The VLDB Journal (2008) 17:1485–1507
DOI 10.1007/s00778-007-0082-x

REGULAR PAPER

Structural optimization of a full-text *n*-gram index using relational normalization

Min-Soo Kim · Kyu-Young Whang · Jae-Gil Lee · Min-Jae Lee

Received: 24 May 2006 / Revised: 11 July 2007 / Accepted: 13 August 2007 / Published online: 13 December 2007

Abstract As the amount of text data grows explosively, an efficient index structure for large text databases becomes ever important. The *n*-gram inverted index (simply, the *n-gram index*) has been widely used in information retrieval or in approximate string matching due to its two major advantages: language-neutral and error-tolerant. Nevertheless, the *n*-gram index also has drawbacks: the size tends to be very large, and the performance of queries tends to be bad. In this paper, we propose the *two-level n-gram inverted index* (simply, the *n-gram/2L index*) that significantly reduces the size and improves the query performance by using the relational normalization theory. We first identify that, in the (full-text) *n*-gram index, there exists redundancy in the position information caused by a non-trivial multivalued dependency. The proposed index eliminates such redundancy by constructing the index in two levels: the front-end index and the back-end index. We formally prove that this two-level construction is identical to the relational normalization process. We call this process *structural optimization* of the *n*-gram index. The *n*-gram/2*L* index has excellent properties: (1) it significantly reduces the size and improves the performance compared with the *n*-gram index with these improvements becoming more marked as the database size gets larger; (2) the query processing time increases only very slightly as the query length gets longer. Experimental results using real databases of 1 GB show that the size of the *n*-gram/2*L* index is reduced by up to 1.9–2.4 times and, at the same time, the query performance is improved by up to 13.1 times compared with those of the *n*-gram index. We also compare the *n*-gram/2*L* index with Makinen's compact suffix array (CSA) (Proc. 11th Annual Symposium on Combinatorial Pattern Matching, pp. 305–319, 2000) stored in disk. Experimental results show that the *n*-gram/2*L* index outperforms the CSA when the query length is short (i.e., less than 15–20), and the CSA is similar to or better than the *n*-gram/2*L* index when the query length is long (i.e., more than 15–20).

Keywords Text search · Inverted index · *n*-gram · Multivalued dependency

M.-S. Kim (✉) · K.-Y. Whang · J.-G. Lee · M.-J. Lee
Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea
e-mail: mskim@mozart.kaist.ac.kr

K.-Y. Whang
e-mail: kywhang@mozart.kaist.ac.kr

J.-G. Lee
e-mail: jglee@mozart.kaist.ac.kr

M.-J. Lee
e-mail: mjlee@mozart.kaist.ac.kr

1 Introduction

As the amount of text data (e.g., web pages and biological sequences) grows explosively, text searching has become one of the most important technologies. For efficient text searching, a number of index structures have been proposed. Among them, the inverted index, which is inherently the disk-based index structure, is the most actively used one for large databases [3,34].

The inverted index uses *words* or *n-grams* as indexing terms [32]. We call the inverted index using *n*-grams as the *n-gram index*. An *n*-gram is a fixed-length string without linguistic meaning. The *n*-grams are extracted by sliding a window of length *n* by one character in the text and recording a sequence of characters in the window at each time. We call it the *1-sliding technique*.

The *n*-gram index has two major advantages—language-neutral and error-tolerant—since terms are extracted by

1-sliding technique [3,18,20]. The first advantage allows us to disregard the characteristics of the language. Thus, the n-gram index is widely used for Asian languages, where complex linguistic knowledge is required for identifying words, or for biological sequences, where a clear concept of the word does not exist. The second advantage allows us to retrieve documents with some errors (e.g., typos) as the query result. Thus, the n-gram index is widely used for approximate string matching.

Nevertheless, the n-gram index has also drawbacks: the size tends to be large, and the performance of queries—especially, long ones—tends to be bad [3,18]. These drawbacks stem from a large volume of position information of n-grams extracted by the 1-sliding technique. Here, the *position information* represents the document identifier and the offsets within the document where an n-gram occurs. There have been a number of efforts to reduce the size of the n-gram index. The compression of the inverted index is widely employed [26,32]. However, this scheme requires additional compression and decompression costs since it compresses posting lists during indexing and decompresses them during query processing. On the other hand, some methods have been proposed to extract n-grams sparsely [18,27]. However, this scheme lowers search accuracy since it reduces the index size by omitting some position information.

In this paper, we propose the *two-level n-gram inverted index* (simply, the *n-gram/2L index*) that significantly reduces the size and improves the query performance by using the relational normalization theory. We first identify that redundancy in the position information exists in the (full-text) n-gram index and that this is caused by a non-trivial multivalued dependency (MVD). Then, in order to eliminate such redundancy, we construct the index in two levels—(1) the front-end index and (2) the back-end index—in the same way as we decompose the relation into two relations for normalization. Here, the back-end index is constructed by using subsequences extracted from documents, and the front-end index constructed by using n-grams extracted from those subsequences. Our method achieves the reduction of the size and the improvement of the performance through structurally reforming the index, i.e., replacing the original n-gram index with two smaller ones. We note that there is no compression of the posting lists or omission of position information. From this point of view, we call our method *structural optimization* of the n-gram index.

The n-gram/2L index has four excellent properties. First, the size of the n-gram/2L index is scalable with the database size. That is, compared with the conventional n-gram index, the reduction of the index size becomes more marked in a larger database. Second, the query performance of the n-gram/2L index is also scalable with the database size. That is, compared with the conventional one, the improvement of the query performance becomes more marked in a larger database. Third, the query processing time increases at a lower rate in the n-gram/2L index than in the n-gram index as the query length gets longer. We investigate the reasons for these desirable properties in Sect. 4.

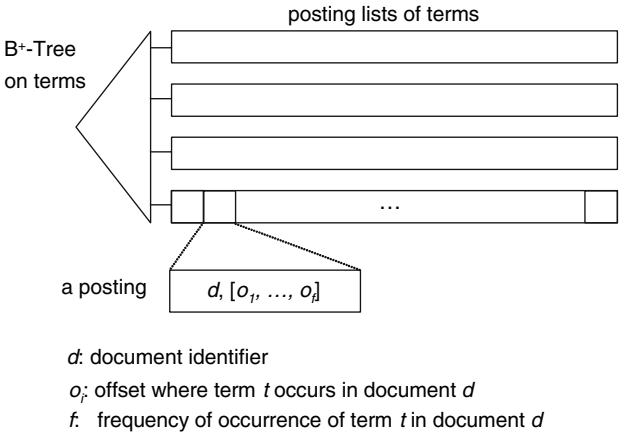


Fig. 1 The structure of the inverted index

The rest of this paper is organized as follows. Section 2 explains the n-gram inverted index. Section 3 proposes the structure and algorithms of the n-gram/2L index. Section 4 presents the formal model of the n-gram/2L index and analyzes the size and query performance. Section 5 discusses implementation of the n-gram/2L index. Section 6 presents the results of performance evaluation. Section 7 describes existing work related to the n-gram index. Section 8 summarizes and concludes the paper.

2 Preliminary

In this section, we explain the inverted index and the n-gram index. The *inverted index* is a term-oriented mechanism for quickly searching documents containing a given term [3]. Here, a *document* is a finite sequence of characters, and a *term* a subsequence of a document.

The inverted index consists of two major components: terms and posting lists [32]. A *posting list*, which is related to a specific term, is a list of postings that contain information about the occurrences of the term. A *posting* consists of the identifier of the document that contains the term and the list of the offsets where the term occurs in the document. For each term t, there is a posting list that contains postings $\langle d, [o_1, \ldots, o_f] \rangle$, where d is a document identifier, $[o_1, \ldots, o_f]$ is a list of offsets o, and f is the frequency of occurrence of the term t in the document d [26]. Postings in a posting list are usually stored in the increasing order of d, and offsets within a posting in the increasing order of o. Besides, an index such as the B+-tree is created on terms in order to quickly locate a posting list. Figure 1 shows the structure of the inverted index.

The inverted index is classified into two types depending on the method of extracting terms: (1) the word-based inverted index using a word as a term and (2) the n-gram index using an n-gram as a term [15,20,32]. We focus on the n-gram index in this paper. Let us consider a document d as a sequence of characters $c_0, c_1, \ldots, c_{w-1}$. An n-gram is a subsequence of length n. Extracting n-grams from a document d can be done by using the 1-sliding technique, that is, sliding a window of length n from c_0 to c_{w-n} and storing the characters located in the window. The ith n-gram extracted from d is the sequence $c_i, c_{i+1}, \ldots, c_{i+n-1}$.

Example 1 Figure 2 shows an example of the n-gram index. Suppose $n = 2$. Figure 2a shows the set of documents. Figure 2b shows the 2-gram index built from these documents. A 2-gram AB occurs in document 0 at the offset 0 and 5. Thus, in the posting list of the term AB, there is a posting $\langle 0, [0, 5]\rangle$ indicating AB occurs in document 0 at the offset 0 and 5.

Query processing is done in two steps: (1) extracting n-grams from a given query string and searching the posting lists of those n-grams; and (2) performing merge join between those posting lists using the document identifier as the join attribute [3]. For exact-match queries, we are able to improve the query performance by splitting a query string into disjoint n-grams.

For example, suppose we execute a query "ABB" by using the n-gram index in Fig. 2. In the first step, the two 2-grams "AB" and "BB" are extracted from the query, and two posting lists of those 2-grams are searched. In the second step, merge join between those two posting lists is performed in order to find the documents where the two 2-grams "AB" and "BB" occur consecutively—constituting "ABB". As the query result, the document identifiers 0 and 2 are returned.

