A New Character-based Indexing Method using Frequency Data for Japanese Documents

OGAWA Yasushi and IWASAKI Masajirou
{ogawa,iwasaki}@ic.rdc.ricoh.co.jp
Information and Communication Research Center, RICOH Co., Ltd.

Abstract

A character based indexing is preferable for Japanese IR systems since Japanese words are not segmented. This paper proposes a new character indexing method to enhance our previous method which divided character pair index entries into disjoint groups based on character classes. Since frequency data is used to determine hashed entries for character pairs and to establish a special string index, both search speed and precision are improved. Moreover, bit strings are managed using small and large blocks, so registration and retrieval are accelerated. Experiments using patent abstracts showed these proposals are quite effective.

1 Introduction

Although word-based indexing is widely used in many languages such as English [19], it is not easily applied to Japanese. This is because Japanese is an agglutinative language with no apparent separation between words, and thus compound words are easily generated to express complex objects and ideas. As for handling compound words, there are two methods [5][15]; one is to treat a compound as an inseparable unit and the other is to divide it into its component words. The first method, where compounds are easily identified by parsing texts based on a rather limited number of functional words and notational and grammatical rules, cannot retrieve documents with compound words which only partially match to the query word. In the second method, partial matching can be easily carried out since the components have already been identified during registration, but identifying component words is a problem because a large lexical dictionary is necessary which makes it difficult to set up and maintain a retrieval system.

On the other hand, character-based indexing methods such as n-gram indexing don't depend on word segmentation and allow partial matching from the beginning [17][20]. This is a suitable method for Japanese texts, and has been widely used in retrieval systems for Japanese documents [14]. These methods sometimes retrieve falsely matched documents **(false drops)**, but these false drops can be reduced by increasing the number of distinct entries (keys) [1][4]. To attain a practical level of precision, pairs of adjacent characters, i.e. 2-grams, are used as index entries in most Japanese retrieval systems [13]. However, since the number of distinct character pairs is huge due to the large character set (about 7,000 characters), it is desirable to reduce distinct pairs [6][8][11].

We have previously proposed character-based indexing using both single character and character pair entries [10][16]. A novel feature of our method was that character pairs were hashed so that pairs of different character classes (characters are clustered into several classes in Japanese) never share the same hashed value. Although the method achieves get better precision with a relatively small number of entries, there are two problems. One is performance degradation due to the use of both single character and character pair indexes. The other problem is performance degradation due to the fact that the bit string for each entry's occurrences in registered documents is stored using fixed length blocks. This paper proposes performance enhancement methods to solve these problems.

This paper is organized as follows: Section 2 explains the character classes of Japanese for readers unfamiliar with Japanese. Section 3 describes our previous method and the problems associated with the method. Our proposals are described in the following two sections. Section 4 details a new index entry organization based on frequency information, and Section 5 presents an effective method of managing bit strings. Evaluation results are reported in Section 6. Section 7 concludes the paper.

2 Japanese Character Classes

We will briefly describe the different classes of Japanese characters for the benefit of those who are not familiar with the language.

In Japanese, there are several classes of character including Kanji, Katakana and Hiragana, each of which has different functions. Kanji is based on Chinese ideograms. Kanji characters may be used individually to form words with simple meanings, or strung together to form words with more complex meanings. JIS (Japanese Industrial Standards) have designated 6353 characters for general use, but in fact we use only about 1000 of them in a daily life. Hiragana and

Permission to make digital/hard copies of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/ server notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the Association for Computing Machinery, Inc. (ACM). To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
SIGIR'95 Seattle WA USA© 1995 ACM 0-89791-714-6/95/07.$3.50

Table 1: Kanji and Katakana words' statistics

	Kanji		Katakana	
length	num. of dist. words	total frequency	num. of dist. words	total frequency
1	1120	256062	26	93
2	9983	514645	275	8889
3	23552	151983	944	83976
4	39995	148174	1401	48238
5	24717	48746	1557	23111
6	18583	30888	2526	20036
7	8533	11790	2688	12360
8	4692	6413	2030	6397
9	2075	2641	1529	4246
10	960	1124	976	2353
over 10	846	991	1729	2789

Katakana are phonetic characters which represent about 80 different sounds. Hiragana is used for particles, auxiliary verbs, and conjugational parts, while Katakana is mainly used to represent words from foreign languages.

In Japanese, words in queries are mainly nouns and verb-heads, which consist of Kanji or Katakana. Their features, however, differ in accordance with their functions and the character set size. Kanji words are generally rather short, and many distinct words appear in Kanji while the average frequency of each word is small. Katakana words, on the other hand, are much longer because they are the phonetic representations of loan words. The number of distinct words is small but the average frequency of each word is high.

Table 1 shows the numbers of Kanji and Katakana words appearing in a total of 100,000 different abstracts on Japanese Patents (about 14M bytes in total). The averages of word length and frequency [1] are 2.51 and 8.69 for Kanji, and 4.39 and 13.55 for Katakana. These values confirm the characteristics described above.

