For this case, it is clear that the context A is entirely contained in the context B.

(2)The context B is partitioned from the context A. For this case, it is clear that the context B is entirely contained in the context A.

(3)Both the context A and the context B are partitioned from a context of a higher level, say context X for any. For the case, the context A and the context B must be nonoverlapped. Thus the context A and the context B have no common subtext.

From these discussions, the theorem is proved. □

From the previous argument, a document has three important constituents: the text, the logical structure and the layout structure. It is clear that the text elements within a document is hierarchially partitioned from the text of the document. For example, referred to figure 2, the text of document is first partitioned into a number of nonoverlapped but concatenated pieces of text in which each forms a chapter; the text of a chapter is then partitioned into several nonoverlapped but concatenated subtexts in which each forms a section; and so on. From theorem-1, the logical structure of a document is a context structure. In addition, from theorem-2 each text element within the context structure satisfies the property of hierarchical partitioning. On the other hand, if a page is seen as the basic layout element, then the layout structure of a document is also a context structure.

Since the context structure of a document is really a tree structure, it can be explicitly represented by a tree, called *context tree*. In the context tree, each node denotes a context of the document. The properties of a context tree, as shown as follows, are easily derived from definition-2, theorem-1, and the natures of a document.

Prop.1: A subtree of the context tree is also a context tree.

Prop.2: Given any two nodes of the context tree, say nodes X and Y, the node Y is a *descendant* of the node X if and only if the context denoted by the node Y is entirely contained in the context denoted by the node X. Conversely, the node X is an *ancestor* of the node Y. In addition, if there does not exist any node Z such that the node Z is a descendant of the node X and the node Z is also an ancestor of the node Y, then the node Y is a child of the node X or the node X is the parent of the node Y.

Prop.3: For any node of the context tree, the number of its children cannot be prespecified by a constant. That is, a context contains a non-predictable number of smaller contexts.

Prop.4: 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.

Prop.5: If nodes Y_1 , Y_2 ,..., Y_m are the children of node X in the context tree, then the length of the context denoted by the node X is the sum of the lengths of the contexts denoted by the nodes Y_1 , Y_2 ,..., Y_m . In addition, for any context, its length cannot be prespecified by a constant.

To be short, there are some conventions used in this paper. First, the context denoted by a 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 node, i.e., the *context tree X* means that it has the root node X. Third, the *height* of a context tree is referred to as the maximal number of levels of the tree from the root to leaf nodes.

From the arguments above, a document can be completely represented in terms of its text and its context structures. Two documents are said to be of the same class if they have similar lengths of texts and similar heights of context trees. A textual database is always composed of a set of documents of the same class. Consider two different classes of documents: short-shallow documents and long-deep documents. A short-shallow document is short in the length of its text and shallow in the height of its context tree, such as an abstract, a bibliography, a letter, etc. In general, it has the length of text no more than a thousand words and the height of context tree no more than three. For a textual database which is composed of short-shallow documents, a whole document is always served as the search unit for retrieving textual information from it. On the other hand, a long-deep document is long in the length of its text and deep in the height of its context tree, such as a book. In general, it has the length of text from thousands of words to millions of words and the height of context tree no less than four. For a textual database which is composed of long-deep documents, such as the Chinese History Documents Database [Hsie88], a whole document is too large to be served as the search unit for retrieving textual information from it. Instead of a whole document, a paragraph, a page, or a section is suitable to be served as the search unit. All of the whole document, the paragraphs, the pages and the sections are the contexts of documents. Hence the contexts of documents are able to be served as the search units for retrieving textual information from a textual database. A hypothesis is then proposed to be the conclusion of 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 be served as the search units for retrieving textual information from a textual database.

3. IMPLEMENTATION OF CONTEXT STRUCTURES

There are two different ways to implement context structures: the implicit representation and the explicit representation. The *implicit representation* is one that the context structure is embedded in the text and is recognized by a program. In an implicit representation, a set of specific context delimiters must be inserted into the text in order to identify each context, such as that discussed in [Emra83]. A context is then surrounded by a pair of beginning and ending delimiters. The text scanner can then be used to find the beginnings and ends of contexts. A higher-level context which contains the current context or a lower-level context which is contained in the current context can also be searched by scanning the text forward and backward. In the implicit representation, the logical structure and the layout structure of a document can be simultaneously represented by using two different sets of context delimiters.

The advantages of the implicit representation are the context structure is relatively simple, it requires no space overhead excepting the space for context delimiters, and it is relatively easy to maintain. However, the time required for scanning a context from the whole database is O(L), where L denotes the size of the whole database. The major disadvantage of the implicit representation is that one must retrieve textual information by

using the access method based on full text scanning [Falo85]. Even a refined string pattern matching method, such as that proposed in [AhCo75] and [BoMo77], can be used to improve the access performance, the full text scanning still consumes too much time for retrieving textual information. Hence it is not reasonable to implement a large textual database by using only the implicit representation method.

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 denotes a context of the context structure. Hence instead of inserting a number of sets of specific context delimiters in the text, the explicit representation uses a set of explicit context trees (or called context trees for short) to reflect the context structures of a text. The data structure of a context tree is proposed in definition-3 and the explicit representation is then proposed in definition-4.

Definition-3: Explicit context tree. An *explicit context tree* is a tree representing a designated context structure of a given text. In the tree, each node uniquely denotes a context of the context structure, and it has a local name and a vector consisting of a pair of pointers, (BP,EP). Where the pointers, BP and EP, point to the beginning position and the ending position of the context denoted by this node, respectively.

Defintion-4: Explicit representation. The *explicit representation* for expressing the context structures of a given text T is defined as a finite set, $(T,H_1,H_2,...,H_k)$, consisting of the text T and k explicit context trees. Where the text T was defined in definition-1 and each explicit context tree H_i, $1 \le i \le k$, representing a designated context structure of the text T, was defined in definition-3.

A textual database is basically an instance of the explicit representation, e.g., as shown as figure 4. In figure 4, the 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 structure and the layout structure are mentioned in section 2. The storage structure represents how the text T is stored in the storage of a particular computer system. In addition, there is an example of the context tree shown in figure 5.

Referred to the characteristics of a tree [Knut73], for each node X of the context tree, there exists a unique search path from the root to node X. The *path name* of node X is then defined as the list which consists of the local names of the nodes along the search path for node X. The number of nodes (or local names) in a path name is referred to as the length of the path name. In the explicit representation, because each context is uniquely denoted by a node of the context tree, the path name of node X is also regarded as the *context-id* of context X. For example, referred to figure 5, the path name of node d_5 is $ab_3\ c_4\ d_5$; thus the context-id of context d_5 is also $ab_3\ c_4\ d_5$. The context-id of a context stands for the search path on the context tree in which the context should be located. That is, by means of the context tree, the beginning and the end of a context, which are indicated by a vector (BP,EP), can be easily searched when its context-id is given. A higher- level context which contains the

current context or a lower-level context which is contained in the current context is also easily searched via the links on the context tree. The time required for searching a context by means of the context tree is O(n), where n denotes the length of the context-id of the context. In general, n is roughly O(log L) and is much smaller than L, where L denotes the length of the whole text of a textual database. Hence, from the viewpoint of search performance, the explicit representation is much better than the implicit one.

The following is an additional property of the context tree :

Prop.6: For any node X of the context tree, except for the root, if the length of its path name is k, k>1, then the path name of node X can be formally presented as the list $N_1\ N_2\ ....N_k$. It is obvious that node X has k−1 ancestors and each of them is denoted by a proper prefix of the path name of node X. Where a proper prefix of the list $N_1\ N_2\ ....N_k$ is defined as a sublist of the form $N_1\ N_2\ ....N_j$, $1 \leq j < k$.

For example, referred to figure 5, node d_5 has three ancestors because the length of its path name, $ab_3\ c_4\ d_5$, is four. The path names of these ancestors are a, ab_3 , and $ab_3\ c_4$.

Theorem-3: Given any context X of a document, except for the document itself. If the length of the context-id of context X is k, then k>1 and there exist k−1 higher-level contexts of the document in which each contains the context X. In addition, each of these higher-level contexts is denoted by a proper prefix of the context-id of context X.

Proof: Because the path name of node X is also the context-id of context X, it is trivial to derive this theorem from prop.2 and prop.6. □

Theorem-4: Given any two contexts, say X and Y, of a document, only one of the followings holds:

(1) Context X is entirely contained in context Y if and only if the context-id of context Y is a proper prefix of the context-id of context X.
(2) Context Y is entirely contained in context X if and only if the context-id of context X is a proper prefix of the context-id of context Y.
(3) Context X and context Y has no common text if and only if the context-id of context X is not a proper prefix of the context-id of context Y and vice versa.