3 n-gram/2L index

3.1 Index structure

Figure 3 shows the structure of the n-gram/2L index, which consists of the back-end index and the front-end index. The *back-end index* stores the offsets of subsequences within documents, and the *front-end index* the offsets of n-grams within subsequences. Here, subsequences have either fixed-length or variable-length.

3.2 Index building algorithm

In this section, we present the algorithms for building the n-gram/2L index. We present the basic index building algorithm using fixed-length subsequences in Sect. 3.2.1 and the enhanced index building algorithm using variable-length subsequences in Sect. 3.2.2. Variable-length subsequences allow us to reduce the size of the index and improve the query performance compared with fixed-length subsequences.

3.2.1 Basic index building algorithm

The n-gram/2L index is built through the following four steps: (1) extracting subsequences, (2) building the back-end index, (3) extracting n-grams, and (4) building the front-end index. When extracting subsequences, the length of subsequences is fixed to be m, and consecutive subsequences overlap with each other by $n - 1$. The purpose for this overlap is to prevent missing or duplicating n-grams, i.e., we extract no more or no less n-grams than is necessary. We formally prove the correctness of this method in Theorem 1. Hereafter, we call the subsequence of length m as the *m-subsequence*. The m-subsequence starting from the character c_i is the sequence $c_i, c_{i+1}, \ldots, c_{i+m-1}$. We note that n denotes the length of the n-gram, and m the length of the m-subsequence. We denote the n-gram/2L index constructed with m-subsequences by the *n-gram/2L-m index*.

Theorem 1 *If m-subsequences are extracted such that consecutive ones overlap with each other by $n - 1$, no n-gram is missed or duplicated.*

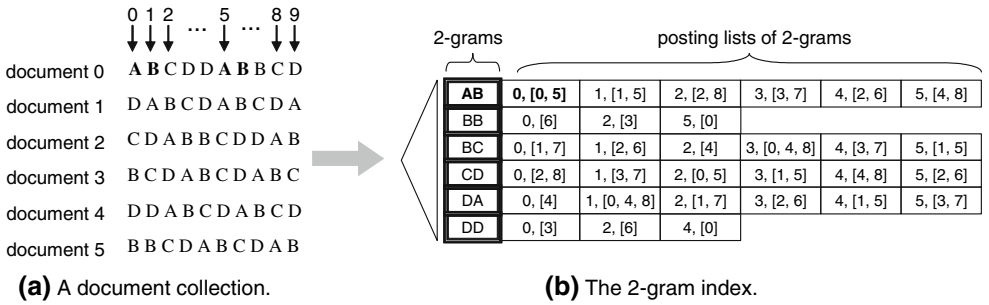
Proof We prove the Theorem by showing that if there is an n-gram duplicated or missed, consecutive m-subsequences do not overlap with each other by $n - 1$.

Case 1: If there is an n-gram duplicated, i.e., an n-gram belongs to two m-subsequences, those m-subsequences should overlap with each other by at least n (Fig. 4b).

Case 2: If there is an n-gram missed, i.e., an n-gram belongs to no m-subsequence, the last m-subsequence ending with the previous n-gram and the first m-subsequence starting with the next n-gram should overlap with each other by less than $n - 1$ (Fig. 4b).

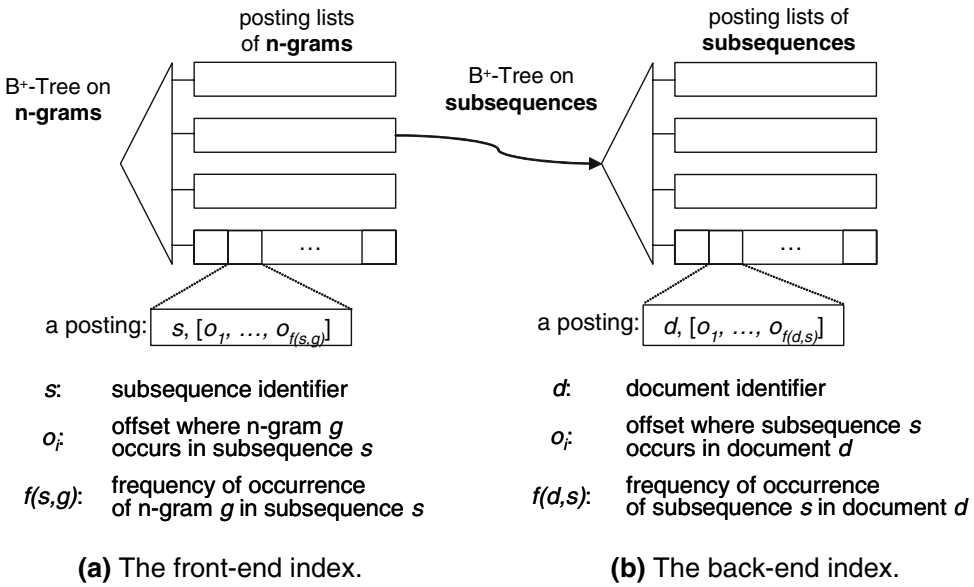
Thus, if m-subsequences are extracted such that they overlap with each other by $n - 1$, no n-gram is missed or duplicated. □

Figure 5 shows the basic algorithm for building the n-gram/2L index. We call this algorithm *Basic n-Gram/2L Index Building*. In Step 1, the algorithm extracts m-subsequences from a set of documents such that they overlap with each other by $n - 1$. Suppose that a document is the sequence of characters $c_0, c_1, \ldots, c_{w-1}$. The algorithm extracts m-subsequences starting from the character $c_{i*(m-n+1)}$ for all i where $0 \leq i < \lfloor \frac{w-n+1}{m-n+1} \rfloor$. If the length of the last m-subsequence is less than m, the algorithm pads blank characters to the m-subsequence to guarantee the length of m. In Step 2, the algorithm builds the back-end index



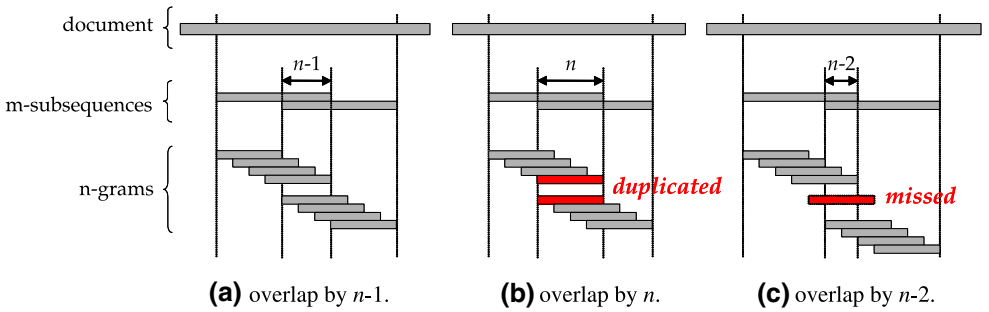
(a) A document collection. **(b)** The 2-gram index.

Fig. 2 An example of the n-gram index



(a) The front-end index. **(b)** The back-end index.

Fig. 3 The structure of the n-gram/2L index



(a) overlap by n-1. **(b)** overlap by n. **(c)** overlap by n-2.

Fig. 4 The cases where m-subsequences overlap with each other

using the m-subsequences obtained in Step 1. For each m-subsequence s occurring f times in a document d at offsets $o_1, \ldots, o_f$, a posting $\langle d, [o_1, \ldots, o_f]\rangle$ is appended to the posting list of s. In Step 3, the algorithm extracts n-grams from the set of m-subsequences obtained in Step 1 by using the 1-sliding technique. In Step 4, the algorithm builds the front-end index using the n-grams obtained in Step 3. For each n-gram g occurring f times in an m-subsequence v at offsets $o_1, \ldots, o_f$, a posting $\langle v, [o_1, \ldots, o_f]\rangle$ is appended to the posting list of g.

Example 2 Figure 6 shows an example of building the n-gram/2L index. Suppose that $n = 2$ and $m = 4$. Figure 6a shows the set of documents, which is the same set of documents as is in Fig. 2a. Figure 6b shows the set of the 4-subsequences extracted from the documents. Since

Algorithm *Basic* ***n-Gram/2L Index Building*****:**

Input: (1) The document collection D, (2) The length m of subsequences , (3) The length n of n-grams

Output: The n-gram/2L index

Algorithm:

Step 1. Extraction of m-subsequences: for each document in D

1.1 Suppose that a document d is a sequence of characters $c_0,c_1,...,c_{w-1}$;
extract m-subsequences $c_{i*(m-n+1)}\, c_{i*(m-n+1)+1} \cdots c_{i*(m-n+1)+m-1}$ $(0 \le i < \lfloor (w-n+1)/(m-n+1) \rfloor)$ and
record the offsets of the m-subsequences within d.

1.2 If the length of the last m-subsequence is less than m,
pad blank characters to the m-subsequence.

Step 2. Construction of the back-end inverted index: for each m-subsequence obtained in Step 1

2.1 Suppose that an m-subsequence s occurs in a document d at offsets $o_0,o_1,...,o_f$;
append a posting $<d, [o_0,o_1,...,o_f]>$ to the posting list of s.

