A Bucket Index Scheme for Error Tolerant Chinese Phrase Matching

Min-Wen Du
Dept. of Information Science
National Chiao-Tung Univ.
Hsinchu, Taiwan
Mwdu@cis.nctu.edu.tw

Hsi-Hso Wu
Dept. of Information Science
National Chiao-Tung Univ.
Hsinchu, Taiwan

Abstract *-- Error tolerant capability is very desirable in designing a Chinese computer input method. It is especially useful in designing an input method based on speech recognition technology because where errors are inevitable. In practical applications, such as natural language speech recognition, we need to handle very large phrase tables. How to do error tolerant phrase matching with very large phrase tables in a real -time speech recognition environment is the problem we are facing.*

This paper developed an index scheme to help the error tolerant phrase matching calculations with very large phrase tables. The approach is based on three concepts. 1. Cartesian Product File. 2. Covering between buckets. 3. Gradual expansion of search region. The results sho that doing multiple error tolerant phrase matching with very large phrase tables in real-time is feasible.

Index terms – Error tolerant, phrase matching, index scheme

I. Introduction

I.1 The Problem

Error tolerant capability is very desirable in designin g a Chinese computer input method. Because of the non -alphabetic nature of Chinese language, traditional Chinese input methods require a user to encode every Chinese character into a symbol sequence to input it into a computer. Chinese input methods are normally designed according to either the patterns of Chinese characters or their phonetic symbol representations or the mix of the two. The encoding processes often require non -trivial mentor efforts, which cause errors easily.

Using speech recognition as a means to communicate with a computer has become feasible recently, thanks to the research work done in the past few decades and the rapid improving of computer speed in these years. Today, a Pentium PC can process speech recognition in real -time. The very nature of using speech as a computer communication medium is that people can do it effortlessly. It is foreseeable that people will increasingly use speech t communicate with a computer. Also, non-alphabeti languages such as the Chinese language will be nefit from the speech recognition technology the most.

Error tolerant technique is important in designing application programs such as a dictation machine because people still need to correct a large amount of errors manually after speech recognition algorithms are applied. The technique is even more important in designing input methods for Chinese language because there is a high percentage of wrong pronunciation in Chinese speaking. Chinese language has many confusing consonants, such as "ㄗ" and "ㄓ", and confusing vowels, such as "ㄞ" and "ㄚ". It is hard for many people to pronounce those ambiguous syllables clearly. Also, because that the Chinese language is non -alphabetic, a person does not know how to pronounce a character if that character is new to him. Pronouncing by arbitrary guessing according

to half character or a component of it is a very common practice for Chinese speaking people.

In this paper we will study the problem of doing error tolerant phrase matching with a very large phrase table. This problem is a fundamental one in speech recognition because any sentence matchin can be decomposed into sub -problems of phrase matching. Our objective is to develop an algorithm that can find the most likely phrase or phrases efficiently giving a matching phrase that may contain some errors.

I.2 Assumptions and Considerations

Our study will be based on the followin assumptions and considerations.

1. P hrase table contents

We consider every Chinese phrase as a sequence of Chinese characters associated with a sequence of syllable codes. Because Chinese language is mono-syllabic, the character sequence and the syllable code sequence are always of equal length. The *i*th syllable represents the pronunciation of the *i*th character of the phrase.

We will limit our discussion to only the Mandarin dialect of the Chinese language. In our current phrase collection, there are 1412 distinguishable Mandarin syllables. Each of these syllables is represented by a 16-bit internal code (*I_code*). Each Chinese c haracter will also be represented by a 16-bit code (*BIG-5 code*).

2. Error model

Because the phrase matching considered here is a sub-problem of the sentence matching problem in speech recognition, the phonetic sequence part of a phrase is the main information of our concern Therefore, every phrase can be thought of as simpl an I_code sequence. The problem of finding the most likely phrases giving a phrase containing errors is ver similar to the *Approximate String Matching* (ASM) problem defined on a set of symbol strings [4], [5], [6], where the problem has been formulated as finding the nearest neighbors of a given string among a set of possible words.

There are some fundamental differences between the ASM problem of a set of character strings and the error tolerant phrase matching problem in Chinese speech recognition, however. In the ASM problem, the insert, delete, change a character, and transpose two characters are considered to be the most common errors to occur. In Chinese speech recognition, the i nsert, delete, and transpose errors seldom occur. Which may be attributed partly to that the Chinese language is mono-syllabic, and it is relatively easy to do correct syllable segmentation on the speech input.