Proof: It is trivial to derive this theorem from the theorem-2 and theorem-3. □

Because any level of contexts can be located by means of the context trees, the explicit representation has the ability to provide a flexible unit for retrieving the textual information. As an example, for a textual database consisting of a set of long-deep documents, one can retrieve the textual information in pages, in paragraphs, or in sections. The explicit representation also provides some important advantages on retrieval mechanism of a textual database. First, a sophisticated retrieval method is easily applied to the explicit representation. There have some such retrieval method mentioned in [Falo85] and [Ozka86], e.g., inversion of terms, surrogates of contexts, etc. Second, by means of specifying a subtree of a context tree, one can reduce the search space and then speeds up the retrieval performance. Third, for example, suppose that the inversion method is applied to the textual database. From theorem-3, each higher-level context which contains a designated context X

is represented by a proper prefix of the context-id of context X. The inverted terms for context X also stand for all higher-level contexts containing context X. Hence the size of inversion table should be reduced.

However, the disadvantages of the explicit representation are: it requires space overhead to store the additional context trees and both the text and the context trees have to be updated in order to maintain a context. Contrarily, in the implicit representation, only the text must be updated for maintaining a context. In the explicit representation, for a context tree, the *maintenance cost* of a context is defined as the number of nodes in which the vector (BP,EP) in each node must be updated when the context is maintained. For any context tree, the maintenance cost of a context is very difficult to compute accurately. But for a nearly balanced context tree, the maintenance cost of a context can be roughly calculated as follows.

A context tree is said to be *nearly balanced* if and only if the context tree satisfies the following properties: (1) each node, except for a leaf node, has a nearly equivalent number of children; and (2) the search path of each leaf node is nearly equivalent in the length. From theorem-5, discussed in the next paragraph, the average maintenance cost of a leaf context on a nearly balanced context tree is about N/2, where N denotes the number of nodes on the context tree. To maintain a context always invokes the maintenances of some leaf contexts contained in the context. Hence it is reasonable to said that the maintenance cost of a context is O(N). For a huge textual database, because the number N is very large, the maintenance cost becomes a heavy burden for updating a context. In the explicit representation, the addressing mechanism used in a context tree is known as the *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 this paper. It will be discussed in section 4.3 that ECHO can reduce the maintenance cost from O(N) to O(log N).

Theorem-5 : 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, it is reasonable to assume that each node of the context tree, except for a leaf node, has an average number, m, of children and the average length of search path of each leaf node 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 root to the level of leaf nodes, are named by $level_1$,$level_2$, ..., $level_n$, respectively. It is obvious that $level_i$, $1 \le i \le n$, has m^{i-1} nodes. The total number of nodes on the context tree is then given by

(e1) $N = \Sigma_{1 \le i \le n} m^{i-1} = (m^n - 1)/(m-1)$.

Suppose that a leaf context X is maintained and the maintenance causes a change in the length of context X. It is obvious that the length of each higher-level context containing context X is then changed and the location of each context behind context X is also moved.

Hence each node on the search path of node X or on a search path in the right side of 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 in 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 an equivalent probability, TC is given by

(e2) $TC = \Sigma_{1\le j\le M} C_j = \Sigma_{1\le j\le M} (\Sigma_{1\le i\le n} c_{ij}) = \Sigma_{1\le i\le n} (\Sigma_{1\le j\le M} c_{ij})$.

Where $M=m^{n-1}$ 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 for maintaining an individual leaf context. For each $level_i$, $1\le i\le n$, there have $L=m^{i-1}$ nodes on this level and each of these nodes has m^{n-i} leaf elements. A node X is said to be a leaf element of node Y if and only if node X is a leaf node and either node X is a descendant of node Y or node X = node Y. As a convention, all nodes on $level_i$, from left to right, are named by $node_{i1}$, $node_{i2}$,....,$node_{iL}$, respectively. From the discussion before (e2), for $level_i$, from $node_{ik}$ to $node_{iL}$ must be updated for maintaining each leaf element of $node_{ik}$. That is, all the leaf elements of $node_{ik}$ have same value of c_{ij} which is L-k+1. Thus the value of $\Sigma_{1\le j\le M} c_{ij}$ can be calculated as following.

(e3) $\Sigma_{1\le j\le M} c_{ij} = \Sigma_{1\le k\le L} m^{n-i}(L-k+1) = (m^{n-i})L(L+1)/2$

From equations (e1), (e2), and (e3), the value of TC is then given by

(e4) $TC = \Sigma_{1\le i\le n} (m^{n-i}L(L+1)/2) = M(N+n)/2 = MN/2, N>>n.$

Last, the average maintenance cost of a leaf context is $TC/M = N/2$. □

4. EXPLICIT CONTEXT-HIERARCHICAL ORGANIZATION

Definition-5: ECHO tree. An *ECHO tree* is a modified context tree in which a relative addressing mechanism is used instead of an absolute addressing mechanism. In an ECHO tree, each node has a local name and a vector (D,L). For the root, the D denotes the beginning position of the root context. For each of other nodes, say X for any, the D denotes the distance between the beginning positions of the contexts denoted by the node X and by the parent of the node X. For each node, the L denotes the length of the context denoted by this node.

As conventions, for each node of an ECHO tree, say X for any, the D-value of the node X is denoted by $D(X)$ and the L-value of the node X is denoted by $L(X)$.

Theorem-6: For each node of an ECHO tree, the value of the vector (D,L), except for D(root), can be computed as follows.

(1) If a node X is a leaf node, then L(X) is actually the length of the leaf context X, otherwise, $L(X)=L(Y_1)+L(Y_2)+....+L(Y_k)$. Where $Y_1, Y_2, ..., Y_k$ denote all the children of the node X.

(2) If a node Y is the leftmost child of its parent, then D(Y) = 0, otherwise, D(Y) = D(Z)+L(Z). Where the node Z is the nearest left sibling of the node Y.

Proof: From definition-5 and prop.5, it is easy to show the part (1) of the theorem. The part (2) is proved as follows. For each non-leaf node X of the ECHO tree, suppose that the node X has k children which are respectively named by $Y_1, Y_2, ..., Y_k$, from the leftmost one to the rightmost one. From definition-2, the formation of a context structure is based on the hierarchical partitioning. It is obvious that the context Y_1 and the context X have the same beginning position. Hence $D(Y_1)=0$. In addition, it is easy to show

(e5) $D(Y_i)=D(Y_{i-1})+L(Y_{i-1})$, $1<i\leq k$.

Where the node Y_{i-1} is the nearest left sibling of the node Y_i. From above, the theorem is proved. □

Definition-6: ECHO structure. The *ECHO structure* for expressing the context structures of a given text T is defined as a finite set, $(T,E_1,E_2,...,E_k)$, consisting of the text T and k ECHO trees E_1, E_2, ..., E_k. Where the text T was defined in definition-1 and each ECHO tree E_i, $1\leq i\leq k$, representing a designated context structure of the text T, was defined in definition-5.

Since the ECHO structure is a modified explicit representation, most of the discussions about the explicit representation are also suitable for the ECHO structure. For example, figure 4 can be also used to depict a textual database , (T,Ec,Et,Es), based on the ECHO structure if the context trees Hc, Ht and Hs are respectively replaced by the ECHO trees Ec, Et and Es. In addition, an example of the ECHO tree which is equivalent to the context tree shown in figure 5 is shown in figure 6. The operations for the ECHO structure can be classified into three types: the search functions, the constructing operations, and the updating operations. They are respectively discussed in sections 4.1, 4.2, and 4.3.

4.1 THE SEARCH FUNCTIONS

There are three basic search functions provided for the ECHO structure $(T,E_1,E_2,...,E_k)$: *get_ ptrs*, *get_ text* and *get_ ids*. They will be defined in definition-7, -8 and -9, respectively.

Definition-7: *get_ ptrs*. Given a context-id $N_1\ N_2\ ...N_m$. The search function *get_ ptrs* computes the beginning position and the ending position of the context $N_1\ N_2\ ...N_m$.

It is inferred from prop.6 and the relevant discussions that each context within the ECHO structure $(T,E_1,E_2,...,E_k)$, say X for any, can be formally represented by a unique context-id, $N_1\ N_2\ ...N_m$, $1\leq m\leq n$, where n is the height of the ECHO tree containing the node X. As a convention, the beginning pointer and the ending pointer of the context X are respectively denoted by $BP(N_1\ N_2\ ...N_k)$ and $EP(N_1\ N_2\ ...N_k)$. In addition, for a node $N_1\ N_2\ ...N_i$, its D-value is denoted as $D(N_1\ N_2\ ...N_i)$ and its L-value is denoted as $L(N_1\ N_2\ ...N_i)$.

Algorithm-1: Performs the search function *get_ ptrs*.
Input: A context-id $N_1\ N_2\ ...N_m$.
Output: $BP(N_1\ N_2\ ...N_m)$ and $EP(N_1\ N_2\ ...N_m)$.
Precedure:

s1: P = $D(N_1)$;
s2: for i := 2 to m do BP := BP + $D(N_1 N_2 ... N_i)$;
s3: EP := BP + $L(N_1 N_2 ... N_m) - 1$;
s4: return BP and EP as the result;

Theorem-7: Algorithm-1 correctly performs the function *get_ptrs*.
Proof: From definition-5, it is easy to derive the equations (e6) and (e7).

$$\text{(e6) } BP(N_1 N_2 ... N_m) = \begin{cases} D(N_1), & m=1 \\ BP(N_1 N_2 ... N_{m-1}) + D(N_1 N_2 ... N_m), & m>1 \end{cases}$$

$$\text{(e7) } EP(N_1 N_2 ... N_m) = BP(N_1 N_2 ... N_m) + L(N_1 N_2 ... N_m) - 1$$

The equation (e8) is then transformed form (e6).

$$\text{(e8) } BP(N_1 N_2 ... N_m) = \Sigma_{1 \le i \le m} D(N_1 ... N_i)$$

From s1 and s2, it is easy to find that the value of BP is the same as that shown in (e8). And from s3 the value of EP is the same as that shown in (e7). Hence the theorem is proved. □

Now let us take context d7 shown in figure 6 as an example. Its context-id is $ab_3 c_5 d_7$, thus we have

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

The result (569,749) is the same as the value of vector (BP,EP) on context d_7 as shown in figure 5.