Step 3. Extraction of n-grams: for each m-subsequence obtained in Step 1

3.1 Suppose that an m-subsequence s is a sequence of characters $c_0,c_1,...,c_{h-1}$;
extract n-grams $c_ic_{i+1}...c_{i+n-1}$ $(0 \le i < h-n+1)$ and
record the offsets of the n-grams within s.

Step 4. Construction of the front-end inverted index: for each n-gram obtained in Step 3

4.1 Suppose that an n-gram g occurs in an m-subsequence s at offsets $o_0,o_1,...,o_f$;
append a posting $<s, [o_0,o_1,...,o_f]>$ to the posting list of g.

Fig. 5 The basic algorithm of building the n-gram/2L index

4-subsequences are extracted such that they overlap by 1 (i.e., $n-1$), those extracted from the document 0 are "ABCD", "DDAB", and "BBCD". Figure 6c shows the back-end index built from these 4-subsequences. Since the 4-subsequence "ABCD" occurs at the offsets 0, 3, and 6 in the documents 0, 3, and 4, respectively, the postings $\langle 0, [0]\rangle$, $\langle 3, [3]\rangle$, and $\langle 4, [6]\rangle$ are appended to the posting list of the 4-subsequence "ABCD". Figure 6d shows the set of the 4-subsequences and their identifiers. Figure 6e shows the set of the 2-grams extracted from the 4-subsequences in Fig. 6d. Since 2-grams are extracted by the 1-sliding technique, those extracted from the 4-subsequence 0 are "AB", "BC", and "CD". Figure 6f shows the front-end index built from these 2-grams. Since the 2-gram "AB" occurs at the offsets 0, 2, 1, and 2 in the 4-subsequences 0, 3, 4, and 5, respectively, the postings $\langle 0, [0]\rangle$, $\langle 3, [2]\rangle$, $\langle 4, [1]\rangle$, and $\langle 5, [2]\rangle$ are appended to the posting list of the 2-gram "AB".

3.2.2 Enhanced index building algorithm

Overview

The method of extracting subsequences affects the size and query performance of the n-gram/2L index. It is preferable to extract subsequences that are occurring repeatedly in the document collection. The more frequently the subsequences occur, the smaller the size of the n-gram/2L index becomes because more repetition of the position information can be eliminated. Furthermore, as subsequences occur more frequently, we tend to have a smaller number of unique subsequences because the number of subsequences extracted from a document collection is inherently limited. A smaller number of unique subsequences helps improve the query performance of the n-gram/2L index. We analyze the index size and query performance in more detail in Sects. 4.2 and 4.3.

Our key observation is that separation by words in the natural language is very useful in extracting subsequences. Since a natural language document consists of words, a word is likely to occur repeatedly in a document collection. Accordingly, a subsequence composed of a word or a sequence of words is likely to occur more repeatedly than an m-subsequence is. Since words have variable lengths, we need to support variable-length subsequences rather than fixed-length ones (i.e., m-subsequences). In this section, we present an enhanced index building algorithm using variable-length subsequences for natural language documents.

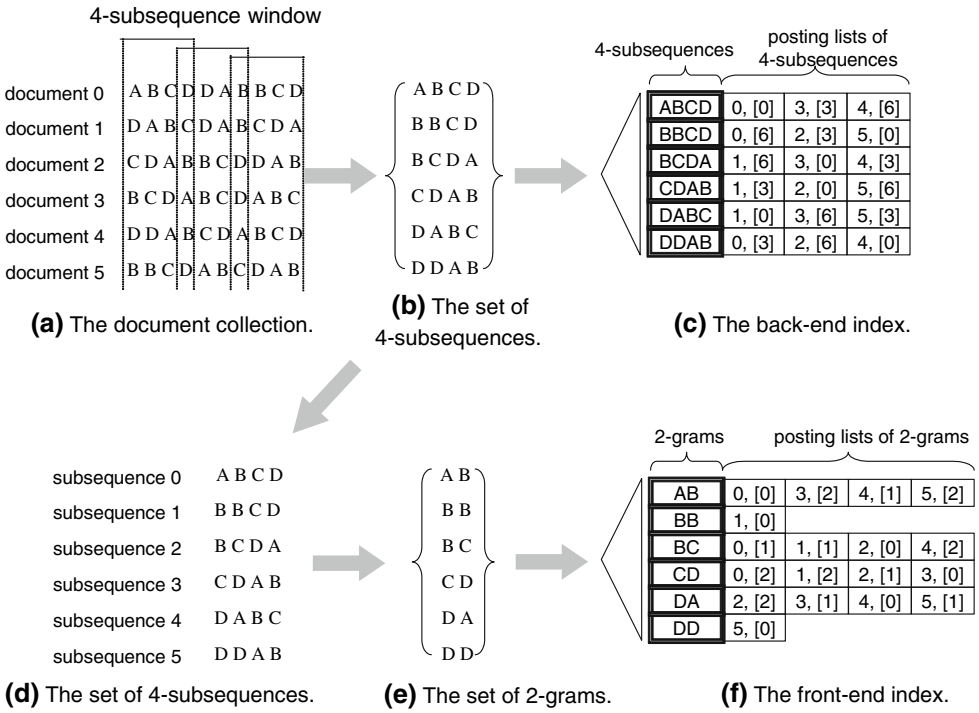


Fig. 6 An example of building the n-gram/2L index

Definition of v-subsequence

The enhanced index building algorithm is identical to the basic index building algorithm except for the method of extracting subsequences. When extracting variable-length subsequences, it is preferable to confine the length of the subsequences within a specified range around the *base length*. To achieve this, we concatenate a short word to consecutive words and split a long word into shorter subsequences. We first denote the base length as v. We present a method of determining the optimal v in Sect. 4.2.2. We then define a *short word* as the one whose length is less than v; a *long word* as the one whose length is greater than or equal to $2v$. A short word is concatenated to consecutive words so as to form a subsequence whose length is greater than or equal to v. A long word is split into subsequences whose length is greater than or equal to v. We call these subsequences as *disjoint v-subsequences*.

In addition to disjoint v-subsequences, we extract a subsequence that overlaps with each of two consecutive disjoint v-subsequences by $n-1$. We call these subsequences as *joining v-subsequences*. Since it overlaps with two disjoint v-subsequences by $n-1$, its length is $2(n-1)$. The purpose for extracting joining v-subsequences is to prevent the n-grams from being missed or duplicated as in Theorem 1. Hereafter, we call both disjoint v-subsequences and joining v-subsequences as *v-subsequences*. We denote the n-gram/2L index constructed with v-subsequences by the *n-gram/2L-v index*.

Extracting v-subsequences

Figure 7 shows the algorithm for extracting v-subsequences. We call this algorithm *v-subsequence Extraction*. In Step 1, we split each long word into multiple subsequences such that the length of each subsequence is greater than or equal to v. In Step 2, we extract v-subsequences. If the length of a word is greater than or equal to v, we extract this word as a v-subsequence. However, if the length of a word is less than v, we concatenate this word with the following words or subsequences to make a v-subsequence. We extract also a joining v-subsequence whenever a distinct v-subsequence is extracted. If the v-subsequence is the last one, and its length is less than v, we concatenate it to the immediately preceding v-subsequence.

Example 3 Figure 8 shows an example of extracting v-subsequences from a document. Suppose that $n = 3$ and $v = 4$. The string "sequence" is a long word because its length (i.e., 8) is equal to $2v$, and thus, is split into "sequ" and "ence." The first occurrence of "A" is a short word because its length (i.e., 1) is less than v, and thus, is concatenated with "text" so as to form "A text". Likewise, we obtain "has many," "A word," "is a sequ," "ence," and "of letters" by concatenating short words. The joining v-subsequences "xtha," "nywo," "dsaw," "rdis," "quen," and "ceof" are also extracted.

Algorithm *v-subsequence Extraction*:

Input: (1) The document collection D

(2) The base length v of v-subsequences

(3) The length n of n-grams

Output: a set of v-subsequences

Algorithm: Execute Steps 1 and 2 for each document in D

Step 1. Splitting long words:

Denote a word w as a sequence of characters $c_0,c_1,...,c_{Len(w)-1}$.

Perform the following step for each word in a document d

1.1 If $Len(w) \geq 2v$, split w into subsequences starting from the character c_i $(0 \leq i \leq \lfloor Len(w) / v \rfloor - 1)$. Here, the length of subsequences is v if $0 \leq i < \lfloor Len(w) / v \rfloor - 1$ or $(Len(w) - v*i)$ otherwise.

Step 2. Extracting v-subsequences:

Denote a document d as a sequence of words or (words split in Step 1.1) $w_0,w_1,...,w_{N-1}$.

Perform the following steps while scanning a document d

2.1 If $Len(w_i) \geq v$, extract w_i as a disjoint v-subsequence and record the offset of w_i within d.

2.2 If $Len(w_i) < v$,

2.2.1 Let $s = w_i$ and perform Step 2.2.1.1 while $Len(s) < v$.

2.2.1.1 Concatenate w_{i+1} to s and increase i by 1.

2.2.2 Extract s as a disjoint v-subsequence and record the offset of s within d.

2.3 Extract a joining v-subsequence s between two consecutive disjoint v-subsequences and record the offset of s within d.

2.4 If the v-subsequence is the last one, and its length is less than v, concatenate it to the immediately preceding v-subsequence.