Therefore, in our discussion we will assume t hat the model contains only change errors, i.e., errors where the I_codes may change to other I_codes in the phonetic sequence of a phrase.

3. Phrases and Large Phrase Tables

For the phrase matching purpose, we may consider a phrase as a sequence of I_c odes. In any application, the I_code sequence of a phrase must be meaningful in the sense that it is actually used in an application domain. In many situations the phrase table can be huge. Given the 1412 distinct syllables, there can be at most 1412^k distinct (phonetically) k-syllable phrases. This upper bound is 2×10^6, 2.8×10^9, and 4×10^{12} for k equal to 2, 3, and 4. It is conceivable that in an application such as the speech recognition of Chinese natural language, the size of the phrase table will be huge, although it is much smaller than the upper bound given above. Note that a phrase is defined here as a sequence of syllables not an idiom in the common sense.

4. R eal-time Performance Requirement

A special requirement in speech recognition applications is that the user must get the response in real-time. During the input phase of a Chinese dictation system, only sub -second response will be acceptable for a continuous operation. In a Chinese text editing application, where human interactions are involved, a few seconds delay of response are ordinarily considered endurable. These real-time requirements set the limits on the algorith performance of our design.

5. S ystem Architecture

We will assume that a two-level storage hierarchy is available to store the phrase table in our algorithm design. We also assume that the two levels of storage are the main memory and a hard disk storage. At present, the size of main memory available for working area in most personal computer is several mega (10^6) bytes. The memory access speed is of the order of 10^8 instruction cycles per second. The size of the hard disk storage is several giga (10^9) bytes, while the disk access time is about 10 ms, i.e., 100 times per second.

In this paper we will design indexes to help data accessing in the two -level storage hierarchy for error tolerant phrase matching calculations. There are many ways to organize t he data structure on a two-level storage hierarchy. Different schemes will be suitable for different sizes of the phrase tables. We will adopt the scheme where the whole phrase table is stored in the hard disk and buffer areas of reasonable large size are available to load the index and data into memory for fast calculations.

II. Error Tolerant Phrase Matching Scheme

II.1 Set of Syllables in Mandarin

In Mandarin there are 21 consonants, 16 vowels, and 5 tones. There are also diphthongs each begins with a vowel of "ㄧ", "ㄨ", "ㄩ" and ends with other vowels.

The pronunciation of every Chinese character, called a syllable, can be represented by

combination of these phonetic symbols. We can represent any syllable by the follow four components

C: Consonant, can be nil.

H: Head of Diphthong, can be either "ㄧ", "ㄨ", "ㄩ" or nil.

V: Vowel.

T: Tone.

In Mandarin, there are 5 tones from tone 0 to tone 4. Tone 0 is the neutral tone. The C, H, V, T components can be packed into a 16-bit code called I_code (internal code). The numbers of bits used for storing the C, H, V, T components are 5, 3, 5, 3 respectively.

Example 1. Fig.1 shows the encoding of "ㄨㄛˇ", the syllable corresponding to "我", in CHVT format. There is no consonant. The head vowel, H, is "ㄨ". The second vowel, V, is "ㄛ". The tone, T, is "ˇ" or 3. So, checking with Fig. 1, we get C=0, H=2, V=2, T=3. Therefore, the I_code of the syllable "ㄨㄛˇ" is 6240 in hexadecimal.

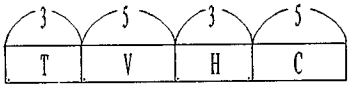

Fig. 1. The I_Code of "我".

In different application domains the numbers of totally used syllables may be different. In our collections of Chinese phrases, there are 1412 distinct syllables.

II.2 Partition of Syllable Domain

Cartesian Product Fil concept has been proved to be a good method to manage files, especially when used in Partial Match Query systems [2], [3]. A Cartesian product file that contains many buckets is a structured file. Each bucket contains data that have highly relevant attributes. The Cartesian product file shows highly cluster property. Fewer disk accesses are needed in a partial match search if the file to be searched is a Cartesian Product File. We will adopt Cartesian product file concept in designing the index scheme and store phrases.

To apply Cartesian product file concept, we need to group together closely related elements in the domain into sub-domains (partitions or blocks). I practical situations of speech recognition applications, we can observe a phenomenon that if we determine that a syllable is *s*, it is much likely that the syllable is actually a syllable *x* in a set *L* (including *s*) other than any syllable outside of *L*. We associate such set *L* with *s* as its *likelihood set*, also denoted as $L(s)$. Likelihood sets can be viewed as sub-domains. In general, syllables in a likelihood set have similar pronunciations. We may partition the set of all syllables according to their pronunciations.