Definition-8: *get_text*. Given a context-id $N_1 N_2 ... N_m$. The search function *get_text* read the context $N_1 N_2 ... N_m$ from the text T.

Algorithm-2: Performs the search function *get_text*.
Input: A context-id $N_1 N_2 ... N_m$.
Output: The text of context $N_1 N_2 ... N_m$.
Procedure:
s1: Apply the search function *get_ptrs* to get the pointers $BP(N_1 N_2 ... N_m)$ and the $EP(N_1 N_2 ... N_m)$;
s2: to read a subtext from the text T from the position $BP(N_1 N_2 ... N_m)$ to the position $EP(N_1 N_2 ... N_m)$;
s3: return the subtext as the result;

Theorem-8: Algorithm-2 correctly performs the function *get_text*.
Proof: From theorem-7, it is easy to show this theorem. □

Definition-9: *get_ids*. Given the beginning position and the ending position of a subtext S, respectively denoted by BP(S) and EP(S). The search function *get_ids* finds a sequence of contexts of the level m by means of a designated ECHO tree such that these contexts contain the subtext S.

Algorithm-3: Find a context $N_1\ N_2\ ...N_m$ in which a given position P is located.
Input: A position P, an integer m specifying the length of context-id, and a node name N_1 which is the root of the designated ECHO tree.
Output: A context-id $N_1\ N_2\ ...N_m$ satisfying the condition
$BP(N_1\ N_2\ ...N_m) \leq P \leq EP(N_1\ N_2\ ...N_m)$.
Procedure:
s1: if $P \geq D(N_1)$ and $L(N_1) \geq P - D(N_1) + 1$ then
begin
s2: if m = 1 then return the root as result;
s3: if m > 1 then
begin
s4: for i := 1 to m − 1 do
begin
s5: $P := P - D(N_1\ N_2\ ...N_i)$;
s6: sequentially search the children of node $N_1\ ...N_i$ to find a node $N_1\ ...N_{i+1}$ such that $D(N_1\ ...N_{i+1}) \leq P$ and $L(N_1\ ...N_{i+1}) \leq P - D(N_1\ ...N_{i+1}) + 1$;
end;
s7: return the context-id $N_1\ N_2\ ...N_m$ as the result;
end;
end;

Algorithm-4: Performs the search function *get_ids*.
Input: The beginning position BP(S) and the ending position EP(S) of a subtext S, an integer m specifying the length of context-ids, and a node name N_1 which is the root of the designated ECHO tree.
Output: A sequence of context-ids denoting the contexts which contain the given subtext S.
Procedure:
s1: apply algorithm-3 to find a context-id $N_1\ A_2\ ...A_m$ such that $BP(N_1\ A_2\ ...A_m) \leq BP(S) \leq EP(N_1\ A_2\ ...A_m)$;
s2: apply algorithm-3 to find a context-id $N_1\ B_2\ ...B_k$ such that $BP(N_1\ B_2\ ...B_m) \leq EP(S) \leq EP(N_1\ B_2\ ...B_m)$;
s3: search the ECHO tree to find all context-ids of the length m which are in the range from $N_1\ A_2\ ...A_m$ to $N_1\ B_2\ ...B_m$;
s4: return all of the context-ids from $N_1\ A_2\ ...A_m$ to $N_1\ B_2\ ...B_m$ as the result;

Theorem-9: Algorithm-4 correctly performs the function *get_ids*.
Proof: The proof can be divided into two parts. We first prove that algorithm-3 is correct. Then we prove that algorithm-4 is correct.
Part 1: A position P is said to be located in a context $N_1\ N_2\ ...N_k$ if and only if
(e9) $BP(N_1\ N_2\ ...N_m) \leq P \leq EP(N_1\ N_2\ ...N_m)$
By applying the equations (e6) and (e7), the inequality (e9) can be transfered as the inequality (e10).
(e10) $D(N_1\ N_2\ ...N_m) \leq P - \Sigma_{1 \leq i \leq m-1} D(N_1\ N_2\ ...N_i)$ and
$L(N_1\ N_2\ ...N_m) \geq P - \Sigma_{1 \leq i \leq m} D(N_1\ N_2\ ...N_i) + 1$

In algorithm-3, s1 prevents the case that the given position P is out of the text T. From s2 through s6, it is clear that the inequality (e10) is correctly performed. Hence algorithm-3 correctly performs the inequality (e9). That is, algorithm-3 can be used to correctly find a context in which the given position P is located.

Part 2: From s1 and s2 of algorithm-4, we have

(e11) $BP(A_1\ A_2\ ...A_m) \leq BP(S) \leq EP(S) \leq EP(B_1\ B_2\ ...B_m)$.

Thus from s3 and s4, it is clear that the given subtext S is contained in the sequence of contexts from $N_1\ A_2\ ...A_m$ to $N_1\ B_2\ ...B_n$. Hence the theorem is proved. □

4.2 THE CONSTRUCTING OPERATIONS

There are two basic constructing operations provided for the ECHO structure (T,E): *tree-constructing operation* and *echomerge*. They will be defined in definition-10 and -11, respectively.

Definition-10: Tree-constructing operation. The *tree-constructing operation* is a series of processes to construct an ECHO tree from a fully marked-up text.

In order to construct a context tree or an ECHO tree from a given text, the context structure of the text has to be previously marked up. That is, a set of markup tokens must be previously inserted into the text to identify each context of the text [Hsie88], [Tsen88] and [TsYH88]. The discussions of the markups can be found in [CoRD87], [Hsie88], [ISO8879] and [PeNJ85]. As a fully marked-up text is available, an ECHO tree of the text can be constructed by the following steps.

(1) A context parser scans the text to recognize each markup token and to construct an intermediate ECHO tree which reflects the context structure of the text. An intermediate ECHO tree differs from an ECHO tree in two things. First, for each leaf node, the L-value is given but the D-value is undefined. Second, for each other node, both the D-value and the L-value are undefined. The context parser is too complicated to be discussed in this paper, and it was discussed in [Hsie88] in detail.

(2) For the intermediate ECHO tree, to compute the D-value and L-value of each node by using the method proposed in theorem-6.

Definition-11: *echomerge*. Given a number of j ECHO structures (T_1, E_1), (T_2, E_2), ..., (T_j, E_j). The operation *echomerge* combines these individual ECHO structures to form a larger ECHO structure (T,E). Each of the given ECHO structure (T_i, E_i), $1 \leq i \leq j$, then becomes a substructure of the ECHO structure (T,E). That means each ECHO tree Ei becomes a subtree of the ECHO tree E.

From the conventions mentioned in section 2, a context tree is named by its root. Thus the root of an ECHO tree X is simply called the node X. An ECHO structure can be formed by combining a number of smaller ECHO structures. Such a combination invokes both the

concatenation of their texts and the combination of their ECHO trees, as mentioned in algorithm-5.

Algorithm-5: Performs the operation *echomerge*.
Input: j ECHO structures (T_1, E_1), (T_2, E_2), ..., (T_j, E_j).
Output: A combined ECHO structure (T,E)
Procedure:
s1: T := null;
s2: create a new node E;
s3: for i := 1 to j do
    begin
  s4: append(T,T_i);
  s5: link the node E_i to the node E such that E_i becomes the rightmost child of E;
    end;
s6: D(E) := the beginning position of T;
s7: $D(E_1) := 0$;
s8: for i := 2 to j do $D(E_i) := D(E_{i-1}) + L(E_{i-1})$;
s9: $L(E) := D(E_j) + L(E_j)$;

Theorem-10: Algorithm-5 correctly performs the operation *combine*.
Proof: From theorem-6, it is easy to show this theorem. □

4.3 THE UPDATING OPERATIONS

There are three basic updating operations provided for the ECHO structure (T,E): *echoinsert*, *echodelete* and *leafmodify*. They will be respectively defined in definition-12, -13, and -14.

Definition-12: *echoinsert*. Given an ECHO structure (T_x, E_x) and a parameter which is either $N_1\ N_2\ ...N_m$:l or $N_1\ N_2\ ...N_m$:r. The *echoinsert* performs either of the following operations, depending upon which parameter is given:
(1) If a parameter $N_1\ N_2\ ...N_m$:l is given, the text T_x is inserted into the text T in the position that immediately precedes the context $N_1\ N_2\ ...N_m$ and the node E_x is linked to the node $N_1\ N_2\ ...N_{m-1}$ such that the node E_x becomes the nearest left sibling of the node $N_1\ N_2\ ...N_m$.
(2) If a parameter $N_1\ N_2\ ...N_m$:r is given, the text T_x is inserted into the text T in the position that immediately follows the context $N_1\ N_2\ ...N_m$ and the node E_x is linked to the node $N_1\ N_2\ ...N_{m-1}$ such that the node E_x becomes the nearest right sibling of the node $N_1\ N_2\ ...N_m$.