Fig. 7 The algorithm for extracting v-subsequences

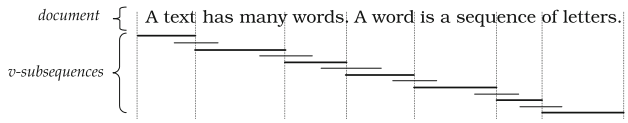


Fig. 8 An example of extracting v-subsequences from a document

3.3 Query processing algorithm

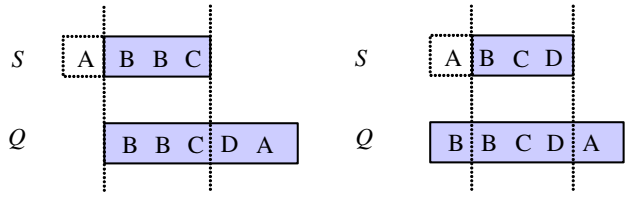
In this section, we present the algorithm for processing queries using the n-gram/2L index. Since the query processing algorithm is identical for both m-subsequences and v-subsequences, we present the algorithm only for m-subsequences.

The query processing algorithm consists of the following two steps: (1) searching the front-end index in order to retrieve candidate results, (2) searching the back-end index in order to refine candidate results. In the first step, we select the m-subsequences that *cover* a query string by searching the front-end index with the n-grams extracted from the query string. The m-subsequences that do not cover the query string are filtered out in this step. In the second step, we select the documents that have a set of m-subsequences $\{S_i\}$ *containing* the query string by searching the back-end index with the m-subsequences retrieved in the first step. The documents including one or more m-subsequences retrieved in the first step represent a set of candidate results satisfying the necessary condition of (i.e., covering) the query. The final results can be obtained in the second step by doing refinement that removes the false positives.

Now, we formally define *cover* in Definition 1 and *contain* in Definition 3.

Definition 1 *S covers Q* if an m-subsequence S and a query string Q satisfy one of the following four conditions: (1) a suffix of S matches a prefix of Q; (2) the whole string of S matches a substring of Q; (3) a prefix of S matches a suffix of Q; (4) a substring of S matches the whole string of Q.

Definition 2 The *expand* function expands a sequence of overlapping character sequences into one character sequence. (1) For a sequence consisting of two character sequences:



(a) S covers Q. **(b)** S does not cover Q.

Fig. 9 Examples of an m-subsequence S covering the query Q

Let $S_i = c_i \cdots c_j$ and $S_p = c_p \cdots c_q$. Suppose that a suffix of S_i and a prefix of S_p overlap by k (i.e., $c_{j-k+1} \cdots c_j = c_p \cdots c_{p+k-1}$, where $k \geq 0$.) Then, $expand(S_i S_p) = c_i \cdots c_j c_{p+k} \cdots c_q$. (2) For a sequence consisting of more than two character sequences: $expand(S_l S_{l+1} \cdots S_m) = expand(expand(S_l S_{l+1}), S_{l+2} \cdots S_m)$.

Definition 3 $\{S_i\}$ *contains* Q if a set of m-subsequences $\{S_i\}$ and a query string Q satisfy the following condition: Let $S_l S_{l+1} \cdots S_m$ be a sequence of m-subsequences overlapping with each other in $\{S_i\}$. A substring of $expand(S_l S_{l+1} \cdots S_m)$ matches the whole string of Q.

Example 4 Figure 9 shows examples of covering. Here, Q is the query and S is an m-subsequence. In Fig. 9a, S covers Q since a suffix of S matches a prefix of Q. In Fig. 9b, S does not cover Q since "BCD" does not satisfy any of the four conditions in Definition 1.

Lemma 1 *A document that has a set of m-subsequences $\{S_i\}$ containing the query string Q includes at least one m-subsequence covering Q.*

Proof We first show the cases that a set of m-subsequences $\{S_i\}$ contains Q in Fig. 10. Let $Len(Q)$ be the length of Q. We classify the cases depending on whether $Len(Q) \geq m$ (Fig. 10a) or $Len(Q) < m$ (Fig. 10b and c). In Fig. 10a, the set $\{S_i, \ldots, S_j\}$ contains Q. In Fig. 10b, the set $\{S_k\}$ contains Q. In Fig. 10c, the set $\{S_p, S_q\}$ contains Q. From Fig. 10 we see that, if the set $\{S_i\}$ contains Q, at least one m-subsequence in $\{S_i\}$ satisfies a condition in Definition 1, covering Q. □

Figure 11 shows the algorithm of processing queries using the n-gram/2L index. We call this algorithm *n-Gram/2L Index Searching*. In Step 1, the algorithm splits the query string Q into multiple n-grams and searches the posting lists of those n-grams in the front-end index. Then, performing merge outer join among those posting lists using the m-subsequence identifier as the join attribute, the algorithm adds the m-subsequences that cover Q by Definition 1 (i.e., m-subsequences satisfying a necessary condition in Lemma 1) into the set S_{cover}. Since an m-subsequence covering Q typically does not have all the n-grams extracted from Q, the algorithm performs merge outer join in Step 1.2. Here, the algorithm uses the offset information in the postings to be merge outer joined in order to check whether the m-subsequence covers Q. In Step 2, the algorithm performs merge outer join among the posting lists of the m-subsequences in S_{cover} using the document identifier as the join attribute. It identifies the set $\{S_i\}$ of the m-subsequences having the same document identifier d_i and performs refinement by checking whether $\{S_i\}$ indeed contains Q according to Definition 3. Since the set $\{S_i\}$ may be a subset of S_{cover}, the algorithm performs merge outer join in Step 2.1. Here, the algorithm uses the offset information in the postings to be merge outer joined in order to check whether $\{S_i\}$ contains Q. If $\{S_i\}$ contains Q, d_i is returned as a query result.

4 Formal analysis of the *n*-gram/2*L* index

In this section, we present a formal analysis of the n-gram/2L index. In Sect. 4.1, we formally prove that the n-gram/2L index is derived by eliminating the redundancy in the position information that exists in the n-gram index. In Sect. 4.2, we present the space complexities of these indexes. In Sect. 4.3, we present their time complexities of searching.

4.1 Formalization

In this section, we observe that the redundancy of the position information existing in the n-gram index is caused by a non-trivial multivalued dependency (MVD) [6] and show that the n-gram/2L index can be derived by eliminating that redundancy through relational decomposition to the fourth normal form (4NF). Since this observation is identical for both m-subsequences and v-subsequences, we present the formalization only for m-subsequences.

For the sake of theoretical development, we first consider the relation that is converted from the n-gram index so as to obey the first normal form (1NF). We call this relation the *NDO relation*. This relation has three attributes N, D, and O. Here, N indicates n-grams, D document identifiers, and O offsets. Further, we consider the relation obtained by adding the attribute S and by splitting the attribute O into two attributes O_1 and O_2. We call this relation the *$SNDO_1O_2$ relation*. This relation has five attributes S, N, D, O_1, and O_2. Here, S indicates m-subsequences, O_1 the offsets of n-grams within m-subsequences, and O_2 the offsets of m-subsequences within documents.

The values of the attributes S, O_1, and O_2 appended to the relation $SNDO_1O_2$ are automatically determined by those of the attributes N, D, and O in the relation NDO. The reason is that an n-gram appearing at a specific offset o in

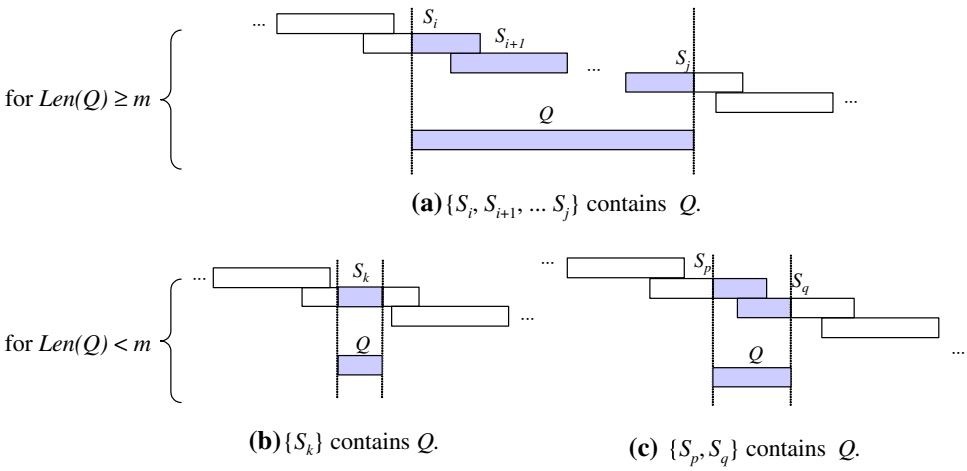


Fig. 10 The cases that a set of m-subsequences contains Q

Algorithm *n-Gram/2L Index Searching*:

Input: (1) The two-level n-gram inverted index

(2) A query string Q

Output: Identifiers of the documents containing Q

Algorithm:

Step 1. Searching the front-end inverted index:

1.1 Split Q into multiple n-grams and search the posting lists of those n-grams.

1.2 Perform merge outer join among those posting lists using the m-subsequence identifier as the join attribute; add the m-subsequences that cover Q by Definition 1 into the set S_{cover}.

Step 2. Searching the back-end inverted index:

2.1 Perform merge outer join among the posting lists of m-subsequences in S_{cover} using the document identifier as the join attribute.

2.1.1 Identify the set $\{S_i\}$ of m-subsequences having the same document identifier d_i and perform refinement by checking whether $\{S_i\}$ contains Q or not according to Definition 3.