3 Character-based Indexing

3.1 Introduction

Character-based indexing is an access method used in large document databases, which treats documents as a string of characters rather than words [1]. For example, an n-gram index [2][9] records occurrences of all n-grams, successive overlapping strings of n characters, in document. When a query is performed, other n-grams are extracted from the query and the qualifying documents are quickly obtained as the intersection of document sets, each of which corresponds to each n-gram from the query. Since words are not segmented, many Japanese document retrieval systems adopt this method [14].

[1] "Total frequency" in Table 1 means the count of all words of a particular length, and "word frequency" is the count of each word. Word frequency is given by dividing total frequency by the number of distinct words.

One disadvantage of character-based indexing is that it sometimes creates false drops. This is because simple index systems such as an n-gram index discard positional information of the entry's occurrence. To exclude false drops, retrieval systems have to scan retrieved documents to check whether they actually qualify for the query. It is, therefore, desirable to reduce false drops so as to decrease the number of documents to be scanned. Note that there is in general an inverse relationship between the false drop rate and the number of distinct entries [1][4].

Although Japanese has a large number of distinct characters, a single character index cannot attain sufficient retrieval precision [7][12]. Indexing any pair of successive characters, however, is not a solution, either. It generates too many index entries; approximately $49 \times 10^6 (= 7,000^2)$ entries [13]. Therefore, hashing must be applied to character pairs to reduce the number of index entries [6][8][11].

3.2 Our Previous Method

Our previous character-based indexing method [10][16] uses single character (1-gram) and character pair (2-gram) indexes. In order to reduce the number of pair entries, character pairs are hashed so that pair entries are divided into several groups according to their character classes. We name the method **character class divided hashing.**

This idea came from the following two considerations. First, since the possibility that characters are used in queries varies depending on their character class, desirable false drop rates also vary. In addition, as the frequency and character set size differ with class, it is desirable to change a hashing scheme based on character classes. If not, a hash method adjusted for the worst case has to be used, but the number of hash entries becomes too large. Second, since a hashed entry may be shared by character pairs of different classes, rarely occuring Kanji pairs may be falsely matched with frequently occuring Katakana pairs. This kind of false matching can be excluded by grouping index entries based on character class.

The following is a hashing procedure for character pairs. At first, a hashed value is obtained by applying a hash function to each character in a pair. Then, an entry value is computed from a pair of these hashed values. In computing the entry value, disjoint ranges are assigned in accordance with classes of the former and latter characters. When the former and latter characters are represented as c_f, c_l; the hash function for class i as h_i (h_i outputs a value between 0 and $d_i - 1$), an entry value eid is given by:

$$eid(c_f, c_l) = \begin{cases} h_1(c_f) + h_1(c_l) * d_1 \\ \qquad : c_f, c_l \text{ in CLASS1} \\ h_1(c_f) + h_2(c_l) * d_1 + d_1^2 \\ \qquad : c_f \text{ in CLASS1}, c_l \text{ in CLASS2} \\ \ldots \\ h_2(c_f) + h_2(c_l) * d_2 + d_1^2 + \cdots \\ \qquad : c_f, c_l \text{ in CLASS2} \\ \ldots \end{cases}$$

As the number of hashed entries for character pairs decreases so as to reduce the size of the index, the false drop rate increases. Thus, a single character index is combined with the character pair index so that the index size may be kept small with a low false drop rate.

To record entry's occurrences in documents, an entry is associated with a bit string in which the position of a bit corresponds to a document, and bit strings for all entries are stored in a file. Each bit string is compressed to reduce the data size and to increase retrieval speed, and the compressed bit string is stored in fixed length blocks which distribute among the file.

3.3 Problems

There are two problems associated with our previous method.

- Performance degradation due to the query length:

  Retrieval time is nearly proportional to the length of query, as retrieval is carried out by character basis. Our method uses both single and pair indexes, so $2m - 1$ entries (m single plus $m - 1$ pair entries) have to be processed for an m character query.

  To address this problem, we propose a new index organizing method (See Section 4).

- Performance degradation related to block size:

  The block size of the bitmap file influences both the insert and retrieval performance, but in opposing ways. In registration, [2] each bit for the extracted entries is written in the tail block of the corresponding entry. Thus, the smaller the block is, the faster the insertion. In retrieval, all the blocks need to be randomly accessed because they are distributedly placed. Because retrieval time is proportional to the number of blocks to be accessed, performance can be improved by enlarging the blocks, thereby reducing the number of blocks. In other words, the insertion and retrieval performances have an inverse relationship to the block size. [3]

  To solve the problem, we have developed a novel bit string management method using both small and large blocks. Section 5 explains the detail.

4 New Character Index Method

As for index organization, we propose a new hashing scheme for character pairs and an effective index whose entries are character strings with more than 2 characters.

[2]We assume each document is inserted in the database when the document is created (on demand insertion).

[3]Since the number of entries is large in our character-based index, blocks are preferable to be small so as to decrease unused areas in blocks.