From prop.2 and prop.6, the node $N_1\ N_2\ ..N_{m-1}$ is the parent of the node $N_1\ N_2\ ..N_m$. A node X is said to be the nearest left (or right) sibling of the node Y if and only if the node

X has the same parent of the node Y and the node X is in the position that immediately precedes (or follows) the node Y.

Algorithm-6: Performs the operation *echoinsert*.
Input: An ECHO structure (T_x, E_x) and a parameter which is either $N_1\ N_2\ ...N_m$:l or $N_1\ N_2\ ...N_m$:r.
Output: The ECHO structure (T,E) which the ECHO structure (T_x, E_x) was inserted into.
Procedure:
s1: apply the search function *get_ptrs* to get the pointers $BP(N_1\ N_2\ ...N_m)$ and $EP(N_1\ N_2\ ...N_m)$;
s2: if the given parameter is $N_1\ N_2\ ...N_m$:l then
begin
s3: insert the text T_x into the text T in the position $BP(N_1\ N_2\ ...N_m)-1$;
s4: link the node E_x to the node $N_1\ N_2\ ...N_{m-1}$ such that the node E_x becomes the nearest left sibling of the node $N_1\ N_2\ ...N_m$;
end;
s5: if the given parameter is $N_1\ N_2\ ...N_m$:r then
begin
s6: insert the text T_x into the text T in the position $EP(N_1\ N_2\ ..N_m)+1$;
s7: link the node E_x to the node $N_1\ N_2\ ...N_{m-1}$ such that the node E_x becomes the nearest right sibling of the node $N_1\ N_2\ ...N_m$;
end;
s8: if the node E_x is the leftmost child of the node $N_1\ N_2\ ...N_{m-1}$
s9: then $D(E_x)=0$
s10: else $D(E_x)=D(Z)+L(Z)$, where Z denotes the nearest left sibling of the node E_x;
s11: for each right sibling of the node E_x, called node W,
do $D(W)=D(W)+L(E_x)$;
s12: for i=m-1 downto 1 do
begin
s13: $L(N_1\ ...N_i)=L(N_1\ ...N_i)+L(E_x)$;
s14: for each right sibling of the node $N_1\ ...N_i$, called node U,
do $D(U)=D(U)+L(E_x)$;
end;

Theorem-11: Algorithm-6 correctly performs the operation *echoinsert*.
Proof: From s1 through s7, it is clear that the ECHO structure (T_x, E_x) is inserted into a proper position of the ECHO structure (T,E). From definition-5, theorem-6 and theorem-7, it is obvious that after the text T_x is inserted into the text T, the following situations occur:
(1) The $BP(E_x)$ is changed. That is the $D(E_x)$ is changed.
(2) The beginning positions of all the right siblings of the node E_x are moved right by $L(E_x)$ from the original positions. That is, the D-values of all the right siblings of the node E_x must be added by $L(E_x)$.

(3) The lengths of all the contexts containing the text T_x are increased by $L(E_x)$. That is, all the L-values of the ancestors of the node E_x are increased by $L(E_x)$.
(4) Follows (3), the beginning positions of all the right siblings of each ancestor of the node E_x are moved right by $L(E_x)$ from the original positions. That is, the D-values of all the right siblings of each ancestor of the node E_x have to be added by $L(E_x)$.

Referred to theorem-6, it is easy to show that the $D(E_x)$ is correctly computed by s8, s9, and s10. In addition, the sitution (2) is performed by s11 and the situtions (3) and (4) are performed by s12, s13 and s14. Hence the theorem is proved. □

Definition-13: *echodelete*. Given a context-id $N_1\ N_2\ ...N_m$. The operation *echodelete* removes the context $N_1\ N_2\ ...N_m$ from the text T and removes the subtree $N_1\ N_2\ ...N_m$ from the ECHO tree E.

Algorithm-7: Performs the operation *echodelete.*
Input: A context-id $N_1\ N_2\ ...N_m$.
Output: The ECHO tree (T,E) in which the context $N_1\ N_2\ ...N_m$ is removed from the text T and the subtree $N_1\ N_2\ ...N_m$ is removed from the tree E.
Procedure:
s1: apply the search function *get_ptrs* to get the pointers $BP(N_1\ N_2\ ...N_m)$ and $EP(N_1\ N_2\ ...N_m)$;
s2: remove a subtext from the position $BP(N_1\ N_2\ ...N_m)$ to the position $EP(N_1\ N_2\ ...N_m)$ from the text T;
s3: for each right sibling of the node $N_1\ N_2\ ...N_m$, called node W,
do $D(W)=D(W)-L(N_1\ N_2\ ...N_m)$;
s4: for i=m-1 downto 1 do
begin
s5: $L(N_1\ ...N_i)=L(N_1\ ...N_i)-L(N_1\ N_2\ ...N_m)$;
s6: for each right sibling of the node $N_1\ ...N_i$, called node U, do $D(U)=D(U)-L(N_1\ N_2\ ...N_m)$
end;
s7: disconnect the link between the nodes $N_1\ N_2\ ...N_{m-1}$ and $N_1\ N_2\ ...N_m$;

Theorem-12: Algorithm-7 correctly performs the operation *echodelete.*
Proof: From s1 and s2, it is clear that the context $N_1\ N_2\ ...N_m$ is correctly removed from the text T. And from s7, it is clear that the subtree $N_1\ N_2\ ...N_m$ is correctly removed from the tree E. From definition-5, theorem-6 and theorem-7, it is obvious that after the context $N_1\ N_2\ ...N_m$ is removed from the text T, the following situations occur:
(1) The beginning positions of all the right siblings of the node $N_1\ N_2\ ...N_m$ are moved left by $L(N_1\ N_2\ ...N_m)$ from the original positions. That is, the D-values of all the right siblings of the node $N_1\ N_2\ ...N_m$ must be decreased by $L(N_1\ N_2\ ...N_m)$.
(2) The lengths of all the contexts containing the context $N_1\ N_2\ ...N_m$ are decreased by $L(N_1\ N_2\ ...N_m)$. That is, the L-values of all the ancestors of the node $N_1\ N_2\ ...N_m$ are decreased by $L(N_1\ N_2\ ...N_m)$.

(3) Follows (2), the beginning positions of all the right siblings of each ancestor of the node $N_1\ N_2\ ...N_m$ are moved left by $L(N_1\ N_2\ ...N_m)$) from the original positions. That is, the D-values of all the right siblings of each ancestor of the node $N_1\ N_2\ ...N_m$ have to be decreased by $L(N_1\ N_2\ ...N_m)$.

It is clear that the sitution (1) is performed by s3 and the situtions (2) and (3) are performed by s4, s5, and s6. Hence the theorem is proved. □

From the characteristics of the context structure, it is obvious that the modification of a context is actually realized by modifying some leaf contexts. As a convention, the modified version of a leaf context X is called *context X'*. The modification of the leaf context X may causes a change in the L(X), i.e., $L(X') \neq L(X)$. The ECHO tree E has to be updated by using the operation *leafmodify* if $L(X') \neq L(X)$.

Definition-14: *leafmodify*. Given a leaf context X and its modified version X'. The operation *leafmodify* replaces the leaf context X by the modified version X'. In addition, the operation *leafmodify* updates the (D,L) vectors of the relevant nodes if $L(X') \neq L(X)$.

Algorithm-8: Performs the operation *leafmodify*.
Input: A modified leaf context X' and its context-id $N_1\ N_2\ ...N_n$.
Output: The ECHO structure (T,E) in which the leaf context $N_1\ N_2\ ...N_n$ is modified.
Procedure:
s1: apply the search function *get_ptrs* to get the pointers $BP(N_1\ N_2\ ...N_n)$ and $EP(N_1\ N_2\ ...N_n)$;
s2: remove a subtext from the position $BP(N_1\ N_2\ ...N_n)$ to the position $EP(N_1\ N_2\ ...N_n)$ from the text T;
s3: insert the context X' into the text T in the position $BP(N_1\ N_2\ ...N_n) - 1$;
s4: computes L(X');
s5: if $L(X') \neq L(N_1\ N_2\ ...N_n)$ then
begin
s6: for each right sibling of the node $N_1\ N_2\ ...N_n$, called node W,
do $D(W) = D(W) - L(N_1\ N_2\ ...N_n) + L(X')$;
s7: for i=n-1 downto 1 do
begin
s8: $L(N_1\ ...N_i) = L(N_1\ ...N_i) - L(N_1\ N_2\ ...N_n) + L(X')$;
s9: for each right sibling of the node $N_1\ ...N_i$, called node U,
do $D(U) = D(U) - L(N_1\ N_2\ ...N_n) + L(X')$;
end;
end;

Theorem-13: Algorithm-8 correctly performs the operation *leafmodify*.
Proof: Referred to theorem-6, theorem-7 and the discussion before definition-14, it is easy to prove this theorem. □

Theorem-14: For a nearly balanced ECHO structure (T,E), the maintenance cost of a context is O(log N), where N denotes the number of node on the ECHO tree E.

