NTCIR-2 Experiments Using Long Gram Based Indices

Takashi SATO Nao HATTA Koji HIRAIWA
Kihei KOBATA Akihiro FURUSHO Koto HAN
Osaka Kyoiku University
4–698–1 Asahigaoka, Kashiwara, Osaka, Japan
sato@cc.osaka-kyoiku.ac.jp

Abstract

Long gram based indices are experimented at NTCIR-2. In making gram based indices, no analyses such as morphological ones are required. The accessing number, titles, abstracts and keywords are extracted from NTCIR-2 documents. The total index size is 1.43Gbyte and time to make indices is about 100 minutes. Average retrieval time per topic takes 21 seconds since documents are ranked in a Perl program which is simple and not fast. Ranking algorithm used is based on a traditional probabilistic model, and the result is standard average precision.

Keywords: *long gram based index, gram coding, NTCIR.*

1 Introduction

The use of Web pages, on-line documents, electronic books, newspapers, and so on, which we can retrieve directly from the internet using computers, is increasing more and more these days. In order to make the best use of the potential information which these on-line texts contain, we need the means to access objective information efficiently. In particular, we should be able to access arbitrary strings in the text at high speed. Retrievals of fixed keywords, which are prepared in advance, is not sufficient to support retrievals from such diverse sources.

When we retrieve text by uni-string, the output is often too large. So we have to squeeze the result by successive retrievals and strings which relate to a retrieved string may be added. Also, some retrieval systems can adjust queries interactively based on retrieved results. In this way, many retrieval results have to be combined organically in order to serve the user's purpose. Therefore, string retrieval engines should be fast.

To realize fast retrieval from a large text, indices are indispensable. Indices based on sistring are dedicated to fast string retrieval[1, 2, 3, 4, 5]. Suffix array (or PAT array), which is made from sistrings, is often used for string search, however, it is very inefficient when text size exceeds main memory size. Recently, n-gram based indices have drawn attention since they have low construction cost and do not require intensive processing such as bit operations[6, 7, 8, 9, 10].

Not only speed but also size is an important element of indices. Since oriental languages, including Japanese, have a large alphabet, it is often remarked that indices tend to become very large in these cases[6, 10]. Conversely, if we reduce the index size, many false drops occur. Consequently, an index structure which allows both low space overhead and a low false drop rate is required.

To meet the above requirements, we have been studying n-gram based indices which differ from standard n-gram indices in the following ways[11, 12, 13].

[long gram] Usually, strings shorter than the gram are not retrievable[6, 7, 8, 9, 10]. If we want to use a 3-gram index for example, we have to prepare not only a 3-gram but also a uni and a 2-gram index. From the space cost viewpoint, we can not use a large gram index. However, by turning the index into a tree structure, we have seen that shorter grams may be retrieved by sequential scan of leaves so that indices for shorter grams become unnecessary.

[no position] Usually, occurrences of a gram are recorded in terms of a document number and an offset (called position) and the index manages such pairs of data[6, 7, 8, 9, 10]. If a set of grams, which are made from decomposition of a retrieve key, are found in a document, the result is a false drop if the grams are not adjacent in the document. So position is necessary to examine the adjacency of grams. However, we will not store position. This is because, (1)By using long grams, the key often fits inside a gram. We are guaranteed no false drop in these cases. (2)Even if the key is long and divided into grams, false drops seldom occur, because candidate documents are sufficiently limited by the high selectivity of long grams. (3)If necessary, one can determine whether or not the result is a false drop by accessing the original text.

[gram coding] Usually, a gram is made from simple concatenation of characters which compose the

gram[7, 10]. Or some hashing is applied to the gram with some parameters such as frequency[6, 8]. However, hashing causes false drops because of collisions among hashed values. We will code the gram so as to make the index compact and a good filter with no false drop. We avoid intensive processing such as bit operations while searching an index. Once a gram or a set of grams is made from the key, searches of an index are processed efficiently by word or byte sized units.

We experimented our long gram based indices at NTCIR-2. Since our indices are based on grams, no analyses such as morphological ones are required. However, we used NTCIR-2 segmented topic data in order to search words and compound words. We made indices from titles, abstracts and keywords of NTCIR-2 documents. The total index size is 1.43Gbyte and time to make indices is about 100 minutes. Average retrieval time per topic takes 21 seconds since documents are ranked in a Perl program which is simple and not fast. Ranking algorithm used is based on a traditional probabilistic model, and the result is standard average precision.