4.1 New Hashing Scheme for Character Pair Index

4.1.1 Frequency-based Hashing

False drops can be reduced by flattening the distribution of index entry's frequencies in character-based indexes [1][17] and signature files [3][18]. For example, Barton *et al* has proposed character-based indexing that does not use fixed length n-grams for index entries but selects variable length strings to flatten the entry frequency distribution, resulting in better precision [1].

Because the above method is for ASCII codes, it is impractical to apply it directly to Japanese with its numerous characters. However, the idea is applicable to a hashing scheme for character pair indexing in the following way. Since a conventional simple hashing determines a hashed entry from a character code which has no relationship to frequency, the entry frequency, i.e. the sum of frequencies of assigned characters to each entry, varies greatly according to changes in character frequency. Conversely, given the character frequencies, entries can be assigned to characters so that hashed entries are equally balanced by entry frequency. We call the conventional method **code-based,** and the other method using character frequency **frequency-based.**

Compared to the code-based method, because the frequency-based generates a more flat distribution of entry frequencies in hashing character pairs, it attains higher retrieval precision [7][11].

4.1.2 Frequency-based Character Class Divided Hashing

Since frequency-based hashing is effective in improving precision, we adopted it instead of code-based hashing in our character class divided method for computing entry values. Given the same number of hashed entries, the new method attains better precision than before.

In addition, adopting frequency-based hashing has another significant advantage, i.e. faster retrieval. In our previous method, the use of a single character index after introducing a character pair index was not to decrease search precision; one hashed entry is always shared by more than two characters when a simple hash function is applied to character codes, so a single character index is necessary to judge whether each character in the query actually exists in documents. However, there are entries that are monopolized by single characters in frequency-based, because character frequencies vary a lot in Japanese. Such a hashed entry guarantees that a document surely contains a corresponding character, eliminating the need to use a single character index for such characters. Processing m character queries only require access of $m - 1$ entries when all characters are of the monopolizing type. Even in the worst case, the number of entry accesses is $2m - 1$ entries which is the same number of entries in the old method. The fewer accesses result in better performance.

4.1.3 Hash Table Generation

A hash table which maps characters to entries is necessary to enable frequency-based hashing. In our retrieval method, each character class needs hash tables. [4] However, since our method calculates entry value from a pair of hashed values of adjacent characters, the total size of hash tables is much smaller than the size of a table which determines entry values directly from character pairs.

To generate a hash table for a particular class, we first need to determine hashed entry size. Then, depending on the frequency of each character in the class, entries are assigned to characters to make the entry frequencies as even as possible. Because the problem is NP-complete and takes a great deal of time to find the optimum solution, we adopt a simple heuristic algorithm as shown below.

1. Sort characters in descending order of frequencies.
2. Assign the most frequent remaining characters to the entry with the fewest sum of frequencies of the already assigned characters. Removes the character, and repeats the step until all characters are processed.

We show, by experimental results, how well this simple algorithm works. We counted the frequency of a total of 6,353 characters in the patent abstracts mentioned in Section 2. 4,395 of all Kanji characters never appeared, 689 only once, and 1,452 more than 100 times. The most frequent character appeared 73,747 times. When all Kanji characters are hashed or binned into 128 entries according to frequencies. The largest bin contained 73,747 occurrences and the smallest, 22,375 occurrences. On the other hand, using simple code-based method, the largest bin contained 99,881 and the smallest, 2,212.

The retrieval system must be able to know whether an entry is monopolized by one character or not during retrieval. To indicate this, a flag is prepared for each entry, and set on or off during the hash table generation.

Note that it is not always necessary to prepare the whole document set in advance, as precision is not so degraded using hash tables made from a part of it. Refer to the experimental results shown in Section 6.

4.2 Frequent String Index

4.2.1 Introduction

Even after the above improvement was made, at least $m - 1$ entries have to be accessed for m character queries. Although it is possible to speed up the retrieval by employing longer character strings as index entries, it is impractical to apply this method to Japanese because characters are so numerous.

We, therefore, adopt the previously mentioned Barton's idea [1] of selecting variable length strings with high frequency, but modify it for our character based indexing.

[4]Because most characters in queries are either Kanji or Katakana as mentioned before, it may be sufficient to prepare hash tables only for them.

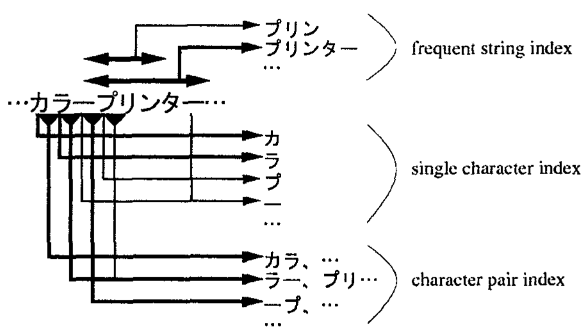


Figure 1: Overall Indexes Organization

First, only strings with more than 2 characters are chosen for index entries because our method already has single and pair indexes. Second, we count the frequency of only the sequences of the same class of characters, specifically Kanji or Katakana, because Japanese has several character classes but queries generally contain only Kanji or Katakana. Compared to counting all possible sequences, which requires large memory, this method is preferable since it is space efficient.