Proof: We first consider the cases of insertion and deletion. From theorem-11 and theorem-12, the maintenance cost, MC_1, of a context $N_1 N_2 ..N_m$ is given by

(e12) $MC_1 = m + \Sigma_{2 \le i \le m} K_i$, $m \le n$.

Where K_i denotes the number of right siblings of the node $N_1 N_2 ...N_i$ and n denotes the height of the ECHO tree E. Since the ECHO tree is nearly balance, it is reasonable to assume that the average number of children of each non-leaf node is K and $n = \log N$. Thus we have

(e13) $m \le MC_1 \le 1 + K(n-1) < K(\log N)$.

It is clear that $MC_1 = O(\log N)$.

Then consider the case that a leaf context $N_1 N_2 ...N_n$ is modified. From theorem-13, the maintenance cost of this case, MC_2, is given by

(e14) $MC_2 = n + \Sigma_{2 \le i \le n} K_i$.

From the assumptions above, we have

(e15) $n = (\log N) \le MC_2 \le 1 + K(n-1) < K(\log N)$

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

4.4 THE ECHO SUBSYSTEM

The ECHO subsystem shown in figure 1 is formally an ECHO structure (T,E_1 , E_2 ,...,E_k). In the ECHO structure, the text T is formed by concatenating the texts of all documents and each ECHO tree E_i , $1 \le i \le k$, represents a kind of context structure of the database. In addition, the ECHO subsystem provides three basic search functions as mentioned in section 4.1. But the constructing operations and the updating operations are rather provided by the maintenance subsystem than provided by the ECHO subsystem.

The constructing operations discussed in section 4.2 can provide a useful basis to build the ECHO structure of a textual database. Recall from section 2 that a textual database is constituted by a set of documents of a same class. Because a textual database is always very large, it is reasonable to build the ECHO structure by applying two processes, first *dividing* and then *combining*, as mentioned below.

Dividing. All the source texts of a textual database are first divided into a number of elementary segments such that each elementary segment forms an individual context and the length of it is suitable for processing. The suitable length of an elementary segment depends on the constraints given by the text processor and the context parser. In practice, for long-deep documents, a suitable elementary segment is probably a document or a part of document, e.g., a chapter. For short-shallow documents, a suitable elementary segment is probably a document or a group of documents. For each elementary segment, the context structure is then marked-up by using the text processor. And then the ECHO tree is constructed by using the tree-constructing operation.

Combining. In general, an elementary segment is much smaller than the whole text T, therefore the ECHO structure cannot be built by directly combining all the elementary segments. Hence multiple combinations of contexts are necessary for building a textual database. That is, a set of relevant elementary segments are first combined to form a larger context by applying the operation *combine*. Then a number of those larger contexts are combined to form a more larger context, and so on, until the textual database is formed.

The discussions above only deal with the case (T,E) that a textual database has only one context structure. In practice, a textual database usually has more than one context structure, e.g., as shown in figure 4. For the latter case, suppose that the database, denoted ($T,E_1 ,E_2 ,...,E_k$), has k, $k>1$, context structures. The necessary constraints of this case are:

(1) Each constituent segment has exactly k ECHO trees in which each represents a kind of context structure.
(2) Only ECHO trees representing the same kind of context structures are able to be combined.

Thus for each elementary segment, it is necessary to apply the tree-constructing operation k times to form k ECHO trees. And then a refined combining operation has to be applied many times to form the final ECHO structure ($T,E_1 ,E_2 ,...,E_k$). The refined combining operation differs from the operation *combine* in replacing each (T_i ,E_i), $1 \leq i \leq j$, refferred to definition-11, by ($T_i ,E_{1i} ,E_{2i} ,...,E_{ki}$).

5. THE TEXT RETRIEVAL SUBSYSTEM

Referred to figure 1, the text retrieval subsystem plays the role of accelerating the performance for retrieving contexts from the ECHO subsystem. The retrieval methods for text, as discussed in [Falo85] and [Ozka86], can be classified into four categories: full text scanning, inversion of terms, surrogates of contexts and clustering. For a textual database, these methods, except for the full text scanning, can be applied to improve the retrieval performance. In the paper, a text retrieval subsystem based on the ARCIM, an inversion method, will be introduced as an example.

A simple inversion method uses an index table in which each entry consists of a term along with a list of pointers, called *posting list*. These pointers point to the contexts containing this term. In an English textual database, a term is actually a word. The advantages of the word inversion are that it is relatively easy to implement, is fast, and provides synonyms easily. Hence the word inversion has been adopted in most commercial systems, such as BRS, DIALOG, MEDIARS, ORBIT and STAIRS [SaMc83]. But the disadvantages are: (1) the storage overhead (50-300% of the size of text files [Hask81]), (2) the cost of updating and reorganizing the index table, if the environment is dynamic, and (3) the cost of merging the posting lists, if they are too long or too many.

However, the word inversion is not suitable for Chinese text. The reasons are of follows. First, a Chinese character string contains no nature delimiters, such as blanks in an English sentence, to separate Chinese words. Second, an automatic and effective word segmentation method for Chinese sentences has not been developed. Thus a character inversion instead of the word inversion is used to access Chinese text. In the paper, a refined character inversion method (abbr. *ARCIM*) will be introduced. It can provide a faster access speed than a simple character inversion method.

Similar to a simple inversion method, the ARCIM uses a character index table, called CIT, and a posting lists table, called PLT, as shown in figure 7. An entry of the PLT, denoted PL(C_i), is a posting list consisting of a variable number of sorted leaf context-ids. Each leaf context-id of the PL(C_i), say X for any, stands for the leaf context X containing the character C_i . In the CIT, each entry consists of a count and a pointer, denoted count(C_i) and pointer(C_i), respectively. The pointer(C_i) points to the starting location of the PL(C_i) and the count(C_i) denotes the number of leaf context-ids of the PL(C_i). In a Chinese computer system, the coding space of Chinese characters is fixed [Tsen86]. Hence the CIT can be organized in a constant size and may be permanently located in the main memory. If a character C_i is given, the count(C_i) and the pointer(C_i) can be accessed by means of a specific hash function which depends upon the coding scheme of the Chinese characters. An example of the hash function can be found from [Tsen82]. In addition, if there exists a character C_j which does not appear in any context, the count(C_j) and the pointer(C_j) will be respectively set to zero and nil and the PL(C_j) does not appear in the PLT.

There are three types of operations provided for the ARCIM: the search operations, the creation operations and the maintenance operations. The search operations will be discussed in section 6. The creation operations and the maintenance operations are respectively mentioned in sections 5.1 and 5.2.

5.1 THE CREATION OPERATIONS FOR ARCIM

Definition-15: ARCIM structure. An *ARCIM structure* is defined as a pair of CIT and PLT, denoted (CIT,PLT). The data structures of the CIT and the PLT and the relationship between the CIT and the PLT have been mentioned above.

For an ECHO structure (T,E_1 ,E_2 ,...,E_k), the text retrieval subsystem is composed of a set of ARCIM structures, denoted ((CIT_1 ,PLT_1), (CIT_2 ,PLT_2), ..., (CIT_k ,PLT_k)), in which each (CIT_j ,PLT_j), $1 \le j \le k$, corresponds to an ECHO tree E_j . Algorithm-9 can be used to create an ARCIM structure (CIT_j ,PLT_j) from both the text T and the designated ECHO tree E_j .

Algorithm-9: Create an ARCIM structure (CIT_j ,PLT_j) from a given sub-ECHO structure (T,E_j).

Procedure:

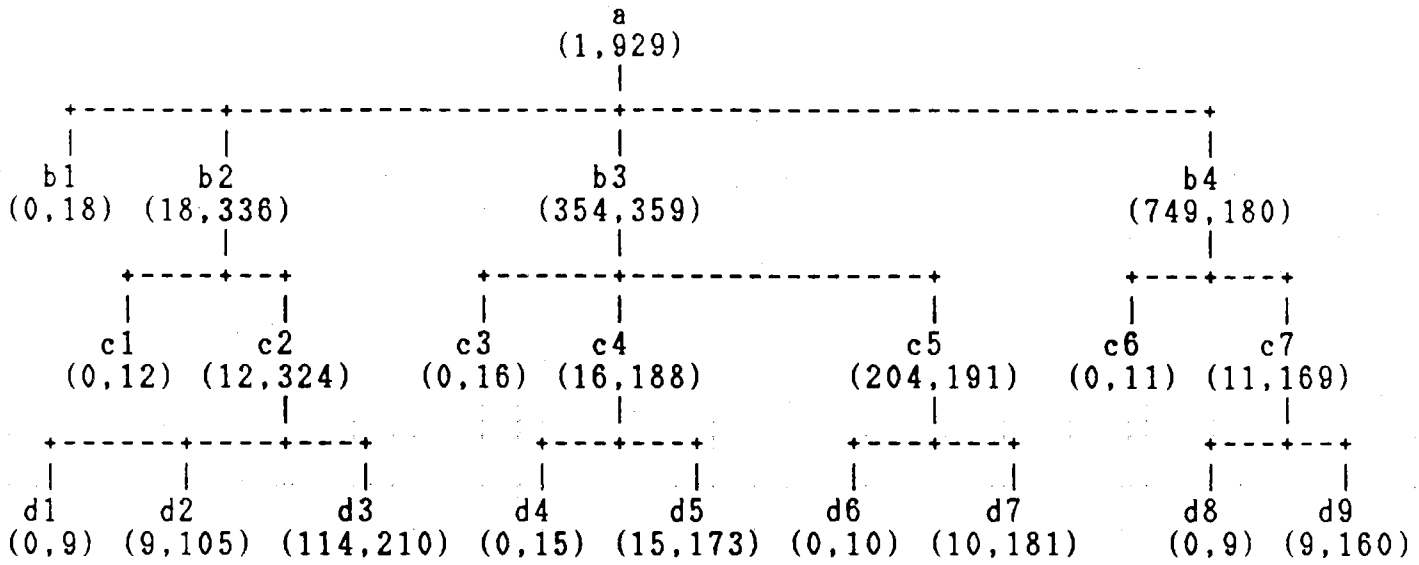

Figure 6 The ECHO tree corresponding to the context tree in Figure 5

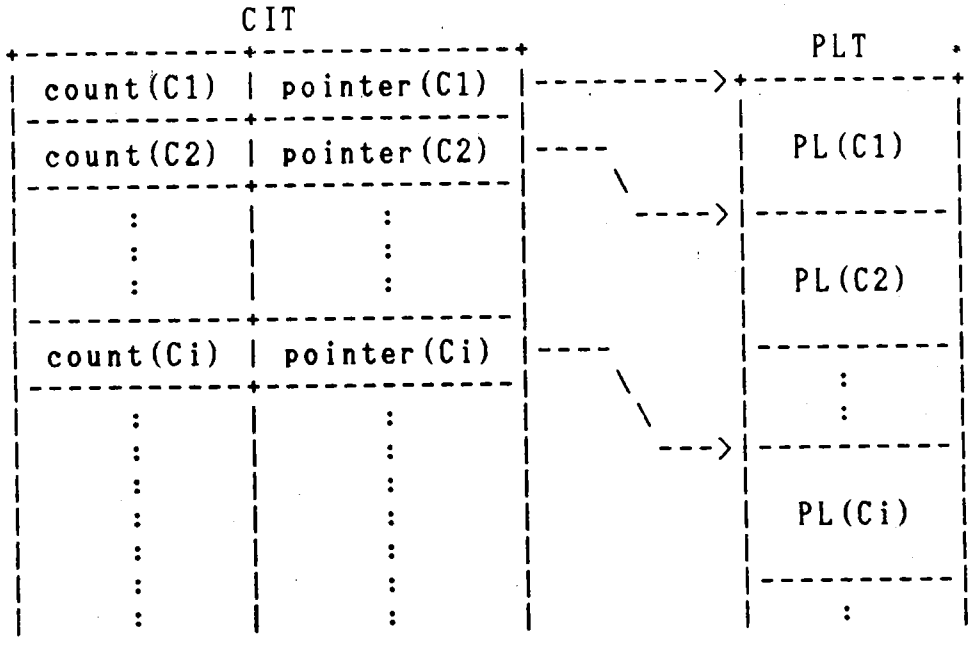

Figure 7 The index organization of the ARCIM

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. Then for each entry, let count(C_i)=0 and let pointer(C_i)=nil. And create a new PLT which contains nothing.

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

(1) Eliminate all blanks, punctuations, and other unnecessary symbols from the leaf context X. The result is called the string X'.

(2) Eliminate all repeated characters from the string X'. The result is called the clist(X).

(3) Append the leaf context-id X to each character C_i of the clist(X). The result is call the cidlist(X). In the cidlist(X), each entry is a pair of a character C_i 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, concatenate all the cidlists to form the cidtable.

Step-4: Sort the cidtable by a 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 Ci 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 by an 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_i), they are respectively the pointer(C_i) and the count(C_i).

(2) Write both the count(C_i) and the pointer(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.

Because Algorithm-9 is too complicated, it becomes very difficult to prove that Algorithm-9 is correct. But it can be verified by testing. For example, Algorithm-9 had been successfully applied to create an ARCIM structure of the CED which is an experimental Chinese electronic dictionary [Tsen88]. In the CED, the space overhead for storing the ARCIM structure, 1.9 Mbytes, is about 30% to the size of the text files, 6.2 Mbytes. The work to create ARCIM structures is once for all. Thus even the creation of an ARCIM structure must consume a lot of computing hours, it is still endurable.

5.2 THE MAINTENANCE OPERATIONS FOR ARCIM

It is obvious that an ARCIM structure has to be maintained when a context is inserted, deleted, or modified. The basic maintenance operations for the cases of insertion, deletion and modification will be respectively mentioned in algorithm-10, -11 and -12.

Algorithm-10: Update the ARCIM structure (CIT,PLT) when a leaf context X is inserted into the ECHO structure.

Procedure:

Step-1: Construct the clist(X) by means of the step-2 of the Algorithm-9. Then sort the clist by an ascending order of the character codes. Suppose that the characters of the clist are $C_1\ C_2 ... C_j$.

Step-2: Insert the context-id X into each PL(C_i), $1 \le i \le j$, keeping the ascending order of context-ids of each PL(C_i).

Step-3: Scan the CIT from the entry for C_1 to the last. Suppose that the entry currently scanned is for a character A.

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

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

(3) For each A, $A > C_j$, let pointer(A):=pointer(A)+j.

Algorithm-11: Update the ARCIM structure (CIT,PLT) when a leaf context X is deleted from the ECHO structure.

Procedure:

Step-1: Construct the clist(X) by means of the step-2 of the Algorithm-9. Then sort the clist by an ascending order of the character codes. Suppose that the characters of the clist are $C_1\ C_2 ... C_j$.

Step-2: Delete the context-id X from each PL(C_i), $1 \le i \le j$.

Step-3: Scan the CIT from the entry for C_1 to the last. Suppose that the entry currently scanned is for a character A.

(1) If $A = C_i$, $1 \le i \le j$, then let count(A):=count(A)−1.

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

(3) For each A, $A > C_j$, let pointer(A):=pointer(A)−j.

Algorithm-12: Update the ARCIM structure (CIT,PLT) when a leaf context X is modified.

Procedure:

Step-1: Construct the old clist(X), called clist(X), and the new clist(X), called clist(X'), by means of the step-2 of the algorithm-9, respectively. Then respectively sort these two clists by an ascending order of the character codes.

Step-2: Compare the clist(X) with the clist(X') to obtain a string $B_1\ B_2 ... B_j$ =clist(X')−clist(X) and a string $D_1\ D_2 ... D_m$ =clist(X)−clist(X'). Where the symbol − denotes a set difference operation. It is clear that the string $B_1\ B_2 ... B_j$ denotes the characters which are added to the leaf context X, while the string $D_1\ D_2 ... D_m$ denotes the characters which are removed from the leaf context X.

Step-3: Apply the step-2 and -3 of algorithm-10 to the string $B_1\ B_2 ... B_j$.

Step-4: Apply the step-2 and -3 of algorithm-11 to the string $D_1\ D_2 ... D_m$.

6. QUERY PROCESSING

The actions that a user queries information from a textual database can be roughly summarized as follows. He selects first a set of terms (keywords) which stand for his topic of interest. Then he queries the database to find a set of contexts containing all or some of the

terms. In general, the terms are conjoined by some operators, as mentioned in [Holl79]. In our system, a query expression has the following form:

(e16) FIND *context-clause*
CONTAIN *search-clause*
scope-clause;

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

(e17) *context-clause* ::= LEAF CONTEXTS | CONTEXTS OF LENGTH k

Where 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 has the context-id of length k.

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

(e18) *search-clause* ::= *search-phrase* {OR *search-phrase*}
(e19) *search-phrase* ::= *term* {AND [NOT] *term*}
(e20) *term* ::= *string* | *wild-card-term* | *ordered-term*

Where the braces {...} denote a repetition of any times and the brackets [...] denote an optional item. The operations for a search clause will be discussed in detail in sections 6.1 and 6.2.

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

(e21) *scope-clause* ::= UNDER *context-id* | FROM $context\text{-}id_1$ TO $context\text{-}id_2$ |
FROM SETS *file-name* {,*file-name*}

A textual database may has more than one context structure, as shown in figure 4. For the case, the phrase "UNDER *context-id*" is necessary in order to specify a designated context structure or a designated sub-context structure in which the retrieved contexts are contained. For example, referred to figure 4, suppose that the query expression

(e22) FIND LEAF CONTEXTS
CONTAIN "textual data?base" OR "information retriev* "
UNDER Ec;

is given. A paragraph, i.e., a leaf context of the logical structure Ec, containing "textual database", "textual data base", "information retrieve", "information retrieval" or "information retrieving" will be retrieved. The phrase "FROM $context\text{-}id_1$ TO $context\text{-}id_2$ " specifies a subset of contexts of a context structure E_i, 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 in the context structure E_i, the context-id_1 must precede the context-id_2. Sometimes, a user needs to search the contexts from some of the results of previous queries or from a specific set of contexts such as the titles of the documents. The FROM SETS phrase is provided for these purposes. A set is actually a file consisting of a number of sorted context-ids. The constraint

of the FORM SETS phrase is that all context-ids of the given files must belong to the same context structure.