We will view all the 1412 Mandarin syllables as one big Domain D. Partitioning *D*, we can get many sub-domains. For example, we may partition D in the following way. Two syllables $I_1 = C_1H_1V_1T_1$ and $I_2 = C_2H_2V_2T_2$ will be put into the same sub-domain iff $V_1 = V_2$. We have $L(I_1) = L(I_2)$.

The domain *D* will be partitioned into 8 sub-domains of similar pronunciations as shown in Fig. 2.

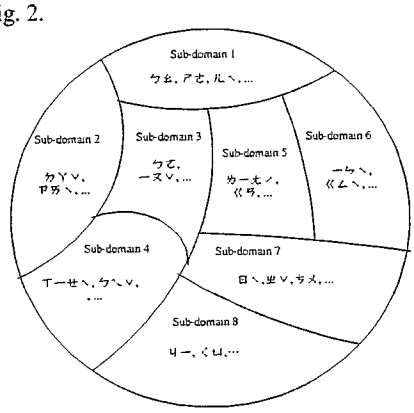

Fig. 2. A partition of Mandarin syllabl domain into 8 sub-domains.

II.3 Cartesian Product of Syllable Domain an Phrase Bucket

The Cartesian product of two sets *D* and *E*, represented as $D \times E$, is the set of all possible pairs (d, e) with *d* in *D* and *e* in *E*. The Cartesian product of more than two sets is defined similarly. Assume that the whole syllable domain *D* is partitioned into *k* sub-domains. Then the Cartesian product of two syllable domains D_1 and D_2 will be naturally partitioned into k^2 groups. Two combinations (a_1, b_1) and (a_2, b_2) will be put into the same group if a_1, a_2 are in the same sub-domain of D_1 and b_1, b_2 are in the same sub-domain of D_2. In general, the Cartesian product of *r* syllabl domains D_1 to D_r will be naturally partitioned into K^r groups. We call each of these groups a bucket of the Cartesian product of the syllable domains.

Let *P* be a table of phrases each of len gth *r*. The buckets of the Cartesian product of *r* syllable domains also partition *P* into subgroups. If two phrases are put into the same group, they should belong to the same bucket of the Cartesian product.

II.4 An Error Tolerant Phrase Matching Procedure

How do we do the error tolerant phrase matching? We describe it in the following paragraph.

Assume that the phrase table that we are dealing with is *P*, and the syllable sequence determined by the speech recognition algorithm is $S = s_1 s_2 \cdots s_n$. To find which phrase or phrases i *P* matches *S* best, we will consider phrases $L(s_1) \times L(s_2) \times \cdots \times L(s_n)$ in *P* first. If the

intersection is not empty, we may load it into the memory and perform a detail matching analysis to determine the best matched phrases. If the intersection is an empty set or phrase is not found, we will relax one component in the Cartesian Product to the whole syllable domain and load those phrases in the enlarged new Cartesian product sets into the memory and perform the detail matching calculations. This procedure will continue (i.e., relax k of the components in the Cartesian product, with k increasing by one each time) until a set of best matched phrases is found.

Let $T = t_1 t_2 \cdots t_n$ be a sequence of n syllables. If k of the components of $t_1 t_2 \cdots t_n$ are not in the corresponding $L(s_i)$, we say that th error distance from $s_1 s_2 \cdots s_n$ to $t_1 t_2 \cdots t_n$ is k blocks, or S is k blocks from T. Because we have already considered the possible change errors in the notion of likelihood set $L(s_i)$, a phrase which is k blocks from S will match S much better than another phrase which is more than k blocks from S. This justifies the best match searching procedure described above.

The remaining problem is, how do we do the error tolerant searching efficiently, in terms of time and memory space to use. In Section 3 we will develop an index scheme to support the searching procedure.

III. Bucket Index Structure

III.1 Buckets and Partitions of Phrase Table

Assume that we are given a set of phrases. Because we consider only change errors, which will not alter the length of a phrase, it is convenient t classify the sentences into equal length phrases tables. Therefore in our discussion we may consider tables of phrases of fixed length. Let P be a phrase table containing phrases of length n.

Let D be the syllable domain, i.e., the set of all the syllables. Let $R = \{R_1, R_2, \cdots, R_k\}$ be a partition of D. Then R also implies a partition on the Cartesian product of $D_1 \times D_2 \times \cdots \times D_n$ where $D_i = D$ for all $1 \le i \le n$. Every $B_1 \times B_2 \times \cdots \times B_n$ with each B_i in R is a bucket of $D_1 \times D_2 \times \cdots \times D_n$. In fact, R also implies a partition on P where two phrases are in the same partition if they belong to the same bucket of $D_1 \times D_2 \times \cdots \times D_n$.