2.1.2 If $\{S_i\}$ contains Q, d_i is returned as the query result.

Fig. 11 The algorithm of processing queries using the n-gram/2L index

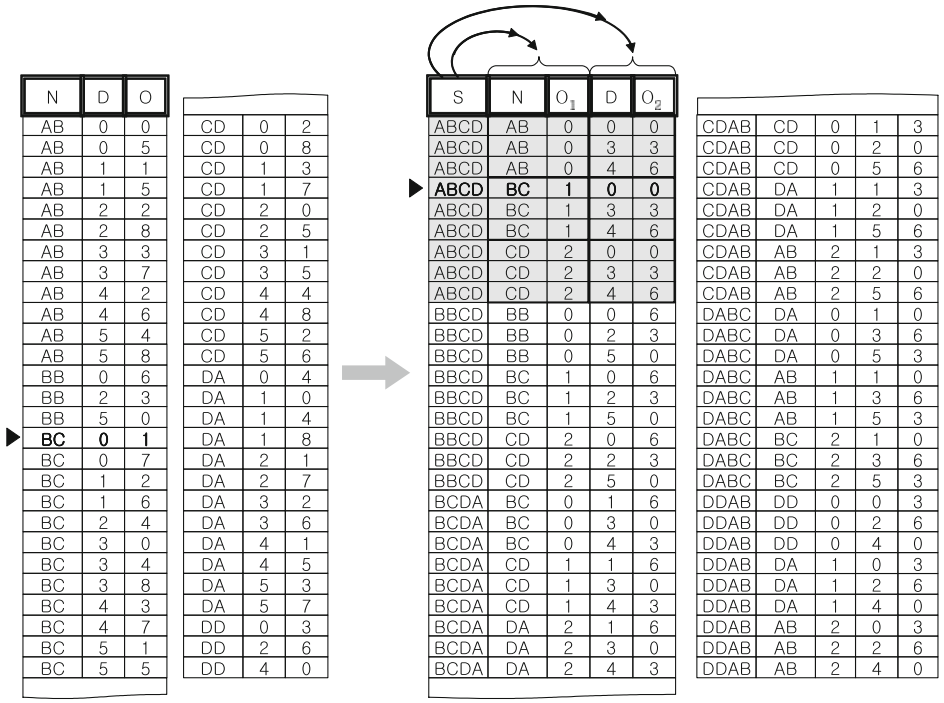
the document belongs to only one m-subsequence as shown in Theorem 1. In the tuple (s, n, d, o_1, o_2) determined by a tuple (n, d, o) of the relation NDO, s represents the m-subsequence that the n-gram n occurring at the offset o in the document d belongs to. o_1 is the offset where the n-gram n occurs in the m-subsequence s, and o_2 the offset where the m-subsequence s occurs in the document d.

Example 5 Figure 12a shows the relation NDO converted from the n-gram index in Fig. 2b. Figure 12b shows the relation $SNDO_1O_2(m = 4)$ derived from the relation NDO in Fig. 12a. Here, the tuples of the relation $SNDO_1O_2$ are sorted by the values of attribute S. In Fig. 12, the marked tuple of the relation $SNDO_1O_2$ is determined by the marked tuple of the relation NDO. Since the 2-gram "BC" at the offset 1 in the document 0 belongs to the 4-subsequence "ABCD" in Fig. 6a, the value of the attribute S of the marked tuple becomes "ABCD". The value of the attribute O_1 becomes 1 because the 2-gram "BC" occurs in the 4-subsequence "ABCD" at the offset 1, and that of the attribute O_2 becomes 0 because the 4-subsequence "ABCD" occurs in the document 0 at the offset 0.

We now prove that non-trivial MVDs hold in the relation $SNDO_1O_2$ (i.e., the n-gram index) in Lemma 2.

Lemma 2 *The non-trivial MVDs* $S \rightarrow\rightarrow NO_1$ *and* $S \rightarrow\rightarrow DO_2$ *hold in the relation* $SNDO_1O_2$. *Here,* S *is not a superkey.*

Proof By the definition of the MVD [6,28], $X \rightarrow\rightarrow Y$ holds in R, where X and Y are subsets of R, if whenever r is a relation for R and μ and ν are two tuples in r, with $\mu, \nu \in$



N	D	O
AB	0	0
AB	0	5
AB	1	1
AB	1	5
AB	2	2
AB	2	8
AB	3	3
AB	3	7
AB	4	2
AB	4	6
AB	5	4
AB	5	8
BB	0	6
BB	2	3
BB	5	0
BC	0	1
BC	0	7
BC	1	2
BC	1	6
BC	2	4
BC	3	0
BC	3	4
BC	3	8
BC	4	3
BC	4	7
BC	5	1
BC	5	5

CD	0	2
CD	0	8
CD	1	3
CD	1	7
CD	2	0
CD	2	5
CD	3	1
CD	3	5
CD	4	4
CD	4	8
CD	5	2
CD	5	6
DA	0	4
DA	1	0
DA	1	4
DA	1	8
DA	2	1
DA	2	7
DA	3	2
DA	3	6
DA	4	1
DA	4	5
DA	5	3
DA	5	7
DD	0	3
DD	2	6
DD	4	0

(a) An example of the NDO relation.

S	N	O_1	D	O_2
ABCD	AB	0	0	0
ABCD	AB	0	3	3
ABCD	AB	0	4	6
ABCD	BC	1	0	0
ABCD	BC	1	3	3
ABCD	BC	1	4	6
ABCD	CD	2	0	0
ABCD	CD	2	3	3
ABCD	CD	2	4	6
BBCD	BB	0	0	6
BBCD	BB	0	2	3
BBCD	BB	0	5	0
BBCD	BC	1	0	6
BBCD	BC	1	2	3
BBCD	BC	1	5	0
BBCD	CD	2	0	6
BBCD	CD	2	2	3
BBCD	CD	2	5	0
BCDA	BC	0	1	6
BCDA	BC	0	3	0
BCDA	BC	0	4	3
BCDA	CD	1	1	6
BCDA	CD	1	3	0
BCDA	CD	1	4	3
BCDA	DA	2	1	6
BCDA	DA	2	3	0
BCDA	DA	2	4	3

CDAB	CD	0	1	3
CDAB	CD	0	2	0
CDAB	CD	0	5	6
CDAB	DA	1	1	3
CDAB	DA	1	2	0
CDAB	DA	1	5	6
CDAB	AB	2	1	3
CDAB	AB	2	2	0
CDAB	AB	2	5	6
DABC	DA	0	1	0
DABC	DA	0	3	6
DABC	DA	0	5	3
DABC	AB	1	1	0
DABC	AB	1	3	6
DABC	AB	1	5	3
DABC	BC	2	1	0
DABC	BC	2	3	6
DABC	BC	2	5	3
DDAB	DD	0	0	3
DDAB	DD	0	2	6
DDAB	DD	0	4	0
DDAB	DA	1	0	3
DDAB	DA	1	2	6
DDAB	DA	1	4	0
DDAB	AB	2	0	3
DDAB	AB	2	2	6
DDAB	AB	2	4	0

(b) An example of the $SNDO_1O_2$ relation. (sorted by attribute S)

Fig. 12 An example showing the existence of non-trivial MVDs in the relation $SNDO_1O_2$

$r, \mu[X] = \nu[X]$ (that is, μ and ν agree on the attributes of X), then r also contains tuples ϕ and ψ, that satisfy three conditions below.

1. $\phi[X] = \psi[X] = \mu[X] = \nu[X]$
2. $\phi[Y] = \mu[Y]$ and $\phi[R - X - Y] = \nu[R - X - Y]$
3. $\psi[Y] = \nu[Y]$ and $\psi[R - X - Y] = \mu[R - X - Y]$.

$X \rightarrow\rightarrow Y$ is non-trivial if $Y \not\subseteq X$ and $X \cup Y \neq R$ (i.e., $R - X - Y \neq \emptyset$) (meaning Y and $R - X - Y$ are non-empty sets of attributes) [6]. That is, a non-trivial MVD holds if, for any value of the attribute X, the Y-values and the $(R - X - Y)$-values form a Cartesian product [25].

Let $\{S_1, \ldots, S_r\}$ be a set of m-subsequences extracted from the document collection, $\{N_{i1}, \ldots, N_{iq}\}$ a set of n-grams extracted from an m-subsequence S_i, and $\{D_{i1}, \ldots, D_{ip}\}$ a set of documents where S_i occurs $(1 \leq i \leq r)$. Then, the set of n-grams $\{N_{i1}, \ldots, N_{iq}\}$ is extracted from every document in the set of documents $\{D_{i1}, \ldots, D_{ip}\}$ since S_i is in these documents. Hence, in the set of tuples whose S-value is S_i, the NO_1-values representing the n-grams extracted from S_i and the DO_2-values representing the documents containing S_i always form a Cartesian product. Suppose that $R = SNDO_1O_2, X = S, Y = NO_1$, and $R-X-Y = DO_2$. Then, the three conditions above are satisfied because the Y-values and the $(R - X - Y)$-values form a Cartesian product in the set of tuples having the same X-values. These conditions above are satisfied also when $Y = DO_2$. We also note that $NO_1 \not\subseteq S, DO_2 \not\subseteq S, NO_1 \cup S \neq SNDO_1O_2$, and $DO_2 \cup S \neq SNDO_1O_2$. Thus, the non-trivial MVDs $S \rightarrow\rightarrow NO_1$ and $S \rightarrow\rightarrow DO_2$ hold in the relation $SNDO_1O_2$. S is obviously not a superkey as shown in the counter example of Fig 12b. □

Intuitively, non-trivial MVDs hold in the relation $SNDO_1O_2$ because the set of documents, where an m-subsequence occurs, and the set of n-grams, which are extracted from that m-subsequence, are independent of each other. If attributes in a relation are independent of each other, non-trivial MVDs hold in that relation [6,25,28]. In the relation $SNDO_1O_2$, due to the independence between the set of documents and the set of n-grams, there exist the tuples corresponding to all possible combinations of documents and n-grams for a given m-subsequence.