2 Definitions

The alphabet is denoted by Σ. $\sigma(= |\Sigma|)$ denotes the size of the alphabet. Although any set of symbols can be an alphabet, we used Japanese EUC two byte characters ($\sigma = 8836$) in our experiments. A *text* retrieved is a collection of m *documents*. The number of characters in the i-th document is $n_i (1 \leq i \leq m)$. The size of the whole text is $n(= \sum_{i=1}^{m} n_i)$, which is considered too large to be put in main memory. Instead it is put in secondary memory.

We extract strings called *grams* which start at every character from all documents, code and sort them, and then make an index. The index is also put in secondary memory. Although the number of characters which constitute a gram is not constant, coded grams (called gram values) are fixed length w_g[byte] in order to make retrieval fast. Since the index is put in secondary memory, pointers in it are offsets from the head of the file. The size of the pointers is w_p[byte]. The result of retrieval is a set of document numbers. Document numbers can be represented in w_d[byte].

A string sought is a *key* k. A key is composed of characters in Σ and coded in the same method as grams are. The size of a coded key is l_k[bit].

Since the text and index are put in secondary memory, we have to transfer data between main and secondary memory while processing. Data are transferred in blocks of size B[byte].

3 Gram Based Index with Tree Structure

We propose an index which consists of a *root*, a *leaf* and a *locator*(see Fig.1). The elements of a leaf are slots. Slots contain a pair consisting of a gram value and a pointer. The pointer points to a bucket in the locator. Each bucket has document numbers in it. The bucket, which is pointed to by a slot, stores the document numbers where the string corresponding to the gram value in the slot is found.

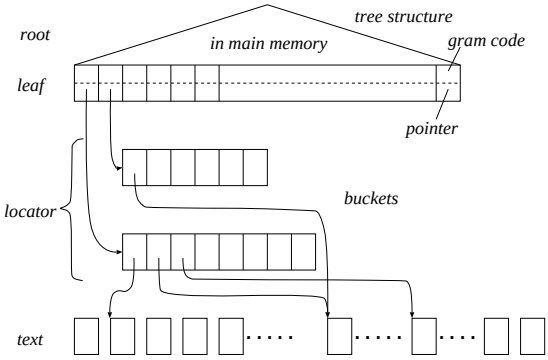

Figure 1. Gram Based Index Structure

As a leaf of a B–tree[14] stores not only keys but also pointers, we can fill a leaf with not only gram values but also document numbers.

However, we separate them to avoid unnecessary reading of document numbers when we are searching for a gram value in the leaf. We can also avoid uncompressing unnecessary document numbers in the case of a compressed index.

The root stores the gram values of regularly spaced slots of the leaf in order to guarantee only one block access to any required slot. The root is compact and can be put into the work space of the main memory provided the retrieved text is smaller than a few dozens of gigabytes. If we can not fit the whole root in main memory, we can make it into a multi-leveled structure, i.e. a tree.

3.1 Space Cost

The size of a slot in the leaf is $w_g + w_p$[byte]. The number of grams made is n, since we make grams starting at every character of all documents. However, we do not save the same gram values repeatedly into the leaf. We call this removal of redundancies unification with associated ratio $u_s (0 < u_s \leq 1)$. The gram values in slots of the leaf are arranged in ascending order. The pointers in slots are also in ascending order since buckets of the locator are concatenated in the order of the corresponding gram values. We can compress ordered number sequences by run-length encoding[16] with a compression ratio α_s. Therefore, we estimate the size of the leaf to be $\alpha_s u_s n (w_g + w_p)$[byte].

As for the locator, since we do not repeatedly save the document number when a gram value occurs more

than once in a document, the locator is unified in the ratio of u_d. Since the occurrence of gram values in a document is less frequent than in the whole text, $0 < u_s \leq u_d \leq 1$. Again, since the document numbers are in ascending order in a bucket, we can compress the leaf in the ratio of α_d using run-length encoding. Thus we estimate the size of the locator to be $\alpha_d u_d n w_d$[byte].

Although we can compress the root by run-length encoding, we ignore its space cost because the root is far smaller than the leaf.

From the above, we estimate the total space cost to be

$$n\{\alpha_s u_s(w_g + w_p) + \alpha_d u_d w_d\} \text{ [byte]}. \quad (1)$$

3.2 Construction Cost

We estimate the construction cost as the number of block accesses required to make an index. First we make batches, which are pairs of a gram value and a document number, and put these in the work space of the main memory. Then we merge these batches to make an index[15]. Each batch is also sorted, unified and coded by run-length encoding.