As the number of string entries increases, the retrieval is carried out quickly because the probability that a query contains the string becomes high. However, the string matching for the entry extraction requires more memory and slows down, and additional bit strings need to be stored in the data file. The number of the string entries is, therefore, limited to several hundreds in our method. In this way, a special index named the **frequent string index** is formed to speed up the retrieval.

The frequent string index seems similar to word-based indexing, but different in that :

- An entry string is not necessarily a word, but may be a meaningless character string, which can be easily found in documents or queries using a simple string match without grammatical parsing.
- Although new words are dynamically added to indexing entries in the word-based method, entry strings are a static set of character strings that are fixed when a document database is created. Entries, therefore, can be managed in a simple manner.

4.2.2 Document Registration and Retrieval

During document registration, we can scan the target document to find all the strings defined in the string index before extracting conventional character entries. It sometimes happens that an entry string is completely included within another entry string. In such cases, both entries are recorded in the index.

For example, when both "プリン"(*pu-ri-n*: pudding) and "プリンタ"(*pu-ri-n-ta*: printer) are defined as the entry strings, and the document contains "カラープリンタ"(*ka-ra-a-pu-ri-n-ta*: color printer), both entries are extracted, while the conventional character indexes are updated in the same way as before. In other words, we record single entries (カ, ラ, ー, プ, ⋯) and pair entries (hashed results from カラ, ラー, ープ, ープ, ⋯) as shown in Figure 1. Lines correspond to extracted entries, which are categorized into three groups, i.e. string, single character and character pairs indexes. Note that only the Katakana parts of indexes are shown here.

In document retrieval, we use similar methods as in document registration. The difference is the treatment of included entries. When an entry string is included within another string, the former is discarded. Moreover, conventional entries included in the extracted string entry are not used during retrieval. The difference is illustrated in Figure 1. Among possible entries, only entries not contained within other entries are extracted, as indicated by the thick lines.

5 New Bit String Management using Blocks of Two Sizes

5.1 Introduction

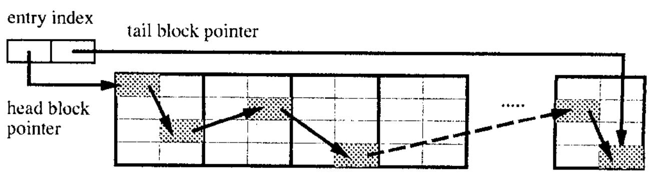
As for the block size problem in implementation, we propose a new bit string management method where bit strings are stored in fixed length blocks of two sizes. We call smaller blocks **buckets** and larger ones **containers.** Containers are the same size as or a few times larger than a disk page. The container size is an integral multiple of the bucket size. [5]

We use small buckets in registration to achieve high speed insertion. After many documents are registered, the retrieval performance is degraded as a larger number of buckets are used to store bit strings. At this point, we invoke a file reconfiguration so that the buckets are gathered to form containers. By processing blocks one entry after another, the merged containers for an entry's bit string are written to the disk sequentially. This operation is named a **block reconfiguration,** as illustrated in Figure 2. Hatched areas represent blocks of an entry's bit string, and small and large rectangles are buckets and containers respectively. After the block reconfiguration, a bit string can be accessed sequentially by larger containers, resulting in improved search response. In this way, the bit string management using two sized blocks with the block reconfiguration enables high performance in both registration and retrieval.

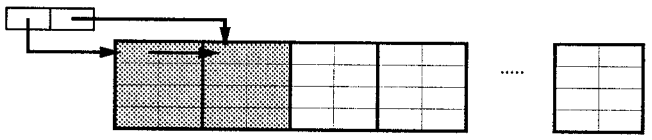
5.2 Block Reconfiguration Algorithm

As mentioned above, buckets are merged into containers one entry after another. However, several buckets may not fill a container, so they are written back into the disk

[5]In our implementation, the container and bucket sizes are usually set to 1K and 64 bytes.



(a) Distributed buckets before block reconfiguration



(b) Sequentially placed containers after block reconfiguration

Figure 2: Block Reconfiguration

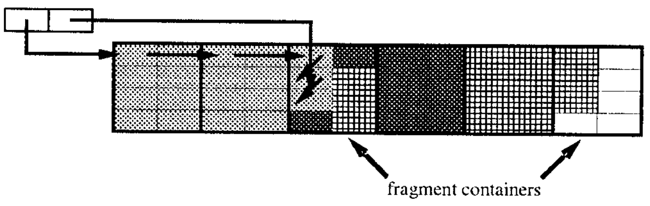


Figure 3: Fragment Containers

as buckets. Containers have to be located so that the beginning of the container agrees with the beginning of disk page for fast access. The remaining buckets are therefore packed in a special **fragment container.** Since one entry's remaining buckets cannot fill a fragment container, it is shared by several buckets of different entries. On the other hand, remaining buckets of an entry occasionally use two fragment containers. Figure 3 shows fragment containers; the left container includes buckets from three entries while remaining buckets of the entry shown by small square hatch are stored in both containers. In summary, after the reconfiguration, there are two kinds of containers: one is a normal container that is filled with data from one entry, and the other is a fragment container that stores several buckets from more than two entries.