III.2 Buckets with Unknown X Components

To perform the error tolerant phrase matching described in Section II.4, we need to be able to do the following sub -task: Given a syllable sequence $S = s_1 s_2 \cdots s_n$, find all the buckets that contain a phrase (a syllable sequence) $T = t_1 t_2 \cdots t_n$ such that S is k blocks from T. If all the buckets of $D_1 \times D_2 \times \cdots \times D_n$ with the domain partition $R = \{R_1, R_2, \cdots, R_k\}$ are stored in the hard disk, this sub -task can be done easily by checking each s_i with each sub-domain R_j. A bucket $B = B_1 \times B_2 \times \cdots \times B_n$ contains a phrase that is k blocks from S if there are exactl k s_i not in B_i. Once these blocks having been found, they can be loaded into the memory to perform a detail matching analysis.

There is a practical issue here, however The number of buckets in a typical application will be too large to allow an efficient implementation. To see this, let's assume that the syllable domain is partitioned into 100 sub -domains. There will be 100^7 buckets to consider for a phrase table containing phrases of length 7

Here the idea of covering comes to pla [4]. We will also implement the bucket covering index scheme of [4] to help our error tolerant phrase matching operations. We will describe the idea informally by using examples.

The X symbol [4] is very useful in explaining the covering idea. It can also be called an unknown symbol. Let 12345 be a simplified notation of the bucket $B_1 \times B_2 \times B_3 \times B_4 \times B_5$. Then 123X5 will be used to represent the set of all the phrases with 4 in 12345 r eplaced by an arbitrary syllable. In other words, 123X5 is the Cartesian Product $B_1 \times B_2 \times B_3 \times D \times B_5$ where D is the syllable domain. It can also be considered as an enlarged bucket of $B_1 \times B_2 \times B_3 \times B_4 \times B_5$.

III.3 Covering Relations Between Enlarge Buckets

Given a phrase table P with phrase length 5, and a syllable sequence $S = s_1 s_2 s_3 s_4 s_5$. Let S be in the bucket 12345. Then a phrase $T = t_1 t_2 t_3 t_4 t_5$ which is 1 block error from S will be contained in one of the following enlarged buckets X2345, 1X345, 12X45, 123X5, and 1234X. If we have built inverted files on arbitrary 4 components of the phrases in the phrase table P, then it is very easy to check whether such T exists in P. The covering idea of [4] tells us that we can do better.

We use [i, j] to denote an inverted index file built on the ith (from left) and jth components of a phrase table. For instance, [1, 3] represents an inverted index file for P built on the 1st and the 3rd syllables of the phrases in P.

To check whether a phras e $T = t_1 t_2 t_3 t_4 t_5$ is in an enlarged buckets B, we may check whether T is in another enlarged bucket which contains B. For example, to check whether T is in the bucket 1X345, we may check whether T is in the bucket 1X3XX. If T is indeed in 1X3XX, we can further examine components 4 and 5 of T to verify that T is also in 1X345. Otherwise, T cannot be in 1X345.

If an enlarged bucket B_1 contains another enlarged bucket B_2, we say that B_1 covers B_2. Fig. 3 shows an example of these covering relationship

between (enlarged) buckets. The covering relation is indicated by "×" symbols at the intersections of the rows and columns.

From the figure, we see that the buckets [1 2] and [4, 5] covers all the buckets X2345, 1X345, 12X45, 123X5, and 1234X. Therefore, inverted indexes built for [1, 2], [4, 5] can be used to check whether a phrase $T = t_1, t_2, t_3, t_4, t_5$ is contained in one of the X2345, 1X345, 12X45, 123X5, and 1234X buckets.

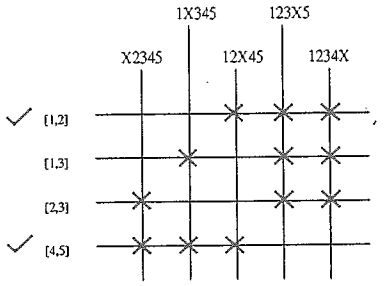

Fig. 3. Covering relations betwee enlarged buckets.