We first estimate the cost to make the batches. $2n/B$ blocks are accessed in order to read the whole text assuming 1 character is 2 bytes. After each batch is made in main memory, it is stored in secondary memory batch by batch. Although a batch has no root since it is not used for retrieval, the leaf and locator have the same structure as in the index. We estimate the cost to make the index as

$$n\{\alpha'_s u'_s(w_g + w_p) + \alpha_d u_d w_d\}/B, \quad (2)$$

where α'_s and u'_s ($0 < \alpha_s \leq \alpha'_s \leq 1, 0 < u_s \leq u'_s \leq 1$) are respectively the batch's unification ratio and its compression ratio due to run-length encoding.

The cost to merge batches is the sum of the cost to read batches (equation (1)/B) and to write an index (equation (2)).

From the above, we estimate the total construction cost to be

$$n\{2+(\alpha_s u_s+2\alpha'_s u'_s)(w_g+w_p)+3\alpha_d u_d w_d\}/B. \quad (3)$$

3.3 Retrieval Cost

The retrieval algorithm and space cost for a tree structured index are described in detail in [12]. We rewrite the equations for retrieval cost found in that paper using the symbols defined in Section 2.

[A] l_k just fits in w_g

$$1 + \lceil M w_d/B \rceil, \quad (4)$$

where M is the number of matches for the retrieval key.

[B] l_k is coded shorter than w_g

$$\lceil \sum_i (w_g + w_P)/B \rceil + \lceil \sum_i M_i w_d/B \rceil. \quad (5)$$

$\sum_i$ means summation over all grams which have the key as prefix. M_i is the number of matches for the i-th matching gram.

[C] l_k is coded longer than w_g

$$\sum_j 1 + \sum_j \lceil M_j w_d/B \rceil. \quad (6)$$

$\sum_j$ means summation over all grams which are substrings of the key. M_j is the number of matches for j-th component gram.

4 Gram Coding

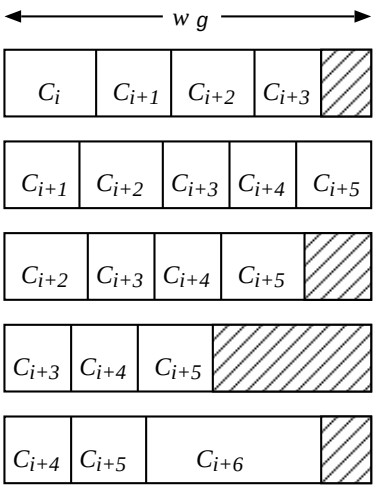

Figure 2. Example of Gram Coding. (Shadowed portions are filled with bit 0)

A character which appears very often is not effective as a filter. Therefore, we will give it a short length for gram coding in order to put more characters in a gram. Conversely, since a rare character is a very effective filter, we give it a longer length. Since we code grams of a fixed length for fast retrieval, the number of characters in a gram is variable. For an equal length of gram value w_g, the proposed method will lower the chance of false drops compared with earlier methods. Conversely, if we allow the same false drop rate, we can choose a smaller w_g. That is, we expect that we can make a more compact index with no false drop retrieval in practice. Figure 2 shows an example of gram coding for a string of $c_i c_{i+1} c_{i+2} c_{i+3} c_{i+4} c_{i+5} c_{i+6}$.

The space, construction and retrieval cost is the same as in Section 3. However, notice that u_s, u_d, α_s and α_d change depending on the coding.

5 Models and Analysis

In order to estimate the false drop rate of retrievals with an index whose grams are coded in the manner of Section 4, we analyze selectivity using a model.

5.1 Models for Characters and Strings

We assume no correlations among characters in the text and in the key. We make a coding table for each character in the alphabet. The Huffman code[17] is used to code characters with reference to their frequency of appearance in the text. We assume this coding can be done nearly optimally, i.e. we can code a character c_i, whose appearance ratio is $p(c_i)(\leq 1)$, in $-\log_2 p(c_i)$[bit].

The average code length of all the characters in the text is represented by $l_c = \overline{-\log_2 p(c_i)}[bit]$. Then the average selectivity per character becomes $1/2^{l_c}$ which is nearly equal to $1/\sigma$ by assumption, i.e. $l_c \simeq \log_2 \sigma$.

5.2 Analysis

To simplify the analysis, we use an average code length of l_c for coded characters.

5.2.1 Selectivity of Grams

A gram is $8w_g$[bit] long. However, the effective bits are reduced to $8w_g - l_c/2$ because $l_c/2$ is a fraction when we pad characters coded with length l_c. Let l_g[bit] denote this effective gram length. Since we assume there are no correlations among characters, the selectivity of a gram becomes

$$1/2^{l_g}. \tag{7}$$

5.2.2 Selectivity of the Key

[A] $l_k \leq l_g$

Since the coded key can fit inside a gram, the selectivity does not depend on l_g. It is estimated directly from l_k.

$$1/2^{l_k}. \tag{8}$$

There are no false drops in this case.