Example 6 Figure 12b shows an example showing the existence of the non-trivial MVDs $S \rightarrow\rightarrow NO_1$ and $S \rightarrow\rightarrow DO_2$ in the relation $SNDO_1O_2$. In the shaded tuples of the relation $SNDO_1O_2$ shown in Fig. 12b, there exists the redundancy that the DO_2-values (0, 0), (3, 3), and (4, 6)

repeatedly appear for the NO_1-values ("AB", 0), ("BC", 1), and ("CD", 2). That is, the NO_1-values and the DO_2-values form a Cartesian product in the tuples whose S-value is "ABCD". We note that there such repetitions also occur in the other S-values.

Corollary 1 *The relation $SNDO_1O_2$ is not in the 4NF.*

Proof A non-trivial MVD $S \rightarrow\rightarrow NO_1$ exists, where S is not a superkey. □

The front-end and back-end indexes of the n-gram/2L index are identical to two relations obtained by decomposing the relation $SNDO_1O_2$ so as to obey 4NF. We prove this proposition in Theorem 2. Thus, it can be proved that the redundancy caused by a non-trivial MVD does not exist in the n-gram/2L index [28].

Lemma 3 *The decomposition (SNO_1, SDO_2) is in 4NF.*

Proof MVD's in SNO_1 are $SO_1 \rightarrow\rightarrow N$, $SNO_1 \rightarrow\rightarrow S|N|O_1$. Those in SDO_2 are $DO_2 \rightarrow\rightarrow S$, $SDO_2 \rightarrow\rightarrow S|D|O_2$. All of these MVD's are trivial ones and do not violate 4NF. □

Theorem 2 *The 4NF decomposition (SNO_1, SDO_2) of the relation $SNDO_1O_2$ is identical to the front-end and back-end indexes of the n-gram / 2L index.*

Proof The relation SNO_1 is represented as the front-end index by regarding N, S, and O_1 as a term, an m-subsequence identifier, and an offset, respectively. Similarly, the relation SDO_2 is represented as the back-end index by regarding S, D, and O_2 as a term, a document identifier, and an offset, respectively. Therefore, the 4NF decomposition (SNO_1, SDO_2) of the relation $SNDO_1O_2$ is identical to the front-end and back-end indexes of the n-gram/2L index. □

Example 7 Figure 13 shows that the relation $SNDO_1O_2$ in Fig. 12 is decomposed into the two relations SNO_1 and SDO_2. In the attribute S of the relation SDO_2, the values in parentheses indicate m-subsequence identifiers. The tuples of the relation SDO_2 are sorted by the m-subsequence identifier. The shaded tuples of the relation $SNDO_1O_2$ in Fig. 12 are decomposed into the shaded ones of the relations SNO_1 and SDO_2 in Fig. 13. We note that the redundancy, i.e., the Cartesian product between NO_1 and DO_2 in Fig. 12 has been eliminated in Fig. 13. We also note that the relations SNO_1 and SDO_2, when represented in the form of the inverted index, become identical to the front-end and back-end indexes in Fig. 6, respectively.

4.2 Analysis of the index size

The parameters affecting the size of the n-gram/2L index are the length n of n-grams and length m of m-subsequences (or the base length v of v-subsequences). In general, n is the value determined by applications. On the other hand, m (or v) is the value that can be freely tuned when creating the n-gram/2L index. In this section, we analyze the size of the n-gram/2L index and present the model for determining the optimal length of m (or v) that minimizes the index size. We denote the optimal length of m by m_o and the optimal base length of v by v_o. We present the analyses for the n-gram/2L-m index in Sect. 4.2.1 and for the n-gram/2L-v index in Sect. 4.2.2.

4.2.1 *n-gram/2L-m index*

In Table 1, we summarize some basic notations to be used for analyzing the size of the n-gram/2L index. Here, we simply regard the index size as the number of the offsets stored because the former is approximately proportional to the latter, the latter representing the occurrences of terms in documents [32].

Now, in order to determine the value of m_o, we define the *decomposition efficiency* in Definition 4.

Definition 4 The *decomposition efficiency* is the ratio of the size of the n-gram index to that of the n-gram/2L index. Thus,

$$\text{decomposition efficiency} = \frac{\text{size}_{n\text{gram}}}{\text{size}_{\text{front}} + \text{size}_{\text{back}}}. \quad (1)$$

The decomposition efficiency in Definition 4 is computed through Eqs. (1)–(5). We count the number of offsets in the index using the number of tuples in the relation $SNDO_1O_2$. The number of tuples in the relation $SNDO_1O_2$ is equal to that of offsets of the n-gram index since the relation $SNDO_1O_2$ is created by normalizing the n-gram index into 1NF. As mentioned in Lemma 2, for any value of the attribute S in the relation $SNDO_1O_2$, the values of attributes NO_1 and those of attributes DO_2 form a Cartesian product. Thus, in the relation $SNDO_1O_2$, the number of tuples having s as the value of S becomes $k_{n\text{gram}}(s) \times k_{\text{doc}}(s)$. Accordingly, the size of the n-gram index can be calculated as in Eq. (2), i.e., the summation of $k_{n\text{gram}}(s) \times k_{\text{doc}}(s)$ for all unique m-subsequences. We obtain the sizes of the front-end index and the back-end index similary. In the relation SNO_1 corresponding to the front-end index, the number of tuples having s as the value of S becomes $k_{n\text{gram}}(s)$. Hence, the size of the front-end index is as in Eq. (3), i.e., the summation of $k_{n\text{gram}}(s)$ for all unique m-subsequences. In the relation SDO_2 corresponding to the back-end index, the number of tuples having s as the value of S becomes $k_{\text{doc}}(s)$. Hence, the size of the back-end index is as in Eq. (4), i.e., the summation of $k_{\text{doc}}(s)$ for all unique m-subsequences. Consequently,

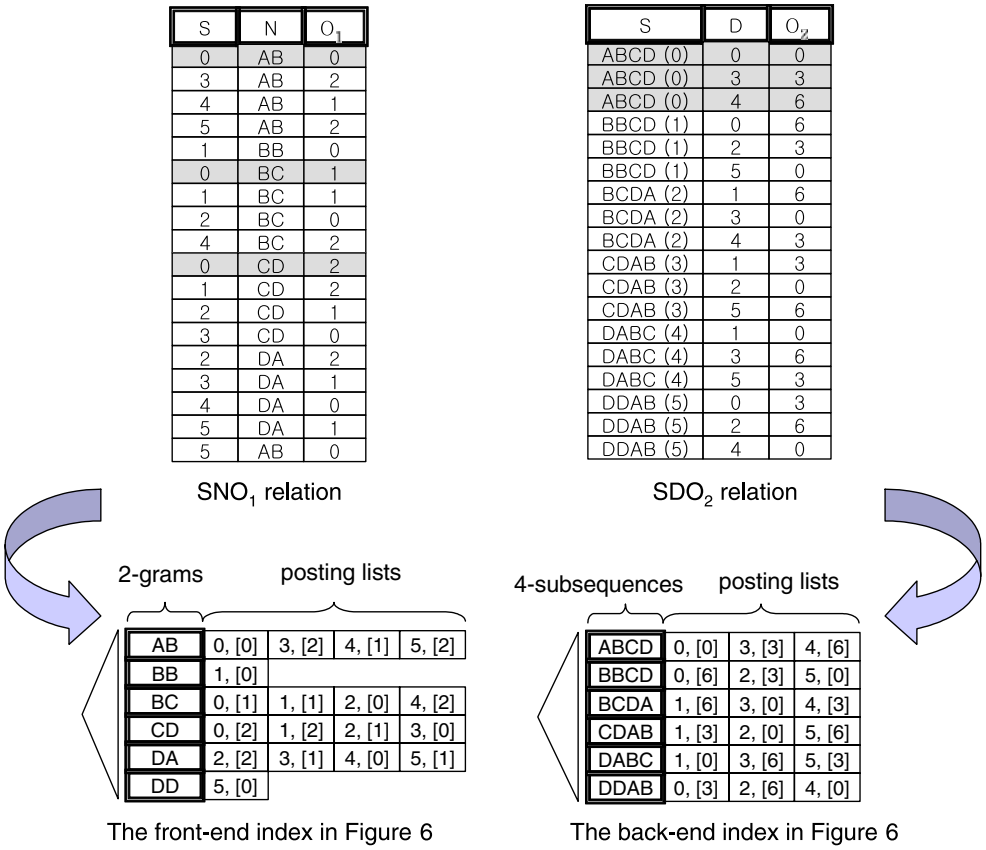


Fig. 13 The result of decomposing the relation $SNDO_1O_2$ in Fig. 12 into two relations