III.4 Bucket Index Structure

Fig. 5. shows the structure of the bucket index system. We store the whole phrase tables in the external disk memory. Buckets of Cartesian products corresponding to a partition of the syllable domain are also stored in the hard disk. Buckets will be l oaded

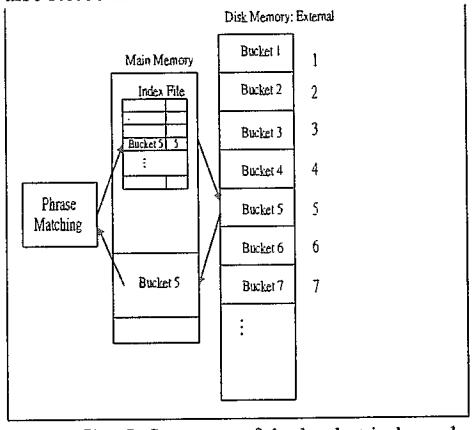

Fig. 5. Structure of the bucket index scheme.

into main memory only when needed. Locating the buckets is helped by the bucket indexes. We assume that the whole index files are loaded into the main memory at system initialization.

III.5 Example of Bucket Searching

Example 1. We are given the phrase (ㄓㄣ, ㄚ, ㄧˇ, ㄙˋ, ㄑㄧㄥˊ), a phrase of length 5 containing 2 block errors. The correct phrase is (ㄓㄣ, ㄞˋ, ㄧˇ, ㄕˋ, ㄑㄧㄥˊ), i.e., "真愛一世情" or "True love is everlasting." Assume that the syllable domain is partitioned into 8 sub-domains as shown in Fig. 2.

The four inverted indexes and buckets correspondin to (ㄓㄣ, ㄚ, ㄧˇ, ㄙˋ, ㄑㄧㄥˊ) is listed below:

[1,2]: "ㄓㄣ" belongs to sub -domain 6, and "ㄚ" belongs to sub-domain 2, so the bucket is bucket(62XXX).

[1,3]: "ㄓㄣ" belongs to sub -domain 6, and "ㄧˇ" belongs to sub-domain 8, the bucket is bucket(6X8XX).

[2,3]: "ㄚ" belongs to sub -domain 2, and "ㄧˇ" belongs to sub-domain 8, so the bucket is bucket(X28XX).

[4,5]: "ㄙˋ" belongs to sub-domain 7, and "ㄑㄧㄥˊ" belongs to sub-domain 6, the bucket is bucket(XXX76).

We assume that the system has built the four [1, 2], [1, 3], [2, 3], and [4, 5] inverted indexes and their corresponding buckets to cover phrases of length 5 containing 2 possible block errors. Only 4 buckets of the phrases will be loaded into the main memory for further detail examination. The flow is

1. #Errors=0. Either [1, 2], [1, 3], [2, 3], or [4, 5] is desirable. If [1, 2] is selected, bucket(62XXX) is loaded into the memory. The phrase is not found within 0 error, go to step 2.
2. #Errors=1. Both of [1, 2] and [4, 5] are used. Bucket(62XXX) and bucket(XXX76) are required t be loaded. Bucket(62XXX) has been loaded in step 1, so only bucket(XXX76) is loaded in this step. The phrase is not found within 1 error, g o to step 3.
3. #Errors=2. All of [1, 2], [1, 3], [2, 3], and [4, 5] are used. Bucket(62XXX), bucket(6X8XX), bucket(X28XX), and bucket(XXX76) are required t be loaded. Bucket(62XXX) and bucket(XXX76) has been loaded in previous steps, so only bucket(6X8XX and bucket(X28XX) are loaded. The phrase is found within 2 errors! The most likely phrase is found.

IV. Performance Analysis

The memory space required to implement the bucket index scheme and the time required t perform the error tolerant phrase matching u sing the scheme are the two items to examine for the performance analysis. Both the space and the time required depend on the error tolerant capability the system wants to implement.

IV.1 Number of Inverted Files Required

We assume that the index scheme discussed in Section 3 is implemented by usin inverted files [8]. The number of inverted files required in an implementation depends on how many block errors the system plan to handle. For example, for a table with phrases of length 5, we need 2 inverted files to cover 1 block error (Fig. 3), and 4 inverted files to cover 2 block errors (Fig. 4). In general we use two syllable positions in building an index file. When the number of the syllables is less than 3, we use only one syllable position for the purpose. Table 1 lists the numbers of index files required for other cases. The * symbol represents "not

supported."

Errors \#Syllables	1	2	3	4	5
0	1	1	1	1	1
1	*	2	3	2	2
2	*	*	*	6	4
3	*	*	*	*	10

Table 1. Number of index files required to cover block errors.