[B] $l_k > l_g$

$n_k = l_k/l_c$ characters are coded in a key, and $n_g = l_g/l_c$ characters are coded in a gram on average. Therefore, a key is divided into $n_k - n_g + 1$ grams on average. We retrieve all grams and compute the set product of all the document number sets so obtained.

For example, we will explain the case when $n_k = 5$ and $n_g = 3$[1]. If $k = c_1c_2c_3c_4c_5$, a set of grams becomes $\{c_1c_2c_3, c_2c_3c_4, c_3c_4c_5\}$. To simplify the analysis, we assume $c_1, c_2, c_3, c_4,$ and c_5 are different from one another.

[1]We intentionally use a small n_g to simplify the analysis.

Using the index, we get document numbers as retrieval results. Since we do not store the starting positions of grams in a document, those documents which contain not only (0)$c_1c_2c_3c_4c_5$ but also (1-1)$c_1c_2c_3$ + $c_2c_3c_4c_5$, (1-2)$c_1c_2c_3c_4$ + $c_3c_4c_5$ or (2)$c_1c_2c_3$ + $c_2c_3c_4$ + $c_3c_4c_5$ are retrieved[2]. Needless to say, the cases (1-1),(1-2) and (2) are false drops. (1-1) and (1-2) are strings which we can make from the key by dividing at one point. Similarly case (2) is made by dividing the key at two points. The selectivities are (0)$1/2^{5l_c}$, (1-1)(1-2)$1/2^{7l_c}$, and (2)$1/2^{9l_c}$ respectively.

In general, the selectivity when the key really is in the text is (0)$1/2^{n_k l_c}$. The selectivity when the key is divided at one point is (1)$1/2^{n_k l_c} \times 1/2^{(n_g-1)l_c}$ and there are ${}_{n_k-n_g}C_1$ such cases. The selectivity when the key is divided at $i(\leq n_k - n_g)$ points is $(i)1/2^{n_k l_c} \times 1/2^{i(n_g-1)l_c}$ and there are ${}_{n_k-n_g}C_i$ cases. Consequently, the total selectivity is the sum of the above.

$$\begin{aligned} &1/2^{n_k l_c} \times \sum_{i=0}^{n_k-n_g} {}_{n_k-n_g}C_i/2^{i(n_g-1)l_c} \\ &\quad = 1/2^{n_k l_c} \times \{1 + 1/2^{(n_g-1)l_c}\}^{n_k-n_g}. \end{aligned} \tag{9}$$

We have applied the binomial theorem to the summation. Except for the case (0), these are false drops, and their probability is

$$1/2^{n_k l_c} \times \{(1 + 1/2^{(n_g-1)l_c})^{n_k-n_g} - 1\}. \tag{10}$$

Therefore, if the above equation $\times n_i$ is far smaller than 1 for $1 \leq i \leq m$, false drops are negligible in practice in our model.

5.3 Limitations of Models

In Section 5.1, in order to simplify the analysis, we assumed that there was no correlation between the characters in the text and those in the key. However, in practice there is correlation among these characters. Therefore, the analytical results do not necessarily reflect real key searches quantitatively. However, based on the analysis we know qualitatively how key and gram length influence the selectivity and the false drop rate. As mentioned before, the code given to each character is determined by its frequency of appearance. The higher the frequency, the shorter the length of the code assigned. Since punctuation characters such as space, period, comma etc. have weak correlation with the strings which precede and follow them, we know that longer codes are appropriate for these characters.

[2]Strings connected by '+' are found separately in the same document.

6 Experimental Results

The computer used is COMPAQ DS-20(CPU: Alpha 21264 500MHz, Main Memory: 4Gbyte and Hard Disks: 8msec average seek time, and 3 msec average latency time). We got a target text by concatenating the files named ntc2-j0g and ntc2-j0k of NACSIS Test Collection 2(NTCIR-2). We extracted Accessing Number(ACCN), Title(TTL), Abstract(ABST) and Keywards(KYWD) from a target text. We made two n-gram based indices, one from TTL and the other is from ABST and KYWD. The size of indices are 95Mbyte and 1,329Mbyte respectively. The time to make indices are 266sec and 4,811sec respectively, 85min in total. We set $w_g = 6$. Table 1 shows the distributions of the number of characters in every gram for two indices. ACCN was put on a separated file and used as a translation table from document numbers which are counted from the top of a target text. Its size is 7Mbyte. Then the total size for retrieval is 1,431Mbyte. Including extraction step, the total time to make indices is about 100min.