Table 1 The notations to be used for analyzing the size of the n-gram/2L index

Symbols	Definitions
$\text{size}_{n\text{gram}}$	The size of the n-gram index
$\text{size}_{\text{front}}$	The size of the front-end index
$\text{size}_{\text{back}}$	The size of the back-end index
$\mathbb{S}$	The set of unique m-subsequences extracted from the document collection
$k_{n\text{gram}}(s)$	The number of the n-grams extracted from an m-subsequence s
$k_{\text{doc}}(s)$	The frequency of an m-subsequence s appearing in the document collection
$\text{avg}_{n\text{gram}}(\mathbb{S})$	The average value of $k_{n\text{gram}}(s)$ where $s \in \mathbb{S}$ $(= (\sum_{s \in \mathbb{S}} k_{n\text{gram}}(s)) / \lvert\mathbb{S}\rvert)$
$\text{avg}_{\text{doc}}(\mathbb{S})$	The average value of $k_{\text{doc}}(s)$ where $s \in \mathbb{S}$ $(= (\sum_{s \in \mathbb{S}} k_{\text{doc}}(s)) / \lvert\mathbb{S}\rvert)$

we obtain Eq. (5) for the decomposition efficiency from Eqs. (1)–(4).

$$\begin{aligned}\text{size}_{n\text{gram}} &= \sum_{s \in \mathbb{S}} \left(k_{n\text{gram}}(s) \times k_{\text{doc}}(s) \right) \\ &\approx |\mathbb{S}| \left(\text{avg}_{n\text{gram}}(\mathbb{S}) \times \text{avg}_{\text{doc}}(\mathbb{S}) \right) \end{aligned} \tag{2}$$

$$\text{size}_{\text{front}} = \sum_{s \in \mathbb{S}} k_{n\text{gram}}(s) \approx |\mathbb{S}| \text{avg}_{n\text{gram}}(\mathbb{S}) \tag{3}$$

$$\text{size}_{\text{back}} = \sum_{s \in \mathbb{S}} k_{\text{doc}}(s) \approx |\mathbb{S}| \text{avg}_{\text{doc}}(\mathbb{S}) \tag{4}$$

$$\text{decomposition efficiency} \approx \frac{\text{avg}_{n\text{gram}}(\mathbb{S}) \times \text{avg}_{\text{doc}}(\mathbb{S})}{\text{avg}_{n\text{gram}}(\mathbb{S}) + \text{avg}_{\text{doc}}(\mathbb{S})}. \tag{5}$$

The decomposition efficiency is computed by using $\mathbb{S}$, $k_{n\text{gram}}(s)$, and $k_{\text{doc}}(s)$, which can be obtained by preprocessing the document collection. This can be done by sequentially scanning the document collection only once (i.e., O(data

size)). To determine m_o, we first compute decomposition efficiencies for several candidate values of m, and then, select the one that provides the maximum decomposition efficiency. Experimental results show that m_o is determined approximately in the range $(n+1) \sim (n+3)$, i.e., longer than n by 1–3, in the document collection of 10 MB–1 GB.

Equations (2)–(4) shows that the space complexity of the n-gram index is $O(|\mathbb{S}|(\text{avg}_{n\text{gram}} \times \text{avg}_{\text{doc}}))$, while that of the n-gram/2L index is $O(|\mathbb{S}|(\text{avg}_{n\text{gram}} + \text{avg}_{\text{doc}}))$. As indicated by Eq. (5), the decomposition efficiency is maximized when $\text{avg}_{n\text{gram}}$ is equal to avg_{doc}. Here, avg_{doc} increases as the database size gets larger, and $\text{avg}_{n\text{gram}}$ does as m gets longer. $\text{avg}_{n\text{gram}}$ also becomes larger in a larger database since we choose a longer m_o to obtain the maximum decomposition efficiency. That is, both $\text{avg}_{n\text{gram}}$ and avg_{doc} become larger as the database size increases. Since $(\text{avg}_{n\text{gram}} \times \text{avg}_{\text{doc}})$ increases more rapidly than $(\text{avg}_{n\text{gram}} + \text{avg}_{\text{doc}})$ does, the decomposition efficiency increases as the database size does. Therefore, the n-gram/2L index has the characteristic of reducing the index size more for a larger database.

4.2.2 n-gram / 2L-v index

The space complexity of the n-gram/2L-v index is identical to that of n-gram/2L-m index since the n-gram/2L-v index is obtained just in the same way as the n-gram/2L-m index is—by decomposing the relation $SNDO_1O_2$ into the relation SNO_1 and SDO_2.

Nevertheless, the size of the n-gram/2L-v index is smaller than that of the n-gram/2L-m index. We explain this advantage using Eqs. (2)–(5). $|\mathbb{S}|$ is decreased because v-subsequences are extracted based on words, which occur more repeatedly in the document collection than m-subsequences do (for example, when the size of the document collection is 1 GB, $|\mathbb{S}| = 1,662,900$ for m-subsequences ($m = m_o = 5$) and $|\mathbb{S}| = 1,015,699$ for v-subsequences ($v = v_o = 4$).) On the other hand, $|\mathbb{S}|(\text{avg}_{n\text{gram}} \times \text{avg}_{\text{doc}})$ in Eq. (2) remains to be the same regardless of the kinds of subsequences because it means the total number of n-grams extracted from the document collection. Thus, a decrease of $|\mathbb{S}|$ means that $\text{avg}_{n\text{gram}}$ and avg_{doc} increase. That is, $\text{avg}_{n\text{gram}}$ and avg_{doc} in the n-gram/2L-v index get larger than those in the n-gram/2L-m index. We note that Eq. (5) increases as both $\text{avg}_{n\text{gram}}$ and avg_{doc} increase since $(\text{avg}_{n\text{gram}} \times \text{avg}_{\text{doc}})$ increases more rapidly than $(\text{avg}_{n\text{gram}} + \text{avg}_{\text{doc}})$ does. Thus, the decomposition efficiency is enhanced if we use v-subsequences.

v_o is determined in the same way as m_o is. Experimental results show that v_o typically is within the range $n-(n+1)$, i.e., longer than n by 0−1, for document collections of 10 MB–1 GB.

4.3 Analysis of the query performance

The parameters affecting the query performance of the n-gram/2L index are m (or v), n, and the length $Len(Q)$ of the query string Q. In this section, we conduct a ballpark analysis of the query performance to investigate the trend depending on these parameters. We present the analyses for the n-gram/2L-m index in Sect. 4.3.1 and for the n-gram/2L-v index in Sect. 4.3.2.

4.3.1 n-gram/2L-m index

For simplicity of our analysis, we first make the following two assumptions: (1) the query processing time is proportional to the number of offsets accessed and the number of posting lists accessed. The latter has a nontrivial effect on performance since accessing a posting list incurs seek time for moving the disk head to locate the posting list; (2) the size of the document collection is so large that all possible combinations of n-grams(=$|\Sigma|^n$) or m-subsequences(=$|\Sigma|^m$), where Σ denotes the alphabet, are indexed in the inverted index (for example, when $|\Sigma| = 26$ and $m = 5$, $|\Sigma|^m = 11,881,376$). Since the performance of query processing is important especially in a large database, the second assumption is indeed reasonable.

The ratio of the query performance of the n-gram index to that of the n-gram/2L index is computed by using Eqs. (6)–(9). Let k_{offset} be the average number of offsets in a posting list, and k_{plist} be the number of posting lists accessed during query processing. Then, the number of offsets accessed during query processing is $k_{\text{offset}} \times k_{\text{plist}}$. In the n-gram index, since k_{offset} is $\frac{\text{size}_{n\text{gram}}}{|\Sigma|^n}$ and k_{plist} is $\frac{Len(Q)}{n}$, the query processing time is as in Eq. (6). In the front-end index of the n-gram/2L index, since k_{offset} is $\frac{\text{size}_{\text{front}}}{|\Sigma|^n}$ and k_{plist} is $(Len(Q)-n+1)$, the query processing time is as in Eq. (7). In the back-end index of the n-gram/2L index, k_{offset} is $\frac{\text{size}_{\text{back}}}{|\Sigma|^m}$. Here, k_{plist} is the number of m-subsequences covering Q and is calculated differently depending on whether $Len(Q) < m$ or $Len(Q) \geq m$. If $Len(Q) \geq m$, the number of m-subsequences $S_{i+1}, \ldots, S_{j-1}$ in Fig. 10a is $(Len(Q) - m + 1)$, and that of m-subsequences S_i or S_j is $\sum_{i=0}^{m-n-1} |\Sigma|^{m-n-i}$. If $Len(Q) < m$, the number of an m-subsequence S_k in Fig. 10b is $\left((m - Len(Q) + 1) \times |\Sigma|^{m-Len(Q)}\right)$, and that of m-subsequences S_p or S_q is $\sum_{i=0}^{Len(Q)-n-1} |\Sigma|^{m-n-i}$. Hence, the query processing time in the back-end index is as in Eq. (8). Finally, Eq. (9) shows the ratio of the query processing times.