Each entry is processed as follows :

1. Prepare two container buffers on memory, one for a normal and the other for a fragment, and create a temporary file.
2. Process all entries one by one as follows :
    (a) Read one bucket from the original file, and copy the data into the normal container buffer.
    (b) When the normal buffer is filled with the data, write the filled buffer to the temporary file, clear the buffer, and repeat the step (a).
    (c) When all the buckets are read, copy the remaining data in the normal buffer to the fragment buffer. If the fragment buffer is filled with data, write it to the temporary file and clear it.

3. Write the fragment buffer if it contains data.
4. Free the buffers and replace the original file by the temporary file.

6 Evaluation

6.1 Evaluation Method

We used the abstracts of Japanese patents which we also used for word counting. Queries were randomly selected from Kanji and Katakana strings extracted from the abstracts. We prepared 30 queries of length 2, 4, 6, 8 and 10, resulting 300 (=30x5x2) queries. For evaluating retrieval precision, we used the false drop rate FDR and the precision rate PRE [6] computed as follows :

$$FDR = \frac{\#(\text{retrieved but irrelevant docs})}{\#(\text{irrelevant docs})}.$$
$$PRE = \frac{\#(\text{retrieved and relevant docs})}{\#(\text{retrieved docs})}$$

Sun SPARCstation 20 model 50 running SunOS 4.1.3 was used in evaluation. Response time was measured in the **cold start** condition where no data is cached on memory.

The new bit string management method with the block reconfiguration was at first evaluated, then was the new index organization. We thought it would be better to evaluate the retrieval speed up caused by the new index organization using the new file structure, so the implementation issue was tested first.

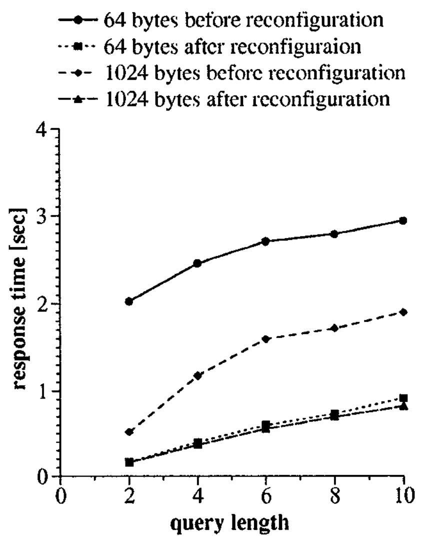
6.2 Baseline Performance

First of all, we show the performance for the previous method, which divides hashed entries into 6 classes, including Kanji and Katakana, and adopts the same code-based hashing which computes modulo 32 of character code for all classes.

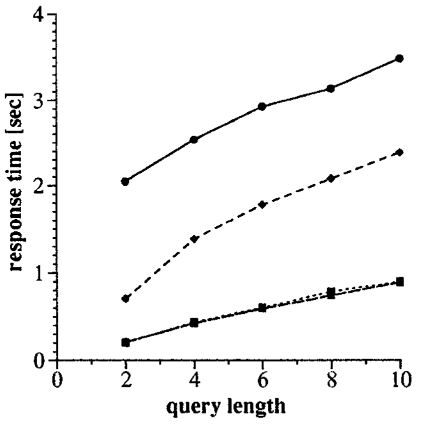
Registration and retrieval time was measured. To register all 100,000 abstracts, it took 1h 48min (1,130 char/sec) with the block size of 64 bytes, and 2h 26min (840 char/sec) with the block size of 1024 bytes. As expected, the smaller the block size is, the faster the registration is. On the other hand, the average search response time over the 300 queries was 2.71 sec for 64, and 1.53 sec for 1024. The smaller blocks slowed down the retrieval. This clearly demonstrated the problem related to the block size.

Next we checked the effect of the query length on retrieval speed. Figure 4 shows the relationship between the query length and response time. The two upper lines represent the response time before block reconfiguration. We found that the response time increased with the query length.

[6]Precision is usually used with recall, but these character based indexing methods always find the documents containing the query string. Thus we only use precision here.



(a) Kanji



(b) Katakana

Figure 4: Query Length vs. Response Time

6.3 Evaluation of New Bit String Management

After the baseline performance profile was outlined, the effects of the block reconfiguration was evaluated. The container size was set to 1024 bytes. To reconfigure, it took 167 sec with 64-byte buckets, and 110 sec with 1024 bytes. We found that the reconfiguration required only a fraction of registration time (about 2.5 % at 64 bytes). The larger block took less time because the number of blocks to be accessed decreased as the block size enlarged.