Table 1. Distributin of the number of characters in every gram ($\times 10^6$)

	2	3	4	5	6
TTL	1.0	1.2	6.8	10.6	0.47
ABST & KYWD	1.5	14.0	121	205	17.8

We made queries from given topic files (topic-j101-150 and topic-w101-150). Keys for indices search are words in Description and Concept field of the above files. Using topic-w101-150 in which compound words are segmented in words, every possible combinations of words are made of a compound word.

We search keys from Description by using TTL index, and keys from Concept by ABST-KYWD index. Then we ranked documents, which are retrieved by search keys for each topic, using probabilistic model[18] written in a Perl program. Processing time to search keys and to rank documents for all 49 topics is 1,034sec, then the average is 21sec.

7 Conclusions

We experimented our long gram based indices at NTCIR-2. Since our indices are based on grams, no analyses such as morphological ones are required. However, we used NTCIR-2 segmented topic data in order to search words and compound words. We made indices from titles, abstracts and keywords of NTCIR-2 documents. The total index size is 1.43Gbyte and time to make indices is about 100 minutes. Average retrieval time per topic takes 21 seconds. Ranking algorithm used is based on a traditional probability model, and the result is standard average precision.

References

[1] Gonnet, G., Baeza-Yates, R. and Snider, T., New Indices for Text: Pat Trees, in *Information Retrieval: Data Structure & Algorithms* chapter 5, Frakes, W. and Baeza-Yates, R. Ed., pp. 66–82 (1992).

[2] Shang, H. and Merrett T., Trees for approximate string matching, *IEEE Trans. Knowledge and Data Eng.*, Vol. 8, No. 4, pp. 540–547 (1996).

[3] Itoh, M., An Efficient Method for Constructing Suffix Arrays of Large Texts, *IPS Japan SIG Notes*, 99-NL-129-5 (1999).

[4] Yamashita, T., Fujio M. and Matsumoto Y., Language Independent Tools for Natural Language, *Proc. 18th ICCPOL*, pp.237–240 (1999).

[5] Ferragina, P. and Grossi, R., Fast string searching in secondary storage: Theoretical developments and experimental results, *Proc. ACM-SIAM Symposia on Discrete Algorithms*, Vol. 7, pp. 373–382 (1996).

[6] Ogawa, Y. and Iwasaki, M., A new character-based indexing method using frequency data for Japanese documents, *In Proc. 18th ACM SIGIR Conf.*, pp. 121–129 (1995).

[7] Sugaya, N. *et al.*, A full-text search system for large Japanese text bases using n-gram indexing method, *Proc. 53th Annual Convention IPS Japan*, 5T–2,3 (1996).

[8] Akamine, S. and Fukushima, T., Flexible string inversion method for high-speed full-text search, *Proc. Advanced Database Symposium '96* (1996).

[9] Matsui K., Namba, I. and Igata, N., Full-text searching engine for large-scale data, *Proc. 1997 IEICE General Conference*, D–4–6 (1997).

[10] Kikuchi, C., A fast full-text search method for Japanese test database, *Trans. IEICE*, Vol. J75-D-1, No. 9, pp. 836–846 (1992).

[11] Sato, T., Fast full test search with free word using TS–file, *Proc. 19th ACM SIGIR Conf.*, p. 342 (1996).

[12] Sato, T., Fast full test retrieval using gram based tree structure, *Proc. ICCPOL '97*, Vol. 2, pp. 572–577 (1997).

[13] Sato, T. *et al.*, Gram based full test search system and its application, *IPSJ SIG Notes*, 98–DBS–114–2 (1998).

[14] Knuth, D.: *The Art of Computer Programming: Vol.3 Sorting and Searching, 2nd Ed.*, Addison-Wesley, Reading, Mass., pp. 481–489 (1998).

[15] Noda, J., Endoh, A. and Sato, T., Creation and update of indices for a n-gram based fill text retrieval system, *Proc. IEICE DEWS'98* (1998).

[16] Zobel, J., Moffat, A. and Sacks-Davis, R., An efficient indexing technique for full-text database, *Proc. 18th Int. Conf. on VLDB*, 1992, pp. 352–362.

[17] Huffman, D.A., A method for the construction of minimum-redundancy codes, *Proc. IRE*, Vol. 40, pp. 1098–1101 (1952).

[18] Robertson, S.E. and Walker, S., Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval, *Proc. 17th Int. Conf. Research and Development in Information Retrieval*, pp. 232–241 (1994).