IV.2 Memory Requirements

The major memory space required is to be used to store the phrase tables, buckets, and inverted files. A phrase table consists of two parts, an I_code part for phonetic information and a BIG -5 code part for Chinese character information. An I_code and a BIG-5 code both require 2 bytes to represent. For a table of phrases with phrase length l and n phrases, the space required is $2 \times l \times n$ for the phonetic part, and the same amount for the BIG-5 code part.

The major part of an inverted index file is a table of the form shown in Fig. 6. Each entry of the table consists of a key field of bucket named by its value at the selected positions of the syllables and a field for the address of a bucket in the disk. As sume that 2 syllable positions which require 2 bytes to represent are used for the key field and 4 bytes are required for the bucket address, each index file requires a space of size $(4+4) \times m$ bytes, where m is the number of buckets used for the phrase table. Let the number of phrases in the table be n, and the average bucket size be s, then m is approximately n/s.

⋮	⋮
11	Address of Bucket(11XXX)
12	Address of Bucket(12XXX)
⋯	⋯
88	Address of Bucket(88XXX)
⋮	⋮

Fig. 6. The structure of the index file.

The content of a bucket file of a phrase table is the same as the phonetic part of the phrase table, except that the entries are grouped according to the bucket structure. Therefore, a space of $2 \times l \times n$ bytes is required for each of the inverted index files, where l is the phrase length and n is the number of phrases in the table.

Note that the phrase tables and the bucket tables can be kept in the hard disk. The index files can be loaded into the memory if there is enough space in the memory. Because the index files are sorted files, they can also be implemented in two-level structures. In that case only a small first-level header index need to be kept in the memory during execution. One extra disk access is required to load a portion of the index file into memory before the searching of the buckets can be done.

IV.3 Execution Time Requirements

The major time spent in performing an error tolerant phrase matching using the bucket index scheme is on 1. Loading of buckets, 2. Detail calculation of the distance of the phrases (syllable sequence) in the loaded buckets and the given possibly erroneous phrase. Assume that buckets are stored on a continuous space in the disk. The loadin of a bucket will require one disk access time, including seek time, rotational delay, and block transferring time [8] from the disk to the memory for the bucket. Both of the seek time and rotational time are always counted in average, so they are always conceived stable. The block transferring time is proportional to the size of the bucket. The time required to do the detail distance calculation is also proportional to the total size of the buckets. Therefore, the time required to perform an error tolerant phrase matching using the bucket index scheme can be estimated by the following formula

$$Matching\_Time = n_D \times t_D + b \times v_B \times t_B + c \times v_B \times t_C ,$$

where n_D = total number of di sk access times, equal to the number of buckets to be loaded,

t_D = disk seek time and rotational delay,

b = a positive constant for the block transferring term,

v_B = total size of the buckets transferred,

t_B = data transferring rate of the disk,

c = a positive constant for the calculation term,

t_C = average instruction time of the CPU.

For today's personal computers, t_D is of the order of 10^{-2} second. t_B is of the order of 10^{-7} second. t_C is of the order of 10^{-8} second. The constants b and c can be assumed to be of the order of 10.

Obviously, t_D is much large than t_B and t_C. Thus, the time required to transfer the block from the disk to the memory and the time required to do the detail distance calculation can be negligible. We will examine how much disk access time required to perform the error tolerant phrase matching usin the bucket index scheme in Section 5.

V. Experimental Results

From formula matching_time we see that the major factors affecting the performance of the error tolerant phrase matching of the bucket index

scheme are n_D and v_B. n_D is the total number of disk access times, equal to the total number of buckets fetched in the process. v_B is the total size of the buckets transferred. In this Section we will test on data of real phrase tables to see how the n_D and v_B values vary.

V.1 The Buckets Creation

We have gathered 60345 Chinese phrases to be processed in our experiments. The phrases are put into sub-tables according to phrase lengths, as shown in Table 2.

Length	1	2	3	4	5
# Phrases	14098	20490	13022	11938	797

Table 2. The number of phrases of different phrase lengths.

We have to create buckets to store the phrases of different phrase lengths. In our experiment, the buckets we created for different phrase lengths are listed in Table 3.

Phrases of length	The indexes to be created
2	[1], [2]
3	[1,2], [1,3], [2,3]
4	[1,2], [1,3], [1,4], [2,3], [2,4], [3,4]
5	[1,2], [1,3], [1,4], [1,5], [2,3], [2,4], [2,5], [3,4], [3,5], [4,5]

Table 3. The list of indexes we created for the experiments.