The retrieval response times were measured and the results are plotted on Figure 4 (shown as the lower two lines). The average response times were 0.57 sec for 64-byte buckets and 0.6 sec for 1024-byte buckets. This means the reconfiguration speeded up the retrieval by 4.75 for 64 bytes, and 2.78 for 1K bytes. These results shows the new management scheme is quite effective. It should be noted that the response time improved greatly even when the container size was same as the bucket size, implying that the performance can improve

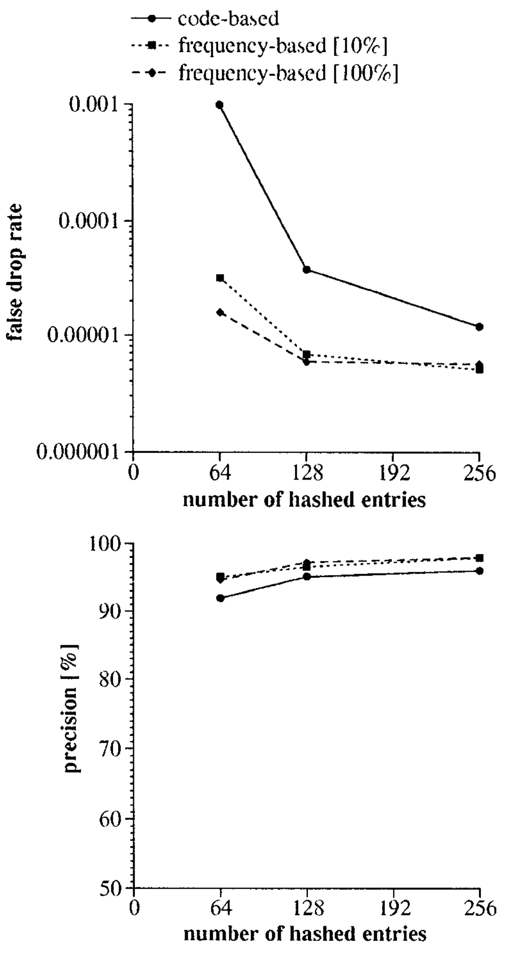


Figure 5: Retrieval Precision Improvements: Kanji

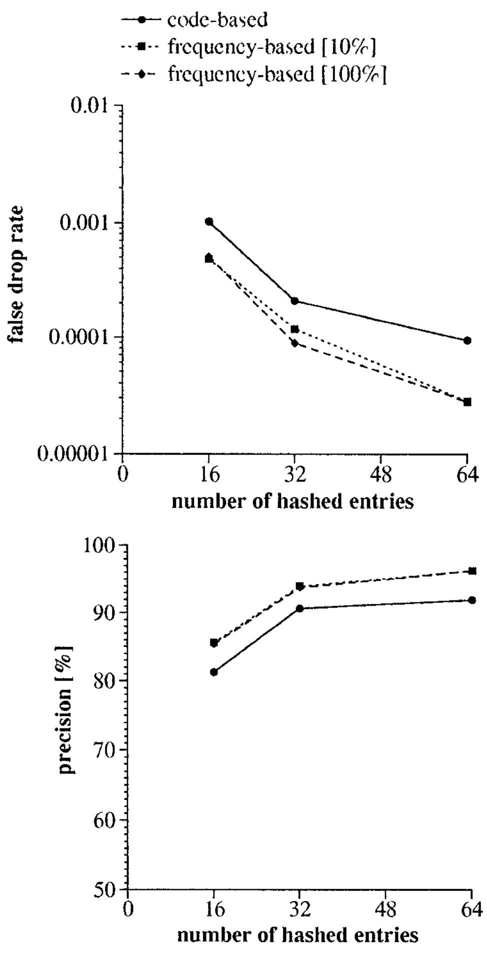


Figure 6: Retrieval Precision Improvements: Katakana

by relocating distributed blocks in successive positions. In the following experiments, the response time was measured after the reconfiguration with 1K-byte containers.

6.4 Evaluation of New Index Organization

6.4.1 Effect of Frequency-based Hashing

We evaluated the frequency-based hashing scheme. First, retrieval precisions, false drop probability and precision rate, were measured for simple code-based and novel frequency-based cases. The number of hashed entries was set at 64, 128, 256 for Kanji and 16, 32, 64 for Katakana.

Results are shown in Figure 5 (Kanji) and Figure 6 (Katakana). Lines for frequency, plotted as "frequency-based [100%]", lie under lines for code-based in *FDR* and vice versa in *PRE*. As indicated, the frequency-based method always outperforms the code-based one.