The query processor invokes three phases for evaluating a given query expression, namely the consistence check, the search process and the post process. When a query expression is given, the query processor first checks whether the scope clause is valid. There are three cases must be considered.

(1) If a phrase "UNDER N_1 ...N_k " is given, the function *get-ptrs* will be applied to confirm that the N_1 ...N_k is a valid context-id.

(2) If a phrase "FROM A_1 ...A_j TO B_1 ...B_k " is given, the query processor first confirms that the context-ids A_1 ...A_j and B_1 ...B_k are of the same ECHO tree, i.e., $A_1 = B_1$. Then the function *get-ptrs* is applied to confirm that both the A_1 ...A_j and the B_1 ...B_k are valid context-ids and the A_1 ...A_j precedes the B_1 ...B_k. The context A_1 ...A_j is said to precede the context B_1 ...B_k if and only if $EP(A_1 ...A_j) < BP(B_1 ...B_k)$.

(3) Assume that a phrase "FROM SETS $file_1$, ..., $file_k$ " is given. The query processor OR-merges these files to form a merged file SET and at the same time it checks that each context-id of the files has the same first local name. The operation OR-merge has been mentioned in detail in [Emra83].

If the consistence check is failed, then the query expression will be rejected.

6.1 THE SEARCH PROCESS

For a textual database, the actions of the query processor are dependent on the retrieval method applied in the database. The retrieval method used in our sysetm is the ARCIM. By using the ARCIM structure, a given search clause can be evaluated by a search process to obtain a set of phrase-level posting lists in which each leaf context-id probably satisfies the search phrase. Then these phrase-level posting lists have to be processed and OR-merged by a post process to form a clause-level posting list as the result. In order to provide a better retrieval performance, a bottom-up and greedy method is involved in both the search process and the post process. The operations of the search process are mentioned in algorithm-13 and -14. In addition, the post process will be mentioned in section 6.2.

Algorithm-13: Evaluate a given search clause to obtain a set of phrase-level posting lists.
Procedure:
Step-1: Partition the given search clause into a set of search phrases.
Step-2: Reconstruct each search phrase by using algorithm-14.
Step-3: For each reconstructed search phrase, say $B_1\ B_2$ ...B_k for any, perform the following operations to obtain a phrase-level posting list.
(1) If $count(B_1) = 0$, then return an empty posting list as the result.
(2) Otherwise, AND-merge $PL(B_1)$ and $PL(B_2)$ to form $PL(B_1\ B_2)$, then AND-merge $PL(B_1\ B_2)$ and $PL(B_3)$ to form $PL(B_1\ B_2\ B_3)$, and so on, until either an empty posting list is obtained or $PL(B_1\ B_2$ ...$B_k)$ is formed. Where $PL(B_1\ B_2$ ...$B_i)$, $1 \le i \le k$, denotes a

posting list in which each leaf context contains all the characters B_1 , B_2 , ..., and B_i . The operation AND-merge was introduced in [Emra83].

Algorithm-14: Reconstruct a search phrase.
Procedure:
Step-1: Eliminate each term following an AND NOT operator and all operators from the given search phrase. The result is a string containing only Chinese characters.
Step-2: Eliminate all repeated characters from the string obtained in step-1.
Step-3: By using the CIT specified by the scope clause, sort the remaining characters in a nondescreasing order of counts.

It is clear that the reconstructed search phrase is a string consisting of a number of non-repeated characters, say $B_1\ B_2\ ...B_k$ for any. The string $B_1\ B_2\ ...B_k$ has the following properties.
Prop.7: $B_i \neq B_j$ for $i \neq j$.
Prop.8: $count(B_i) \leq count(B_j)$ for $i<j$.
Prop.9: $PL(B_1\ B_2\ ...B_i) = PL(B_1) \cap PL(B_2) \cap ... \cap PL(B_i)$ for $1 \leq i \leq k$.
Prop.10: $PL(B_1\ B_2\ ...B_k) \subseteq PL(B_1\ B_2\ ...B_{k-1}) \subseteq ... \subseteq PL(B_1)$.
Prop.11: $count(B_1) = 0$ implies $PL(B_1) =$ null implies $PL(B_1\ B_2\ ...B_k) =$ null.

These properties provide an important basis for the step-3 of algorithm-13. First, by applying prop.11, if $count(B_1) = 0$, we rather immediately let the phrase-level posting list be null than apply AND-merge operations. Second, referred to prop.8, prop.9 and prop.10, it is obvious that the step-3 applies a shortest-first strategy to the AND-merge operations for a reconstructed search phrase. And it is easy to find the similarities between the shortest-first strategy applied to the AND-merge operations and the solution to the problem "optimal storage on tapes", i.e., a greedy method [HoSa78].

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-merge operations. The evaluation of a reconstructed search phrase has a better performance than that of its original form. The reasons are mentioned as follows. First, it is obvious that a reconstructed search phrase contains less number of 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 orginal form. Second, in algorithm-13, a greedy manner is applied to the step-3. Referred to [HoSa78], it is easy to prove that the AND-merge operations invoke the smallest total number of leaf context-ids.

However, the algorithm-14 is an information-lost operation. That is, some constaints 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 there are some "false-dropped" leaf context-ids that will occur in a phrase-level posting list. A leaf context-id is said to be false-dropped if and only if it satifies a reconstructed search phrase but does not satisfy the original search phrase. In addition, if either the phrase "UNDER N_1 ...N_j " or the

phrase "FROM A_1 ...A_j TO B_1 ...B_k " is given as the scope clause, then a leaf context-id X which does not satify the constrain is also a false-dropped leaf context-id.

6.2 THE POST PROCESS

The post process consists of three operations : the elimination of false-dropped leaf context-ids, the adjustment of context-ids, and the merge operations. They will be mentioned below.

Elimination of false-dropped leaf context-ids. In order to improve the precision of the query processing, the false-dropped leaf context-ids have to be eliminated from each phrase-level posting list. There are three cases have to be considered. First, if the scope clause "UNDER N_1 ...N_j " is given, then for each phrase-level posting list, all the leaf context-ids having no prefix N_1 ...N_j must be eliminated. Second, if the scope clause "FROM A_1 ...A_j TO B_1 ...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 ...A_j to B_1 ...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 applying algorithm-13. For each leaf context-id X contained in the PPL_i , 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 contraints 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, suppose that the context clause "CONTEXTS OF LENGTH k" is given. For the case, if a leaf context-id contained in the PPL_i' has the length equal to or less than the number k, it has not to be adjusted. If a leaf context-id contained in the PPL_i' has the length greater than the number k , say N_1 N_2 ...N_k N_{k+1} ...N_m for any, then its postfix N_{k+1} ...N_m must be trancated. In addition to the second case, if the scope clause "FROM SETS $file_1$, ..., $file_k$ " is given, then each context-id of the file SET which is mentioned in (3) before section 6.1, must also be adjusted. The adjusted SET is called SET'.

Merge operations. If the given scope clause is not of the form "FROM SETS $file_1$, ..., $file_k$ ", then the merge operation is that simply OR-merge all the phrase-level posting lists to form the clause-level posting list as the result by applying the shortest-first method. Referred to section 6.1, it is also easy to show that the OR-merge operations invoke the smallest total number of context-ids. If the scope clause "FROM SETS $file_1$,...,$file_k$ " is given, then an

additional AND-merge opertion is necessary. That is, the post process has to AND-merge the clause-level posting list and the SET' to obtain the final result.

6.3 THE MISCELLANEOUS OPERATIONS

In addition to the query operation discussed previously, there are some miscellaneous operations provided by the query processor, e.g., the save-operation, the *get_ prts* operation, and the *get_ ids* operation. They are entioned below.

Save-operation. Recall from (e21) that the FROM SETS phrase allows a user to search the contexts from some of the results of the previous queries. Hence the query processor must provide a save-operation in order to store the result of a query into a designated file.

Operations get_ ptrs and get_ ids. A textual database having more than one context structures, such as that mentioned in figure 4, is able to provide a query which invokes "interactions" among these structures. The interaction between two ECHO trees E_1 and E_2 is defined as follows : Given a number of continual context-ids of the tree E_1 , say X_1 , X_2 , ..., X_i , to find a number of continual context-ids of the tree E_2 , say Y_1 , Y_2 , ..., Y_j , such that the contexts X_1 , X_2 , ..., X_i are contained in the contexts Y_1 , Y_2 , ..., Y_j . As an example, given a textual database, (T, Ec, Et), consisting of a text T and two ECHO trees Ec and Et which represent the logical structure and the layout structure of the text T, respectively. Suppose a user needs to query which paragraphs of the range from page i_1 to page i_2 satisfy a given search clause. Assume that the paragraphs are denoted by leaf nodes of the tree Ec and the pages are denoted by leaf nodes of the tree Et. Because it is impossible to search paragraphs via the tree Et, the query must be tranferred into which paragraphs of the range from paragraph j_1 to paragraph j_2 satisfy the given search clause by applying the interaction between the trees Et and Ec. The interaction can be performed by the following two steps : First, to find the BP(page i_1) and the EP(page i_2) by applying the *get_ ptrs* operations to the tree Et. Second, to find a sequence of paragraphs containing the subtext of the text T from the position BP(pagei_1) to the position EP(page i_2) by applying the *get_ ids* operations to the tree Ec. In addition, the interaction can be used to prevent the inconsistency occurred in a query expression.