In Table 3, the inverted index of the phrases of 1 syllable is not listed because there is onl one choice to build the inverted index. After indexes are created, the buckets will be stored in the disk. Associated index files are also created and stored.

V.2 Domain Partitioning

We are interested in how to partition the whole syllable domain D. Because how you pa rtiti D will change the way the buckets created. It is believed that partitioning D into more sub -domains can decrease the number of phrases of every bucket. The system can then perform the error tolerant phrase matching with fewer phrases among the buckets.

We will compare the effect results of 8, 16, 24, and 32 sub -domains. We will partition D into 8 sub-domains by the similarity of vowels, as shown in Table 3. In Table 7, 16 sub -domains are created b dividing the 8 sub-domains with reference t consonants. Let "ㄅㄆㄇㄈㄉㄊㄋㄌ" be part 1, and "ㄍㄎㄏㄐㄑㄒㄓㄔㄕㄖㄗㄘㄙ" be part 2. If the vowel of the syllable belongs to "ㄚㄞ" and the consonant of the syllable belongs to part 1, then the sub-domain of the syllable is 3.

In the same way, a partition of 24 sub-domains are created by trisecting the 8 sub-domains each into 3 by consonants. Let "ㄅㄆㄇㄈㄉㄊㄋㄌ" be part 1, "ㄍㄎㄏㄐㄑㄒㄓㄔㄕㄖㄗㄘㄙ" be part 2, and "ㄧㄨㄩㄚㄛㄜㄝㄞㄟㄠㄡㄢㄣㄤㄥㄦ" be part 3. A partition of 32 sub-domains are created by dividing each of the 8 sub-domains into four blocks by consonants. Here w use "ㄅㄆㄉㄊ" as part1, "ㄇㄈㄋㄌ" as part 2, "ㄍㄎㄏㄐㄑㄒㄓㄔㄕㄖㄗㄘㄙ" as part 3, and "ㄧㄨㄩㄚㄛㄜㄝㄞㄟㄠㄡㄢㄣㄤㄥㄦ" as part 4.

V.3 Maximum Bucket Siz

Fig. 7 shows the maximum bucket sizes created for the phrase tables of Table 2. Bucket size represents how many phrases actually contained in the bucket. The phrase tables with phrase length less than 3 will use only one syllable position to create buckets. For tables with phrase length equal to 3 or greater than 3, two domain partitions.

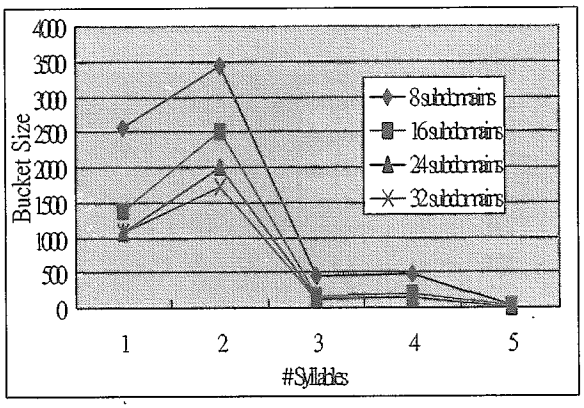

Fig. 7. The maximum bucket sizes of different doma in partitions.

From Fig. 7, we see that the maximu bucket sizes of 24 and 32 sub-domains are very close. This situation may result from that the creation of the sub-domain is not good enough or the phrases we gathered are not fair.

Let's consider the case of phrase table of 4 syllables, 2 possible errors, and 32 sub -domains. In this situation, the system needs to load at most 6 buckets to search the phrases. In the worst case, we can assume that all the 6 buckets each contains 127 phrases. There are onl 762 phrases required to be matched. These phrases are 6.38% of 11938, the number of all the phrases of length 4 in Table 2.

We can find that the more sub -domains we partition the Domain into, the smaller the bucket sizes are. The number of the sub -domains will affect the number of buckets, so increasing the number of sub-domains will increase the number of the buckets, and decrease the size of the buckets.

V.4 Total Number of Buckets Fetched

When we build the bucket index, we can find that some bucket may contain no phrases. Those buckets need not be loaded during execution. Here we are interested in testing how many buckets are there

that actually contain phrases when we apply the buckets index scheme to actual phrase data. We did an experiment to test this. We select 500 phrases with phrase length of 5. Then artificially insert one, two, and three errors into the phrases. By doing this, we obtain 4 sets of testing phrases containing one, two, and three errors. The testing results are shown in Table 4 and Fig. 8.