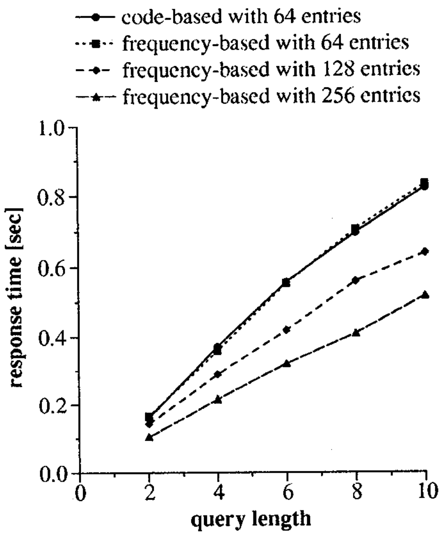
We also tested hash tables generated from only a fraction (10%) of the target document collections. Hash tables made from all the documents were used in the above experiments. Results are plotted in the same figures as "frequency-based [10%]". There are only a few differences between 10% and 100%, and therefore, the entire document collections are not always needed to prepare hash tables.

Next, the response time was measured (Figure 7). As expected, the performance was improved in both Kanji and Katakana, but the effect was not the same. The difference comes from the fact that the number of monopolized entries (see Section 4.1.2) was 2, 21, 97 out of total 64, 128, 256 entries for Kanji, while the number was 2, 14, 57 out of total 16, 32, 64 for Katakana. Compared to the character set size of 6353 for Kanji and 86 for Katakana, the fraction of monopolized entries was much larger for Katakana than for Kanji. This results in a large improvement of Katakana.

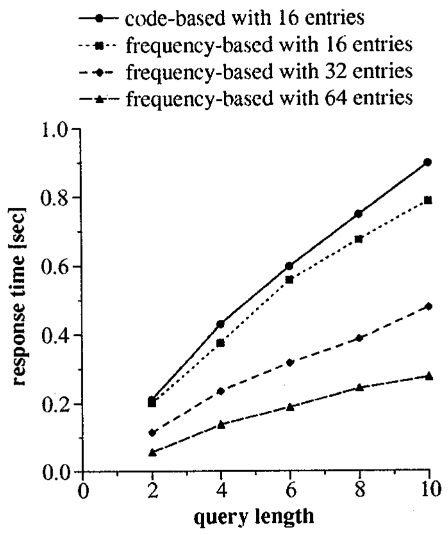
In summary, frequency-based hashing improves mainly retrieval precision for Kanji, but it is, for Katakana, effective in both precision and speed up.

6.4.2 Effect of Frequent String Index

Finally, the effect of the frequent string index for speed improvement was evaluated. In this experiment, the frequency-based hashing was again used and the number of hashed entries was fixed to 128 for Kanji, and 32 for Katakana. Again the effect of improvement is much greater in Katakana than

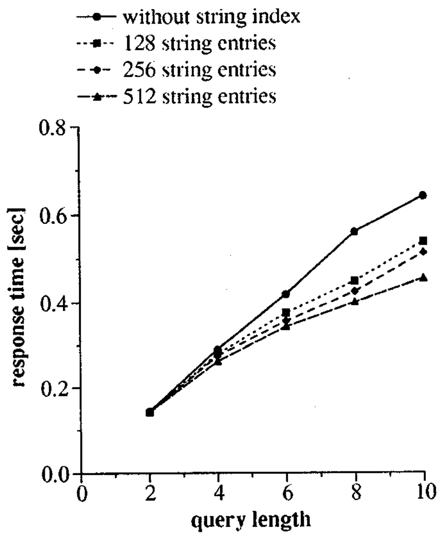


(a) Kanji

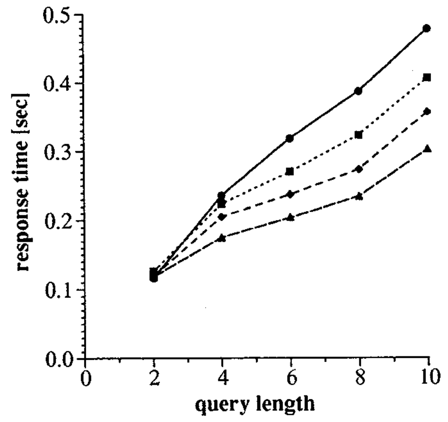


(b) Katakana

Figure 7: Speed Up by Frequency-based Hashing



(a) Kanji



(b) Katakana

Figure 8: Speed Up by String Index

Kanji (Figure 8). This is because there are fewer distinct words in Katakana than in Kanji as mentioned in Section 2, so that the probability that a long query includes one or more of the entry strings becomes larger. Considering the fact that the average word length is also longer for Katakana, the string index is effective for Katakana.

7 Conclusion

This paper proposes a new character indexing method to enhance our previous method. We applied frequency-based hashing to calculate entry values for character pairs. Using frequency data, retrieval performance improved and the false drop rate decreased. In addition, longer queries can be processed at high speed by introducing a frequent string index whose entries have frequently occuring strings with more than 2 characters. Furthermore, the new bit string management method using small and large blocks accelerates both registration and retrieval, as small blocks are used in registration while bit strings are accessed by large blocks which are placed sequentially by the block reconfiguration.