7. THE MAINTENANCE SUBSYSTEM

Recall from section 1 that the maintenance subsystem provides the necessary operations for maintaining the ECHO subsystem and the text retrieval subsystem. Referred to section 4.4, the ECHO subsystem is defined as an ECHO structure, denoted (T,E_1 ,E_2 ,...,E_k). And from section 5, the text retrieval subsystem is defined as a set of ARCIM structures, denoted ((CIT_1 ,PLT_1),(CIT_2 ,PLT_2),..., (CIT_k ,PLT_k)), in which each structure (CIT_i ,PLT_i), $1 \le i \le k$, is the inverted index of the sub-ECHO structure (T,E_i). Hence the

maintenance subsystem must provide the insertion, deletion and modification operations for updating the ECHO structure and the ARCIM structures.

The basic updating operations for a sub-ECHO structure (T,E_i), including *echoinsert*, *echodelete* and *leafmodify*, are mentioned in section 4.3. The operation *echoinsert* is provided for inserting both the text and the ECHO tree of a given context X into the sub-ECHO structure. It is important that before the insertion is performed, the context structure of X must be marked up by using a text processor and then an ECHO tree denoting the context structure has to be built by applying the tree-constructing operation, as mentioned in section 4.2. The operation *echodelete* can be used to remove both the text and the ECHO tree of a designated context from the sub- ECHO structure. In addition, there are two cases of modifications have to be discussed. First, if only a leaf context is modified, then a *leafmodify* operation can be directly used to update the sub-ECHO structure. Second, suppose that a non-leaf context X is modified. Since the context X is actually composed of a set of leaf contexts, the modification of context X can be considered as the modifications of leaf contexts of the context X. Hence the maintenance subsystem must invoke the *leafmodify* operation many times, in which each *leafmodify* operation is for a modified leaf context.

Recall from section 5.2, the ARCIM structure (CIT_i ,PLT_i) must be maintained if the corresponding sub-ECHO structure (T,E_i) is updated. The basic updating operations for an ARCIM structure (CIT_i ,PLT_i) are mentioned in section 5.2. Each of the basic updating operations deals only with the insertion, deletion or modification of a leaf context. Thus for the case of updating a non-leaf context X, the following two steps has to be applied in order to maintain the ARCIM structure.

(1) For the cases of insertion and deletion, the updated context must be decomposed into a set of leaf contexts. For the case of modification, both the original context and the modified context have to be decomposed into sets of leaf contexts.
(2) For each of leaf contexts obtained from step (1), apply a proper basicupdating operation to update the ARCIM structure.

For an ECHO structure (T,E_1 ,E_2 ,...,E_k), it is important that the updating of a sub-ECHO structure (T,E_i) may cause a propagated updating of a sub-ECHO structure (T,E_j), $j \neq i$. For example, the contents of pages must be adjusted if a paragraph is inserted or deleted. The propagated updatings have to be fully manipulated by the system administrator.

8. CONCLUSION

The following are the advantages of the ECHO structure. First, multiple context structures of documents can be provided by the ECHO structure, e.g., the logical structure and the layout structures. Second, contexts of each level can be retrieved by means of the ECHO structure. Thus the ECHO structure has the ability to provide a flexible search unit for retrieving the textual information. Third, the ECHO structure can provide a subrange search to speed up the retrieval performance. That is, a user can specify a subset of the database as the search space. Fourth, the ECHO structure is relatively easy to maintain.

There are just O(log n) nodes of an ECHO tree has to be updated if a context is inserted, deleted or modified, where n denotes the number of nodes of the ECHO tree. Fifth, a sophisticated retrieval method such as ARCIM is easy to attach to the ECHO structure. In addition, the ECHO structure is language independent. For example, the ECHO structure is suitable for either a Chinese textual database or an English textual database.

For an inversion method, the major factors of the search perfojmance are the AND-merge and OR-merge operations. Referred to section 6, the ARCIM can reduce the number of AND-merge operations, the total length of posting lists invoked by the AND-merge operations, and the total length of posting lists invoked by the OR-merge operations. Thus the ARCIM improves the search performance.

However, there are two important constituents of documents excluded from our system at current state. They are the text components of non-character type and the annexed attributes of contexts such as bibliographies of documents and the semantic information of contexts. The representations and operations of them will be studied in the future.

REFERENCES

[AhCo75] Aho, A. V. and Corasick, M. J., "Efficient String Matching: An Aid to Bibliographic Search", *Comm. ACM* 18:6 (June 1975), 333-340.

[BeRG88] Bertino, E., Rabitti, F. and Gibbs, S., "Query Processing in Multimedia Document System", *ACM Trans. Off. Inf. Syst.* 6:1 (Jan. 1988), 1-41.

[BoMo77] Boyer, R. S. and Moore, J. S., "A Fast String Searching Algorithm", *Comm. ACM* 20:10 (Oct. 1977), 762-772.

[CoRD87] Coombs, J. H., Renear, A. H. and DeRose S. J., "Markup Systems and the Future of Scholarly Text Processing", *Comm. ACM* 30:11 (Nov. 1987), 933-947.

[Emra83] Emrath, P. A., *Page Indexing for Textual Information Retrieval Systems*, Ph.D. Diss., Univ. of Illinois at Urbana-Champaign, 1983.

[Falo85] Faloutsos, C., "Access Methods for Text", *Computing Surveys* 17:1 (Mar. 1985), 49-74.

[Hask81] Haskin, R. L., "Special-purpose Processors for Text Retrieval", *Database Engineering* 4:1 (Sept. 1981), 16-29.

[Holl79] Hollaar, L. A., "Text Retrieval Computers", *Computer* 12:3 (Mar. 1979), 40-50.

[Hora85] Horak, W., "Office Document Architecture and Office Document Interchange Formats: Current Status of International Standardization", *Computer* 18:10 (Oct. 1985), 50-60.

[HoSa78] Horowitz, E. and Sahni, S., *Fundamentals of Comptuer Algorithms*, Reading, Potomac, Md.: Computer Science Press, Inc., 1987.

[Hsie88] Hsieh, C. C., et. al., "Full-text Database for Chinese History Documents", In *Proceedings of 1988 International Conference on Computer Processing of Chinese and Oriental Languages*, Aug. 29 - Sept. 1, 1988, Toronto, Canada, 334-340.

[ISO8613] International Organization for Standardization, *Information Processing – Text and Office Systems – Office Document Architecture (ODA) and Interchange Format*, International Standard ISO/DIS 8613, 1986.

[ISO8879] International Organization for Standardization, *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, Draft International Standard ISO/DIS 8879, 1985.

[Knut73] Knuth, D. E., *The Art of Computer Programming: Vol. 1, Fundamental Algorithms*, 2nd Edi., Reading, Mass.:Addison-Wesley Publishing Co., 1973.

[Ozka86] Ozkarahan, E., *Database Machines and Database Management*, Reading, Englewood Cliffs, N.J.:Prentice-Hall Inc., 1986.

[PeJN85] Peels, A. J. H. M., Janssen, N. J. M. and Nawijn, W., "Document Architecture and Text Formatting", *ACM Trans. Off. Inf. Syst.* 3, 4 (Oct. 1985), 347-369.

[SaMc83] Salton, G. and McGill, M. J., *Introduction to Modern Information Retrieval*, Reading, N.Y.: McGiraw-Hill, 1983.

[Tsen82] Tseng, S. S., *The Design of Chinese Character Database*, Chinese Publication, Master Thesis, National Taiwan Institute of Technology, Taipei, Taiwan, 1982.

[Tsen86] Tseng, S. S., "CCCII: The Coding Scheme and The Applications", *Communications of Computing Center, Academia Sinica*, 2:5-8 and 12-13 (Mar., Apr., June, and July, 1986), Chinese Publication.

[Tsen88] Tseng, S. S., et. al., "Approaches on an Experimental Chinese Electronic Dictionary", In *Proceedings of 1988 International Conference on Computer Processing of Chinese and Oriental Languages* , Aug. 29 - Sept. 1, 1988, Toronto, Canada, 371-374.

[TsYH88] Tseng, S. S., Yang, C. C. and Hsieh, C. C., "The Document Representation and A Refined Character Inversion Method for Chinese Textual Database", In *Proceedings of 1988 International Conference on Computer Processing of Chinese and Oriental Languages*, Aug. 29 - Sept. 1, 1988, Toronto, Canada, 376-381.

[Ullm82] Ullman, J. D., *Principles of Database Systems*, 2nd Edi., Reading, London: Pitman Publishing Ltd., 1982.