# Sub-domains \ # Errors	0	1	2	3
8	500	1000	2000	5000
16	500	979	1967	4871
24	500	832	1168	3231

Table 4. The number of searched indexes.

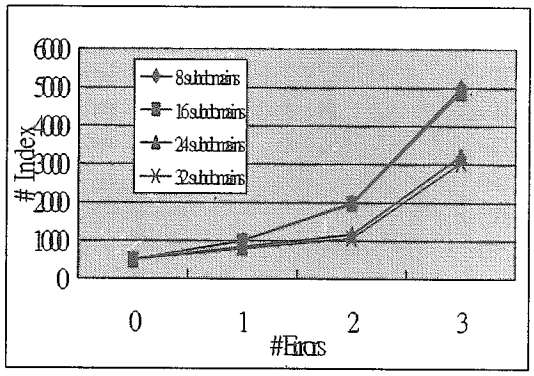

Fig. 8. The number of searched indexes.

According to Table 4, the phrases of 5 syllables should search 1 bucket if there is no error, 2 buckets if there is 1 error, 4 buckets if there are 2 errors, and 10 buckets if there are 3 errors. Thus the number of searched buckets is less than 500, 1000, 2000, 5000 under different number of errors if system processes 500 phrase inputs. The actual number of buckets searched is very close to these upper bounds. The discrepancy comes from that some buckets may be empty and therefore unnecessary to search.

VI. Conclusion

This paper studies the problem of designing error tolerant phrase matching algorithms for very large phrase tables. The design goal is t implement algorithms with sufficient error tolerant capability and with real-time performance. Our approach was based on three major concepts. 1. Cartesian Product File concept, 2. Covering concept of enlarged buckets, 3. Gradual expansion of near-neighbor searching concept.

To achieve the error tolerant ability, the Cartesian Product File concept is used to partition the phrases into buckets according to the similarity of their pronunciations. All the buckets are stored in the external disk memory. To decrease the number of indexes required in the calculation, the Coverin concept is applied in creating the index files. By searching fewer indexes, including empty indexes, fewer buckets are needed to load into the internal memory. Thus fewer phrases are checked. When searching for the best match phrases, we gradually expand the search region. If the best match phrases are in the near neighbor of the given testing phrase, whic h is often the case, only small amount of buckets are needed to be loaded into the memory. It makes handling of multiple errors in real-time possible.

The research reported in this paper provides a foundation to do multiple error tolerant phrase matching with very large phrase tables. Because of the difficulty of collecting actually used phrase data, we have tested the algorithms only on medium size phrase tables. It would be interesting to test the scheme developed on really large phrase tables in the future.

References

[1] L. S. Lee, C.Y.Tseng, H.Y. Gu,F. H. Liu, C. H. Chang, Y. H. Lin, Y. Lee, S. L. Tu, S. H. Hsieh, and C. H. Chen, "Golden Mandarin (I) — A Real-Time Mandarin Speech Dictation machine for Chinese language with Very Large Vocabulary," *IEEE Transactions on Speech and Audio Processing,* vol. 1, no.2, pp. 158-179, Apr. 1993.

[2] C. C. Chang and M. W. Du, "The Hierarchical Ordering in Multiattribute Files," Information Sciences, vol. 31, pp. 41-75, 1983.

[3] C. C. Chang, R. C. T. Lee, and M. W. Du, "Symbolic Gray Code as a Perfect Multiattribute Hashing Scheme for Partial Match Queries," *IEEE Transactions on Software Engineering,* vol. se-8, no. 3, pp. 235-248, May 1982.

[4] M. W. Du, and S. C. Chang, "An Approach to Designing Very Fast Approximate Strin Matching Algorithms," *IEEE Transactions on Knowledge and Data Engineering,* vol. 6, no. 4, pp. 620-633, Aug. 1994.

[5] M. W. Du and S. C. Chang, "A model and a Fast Algorithm For Multiple Errors Spellin Correction, *Acta Informatica,* vol. 29, pp. 281-302, 1992.

[6] J. T. Wang, and C. Y. Chang, "Fast Retrieval of Electronic Messages That Contain Mistyped Words or Spelling Errors, *IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics,* vol. 27, no. 3, pp. 441-451, June 1997.

[7] H. L. Morgan, "Spelling correction in systems programs, *Commun. ACM,* vol. 13, no. 2, pp. 90-94, Feb. 1970.

[8] M. J. Folk, B. Zoelick, *File Structures. Reading,* MA: Addison-Wesley, 1992.