The main feature of the proposed method is that indexes are configured taking into account properties such as the character set size and frequency and the average word length of different character classes. Because Kanji and Katakana are the main character classes used in queries, indexes are organized to fit their properties. The use of frequency-based hashing for Kanji can greatly enhance retrieval precision, which generally tends to stay low since Kanji has a large number of characters and their frequencies change greatly. Although the retrieval speed is not much faster than before, it isn't a problem because the average length of Kanji words is short. On the other hand, Katakana has a rather smaller character set while its words are much longer. Therefore, frequency-based hashing is more effective for improving speed rather than precision. The frequent string index also improves the retrieval speed.

We have proposed and evaluated our indexing method for Japanese documents, but it can be applied to other languages with large character sets. For example, Chinese uses several thousand Kanji characters, Korean has a few thousand Hangul character codes in addition to Kanji.

Future work includes detailed evaluations using different types of documents to see the effect of the document length and the difference in character frequencies. In addition, we study the possibility of implementing more sophisticated retrieval model, such as vector space [19], probabilistic [22] and inference net [21], for our character indexes.

References

[1] I.J Barton *et al.* An information theoretic approach to text searching in direct access systems. *CACM*, Vol. 17, No. 6, pp. 345–350, 1974.

[2] W.B. Canvnar *et al.* N-gram-based text filtering for TREC-2. In *Proc. of TREC-2*, pp. 171–179, 1994.

[3] C. Faloutsos *et al.* Design of a signature file method that accounts for non-uniform occurence and query frequencies. In *Proc. of Int. Conf. on VLDB '85.*, pp. 165–170, 1985.

[4] C. Faloutsos *et al.* Description and performance analysis of signature file methods for office filing. In *ACM Trans. on Office Information Systems*, Vol. 5, No. 3, pp. 237–257, 1987.

[5] H. Fujii and W. B. Croft. A comparison of indexing techniques for Japanese text retrieval. In *Proc. of 16th ACM SIGIR Conf.*, pp. 237–246, 1993.

[6] Y. Fujii *et al.* A singature file create method for full-text search (*in Japanese*). In *Proc. of 48th IPSJ Conf. (4)*, pp. 159–160, 1994.

[7] T. Fukushima *et al.* A signature file compression method for full text retrieval (*in Japanese*). In *Proc. of 47th IPSJ Conf. (4)*, pp. 83–84, 1993.

[8] K. Furuse *et al.* Implementation of signature file access method in a DBMS (*in Japanese*). Technical Report DE94-58, IEICE, pp. 23–29, 1994.

[9] M.C. Harrison. Implementation of the substring test by hashing. *CACM*, Vol. 14, No. 12, pp. 777–779, 1971.

[10] M. Iwasaki and Y. Ogawa. A new character-based indexing method for Japanese texts using reduced adjacent character bitmap tables. In *Int. Symp. on Next Generation Database Systems and Their Applications*, pp. 145–150, 1993.

[11] Y. Kawashimo *et al.* Development of full text search system Bibliotheca/TS (*in Japanese*). In *Proc. of 45th JIPS Conf. (3)*, pp. 241–242, 1992.

[12] K. Kitamura *et al.* Development of CD-ROM version for classical Japanese literature text and its basic concordance system (*in Japanese*). In *Proc. of 41th IPSJ Conf. (4)*, pp. 142–143, 1990.

[13] S. Miyahara *et al.* A study of high-speed full-text retrieval using character transition probability (*in Japanese*). In *Proc. of 40th IPSJ Conf. (4)*, pp. 880, 1990.

[14] Y. Ogawa. Trends in text database studies (*in Japanese*). In *IPSJ Advanced Database System Symp.*, pp. 153–162, 1993.

[15] Y. Ogawa *et al.* Simple word strings as compound keywords: An indexing and ranking method for Japanese texts. In *Proc. of 16th ACM SIGIR Conf.*, pp. 227–236, 1993.

[16] Y. Ogawa *et al.* A new indexing and text ranking method for Japanese text databases using simple-word compounds as keywords. In *Proc. of 3rd Int. Symp. on DASFAA*, 1993.

[17] C.S Roberts. Partial-match retrieval via the method of superimposed codes. *Proceedings of IEEE*, Vol. 67, No. 12, pp. 1624–1642, 1974.

[18] R. Sacks-Davis and A. Kent. Multikey access methods based on superimposed coding technicues. *ACM Trans. on Database Systems*, Vol. 12, No. 4, pp. 655–696, 1987.

[19] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[20] N. Tavakoli and A. Ray. A new signature approach for retrieval of documents from free-text databases. *Information Procesing & Management*, Vol. 28, No. 2, pp. 153–163, 1992.

[21] H. Turtle and W.B. Croft. Inference networks for document retrieval. In *Proc. of 13th ACM SIGIR Conf.*, pp. 1–24, 1990.

[22] C.J. van Rijsbergen. *Information Retrieval.* Butterworths, 1979.

* IEICE stands for "The Institute of Electronics, Information and Communication Engineers" of Japan, and IPSJ for "Information Processing Society of Japan".