$$\text{time}_{n\text{gram}} = \frac{\text{size}_{n\text{gram}}}{|\Sigma|^n} \times \frac{Len(Q)}{n} \tag{6}$$

$$\text{time}_{\text{front}} = \frac{\text{size}_{\text{front}}}{|\Sigma|^n} \times (Len(Q) - n + 1) \tag{7}$$

$$\text{time}_{\text{back}} = \begin{cases} \frac{\text{size}_{\text{back}}}{|\Sigma|^m} \times \left(Len(Q) - m + 1 \ + \ 2\sum_{i=0}^{m-n-1} |\Sigma|^{m-n-i} \right), & \text{if } Len(Q) \geq m \\ \frac{\text{size}_{\text{back}}}{|\Sigma|^m} \times \left((m - Len(Q) + 1) \times |\Sigma|^{m-Len(Q)} + 2\sum_{i=0}^{Len(Q)-n-1} |\Sigma|^{m-n-i} \right), & \text{if } Len(Q) < m \end{cases} \tag{8}$$

$$\frac{\text{time}_{n\text{gram}}}{\text{time}_{\text{front}} + \text{time}_{\text{back}}} = \begin{cases} \frac{\text{size}_{n\text{gram}} \times \frac{Len(Q)}{n}}{(\text{size}_{\text{front}} \times (Len(Q)-n+1)) + \left(\text{size}_{\text{back}} \times (\frac{Len(Q)-m+1}{|\Sigma|^{m-n}} + c)\right)}, & \text{if } Len(Q) \geq m \\ \frac{\text{size}_{n\text{gram}} \times \frac{Len(Q)}{n}}{(\text{size}_{\text{front}} \times (Len(Q)-n+1)) + \left(\text{size}_{\text{back}} \times (\frac{m-Len(Q)+1}{|\Sigma|^{Len(Q)-n}} + d)\right)}, & \text{if } Len(Q) < m \end{cases} \tag{9}$$

$$\text{where } c = 2\sum_{i=0}^{m-n-1} (\tfrac{1}{|\Sigma|})^i, \ d = 2\sum_{i=0}^{Len(Q)-n-1} (\tfrac{1}{|\Sigma|})^i$$

From Eq. (9), we know that the time complexities of those indexes are identical to their space complexities. By substituting $\text{size}_{n\text{gram}}$ with $|\mathbb{S}|(\text{avg}_{n\text{gram}} \times \text{avg}_{\text{doc}})$, Eq. (9) shows that the time complexity of the n-gram index is $O(|\mathbb{S}|(\text{avg}_{n\text{gram}} \times \text{avg}_{\text{doc}}))$ while that of the n-gram/2L index is $O(|\mathbb{S}|(\text{avg}_{n\text{gram}} + \text{avg}_{\text{doc}}))$. The time complexity indicates that the n-gram/2L index has a good characteristic that the query performance improves compared with the n-gram index, and further, the improvement gets larger as the database size gets larger.

From Eqs. (6)–(9), we note that the query processing time increases at a lower rate in the n-gram/2L index than in the n-gram index as $Len(Q)$ gets longer. In the front-end index, the query processing time increases proportionally to $Len(Q)$, but it contributes a very small proportion of the total query processing time because the index size is very small. The size of the front-end index is much smaller than that of the n-gram index because the total size of m-subsequences is much smaller than the total size of documents (for example, when the size of the document collection is 1 GB and $m = 5$, the size of the set of m-subsequences is 13–27 MB). Furthermore, in the back-end index, $Len(Q)$ little affects the query processing time since $|\Sigma|^{m-n}$ is dominant (for example, when $|\Sigma| = 26$, $m = 6$, and $n = 3$, $|\Sigma|^{m-n} = 17,576$). This is also an excellent property since it has been pointed out that the query performance of the n-gram index for long queries tends to be bad [30].

To analyze the query processing time more precisely, we should take the time to locate posting lists into account. Suppose that α is the seek time required for locating a posting list. Then, the total time for locating posting lists is $k_{\text{plist}} \times \alpha$. Hence, by using k_{plist} computed in Eqs. (6)–(8), we derive the time for locating posting lists as shown in Eqs. (10)–(12).

$$\text{plist\_time}_{n\text{gram}} = \alpha \times \frac{Len(Q)}{n} \tag{10}$$

$$\text{plist\_time}_{\text{front}} = \alpha \times (Len(Q) - n + 1) \tag{11}$$

$$\text{plist\_time}_{\text{back}} = \begin{cases} \alpha \times \left(Len(Q) - m + 1 \ + \ 2\sum_{i=0}^{m-n-1} |\Sigma|^{m-n-i} \right), & \text{if } Len(Q) \geq m \\ \alpha \times \left((m - Len(Q) + 1) \times |\Sigma|^{m-Len(Q)} + 2\sum_{i=0}^{Len(Q)-n-1} |\Sigma|^{m-n-i} \right), & \text{if } Len(Q) < m \end{cases} \tag{12}$$

From Eqs. (10)–(12), the time for locating posting lists of the n-gram/2L index is larger than that of the n-gram index by $\text{plist\_time}_{\text{back}}$ at least. In Eq. (12), the dominant factor of k_{plist} is $\left(2\sum_{i=0}^{m-n-1} |\Sigma|^{m-n-i}\right)$ if $Len(Q) \geq m$ and $\left(2\sum_{i=0}^{Len(Q)-n-1} |\Sigma|^{m-n-i}\right)$ if $Len(Q) < m$, where these values increase exponentially as m gets larger. Hence, if we select $(m_o - 1)$ instead of m_o for the length of m-subsequences, we can significantly improve the query performance while sacrificing a small increment of the index size. Consequently, we use $(m_o - 1)$ for performance evaluation in Sect. 6.2.

4.3.2 n-gram/2L-v index

The time complexity of the n-gram/2L index for v-subsequences is identical to that for m-subsequences just like the space complexity. Yet, the query performance of the n-gram / 2L-v index is improved due to its smaller size compared with the n-gram/2L-m index.

Another advantage of the n-gram/2L-v index is that the time for locating posting lists decreases compared with the n-gram/2L-m index. It is because the number of posting lists in the n-gram/2L-v index decreases since we have fewer unique v-subsequences than unique m-subsequences as mentioned in Sect. 4.2.2. Thus, we use v_o as the base length of v-subsequences for performance evaluation in Sect. 6.2.

5 Implementation issues

In this section, we discuss implementation of the n-gram/2L index to improve the efficiency of the index structure. As explained in the query processing algorithm in Fig. 11, we obtain a set of m-subsequence identifiers (of integer type) by searching the front-end index in Step 1. To proceed into Step 2, we need to do the following: (1) to find m-subsequences using the m-subsequence identifiers obtained in Step 1; (2) to locate the posting list in the back-end index for each m-subsequence through the B+-tree index. These two operations may impose significant overhead to query processing performance.

To solve these problems, we store a physical identifier rather than an m-subsequence identifier into the posting of the front-end index. Here, the physical identifier directly points to the posting list in the back-end index. In general, a physical identifier consumes more storage space than an m-subsequence identifer does. However, it does not pose a significant overhead since the size of the front-end index itself is very small as mentioned in Sect. 4.3.1. We adopt this enhanced implementation in this paper.

6 Performance evaluation

6.1 Experimental data and environment

The purpose of our experiments is to compare the size and query performance of the n-gram/2L index with those of the n-gram index. We use the *index size ratio* defined in Eq. (13) as the measure for the index size and the number of page accesses and the wall clock time for the query performance.

$$\text{index size ratio} = \frac{\text{the number of pages allocated for the } n\text{-gram index } (\#\text{Pages}_{n-\text{gram}})}{\text{the number of pages allocated for the } n\text{-gram/2L index } (\#\ \text{Pages}_{n-\text{gram}/2L})} \tag{13}$$

We have performed experiments using two real data sets. The first one is the set of English text databases—WSJ, AP, and FR in the TREC databases[1]—used in information retrieval. We use three data sets of 10 MB, 100 MB, and 1 GB. We call each data set TREC-10M, TREC-100M, and TREC-1G, respectively. The second one is the set of protein sequence databases—nr, env_nr, month.aa, and pataa in the NCBI BLAST web site[2]—used in bioinformatics. We use three data sets of 10 MB, 100 MB, and 1 GB. We call each data set PROTEIN-10M, PROTEIN-100M, PROTEIN-1G, respectively. We remove tags, spaces, special characters, and numbers in the TREC databases making the formats of the TREC data and the PROTEIN data similar in order to exclude the influence of the format to the results of the experiments.

We conduct all the experiments on a Pentium 2.6 GHz Linux PC with 1 GB of main memory and 400 GB Seagate E-IDE disks. To avoid the buffering effect of the LINUX file system and to guarantee actual disk I/O's, we use raw disks for storing data and indexes. We use the inverted index implemented in the Odysseus ORDBMS [29] for all the experiments. The page size for data and indexes is set to be 4,096 bytes.

Experiments for data size

To compare the index size, we measure the estimated and real index size ratios in the PROTEIN and TREC databases while varying m. We use the decomposition efficiency (Eq. 5) presented in Sect. 4.2 to estimate the index size ratio. Then, we show that the estimation of m_o is correct. When creating the n-gram index and the front-end index, we set n to be 3, which is the most practically used one in n-gram applications [14,31]. Besides, when creating the back-end index, we vary m from 4, the minimum of m (i.e., $n+1$), to (m_o+1). Here, if $m = n$, the back-end index is identical to the n-gram index. Thus, m must be longer than n, that is, $m > n$.

To demonstrate the effectiveness of v-subsequences, we compare the size the n-gram/2L-v index with that of the n-gram/2L-m index. We measure the estimated and real index size ratios in the TREC databases while varying v. We show that the estimated v_o is identical to the real v_o. When creating the front-end index, we set n to be 3 as before. Besides, when creating the back-end index, we vary v from 3, the minimum of v (i.e., n), to (v_o+1). We note that the minimum of v is n rather than $n+1$. The reason is that, even if $v = n$, the back-end index is not identical to the n-gram index since the length of v-subsequences can be longer than v. If $v < n$, n-grams can not be extracted from v-subsequences since the length of v-subsequences can be shorter than n.

[1] http://trec.nist.gov.

[2] http://www.ncbi.nlm.nih.gov/BLAST.

Experiments for query performance

To compare the query performance, we measure the number of page accesses and the wall clock time while varying the database size and the query length. To test the effect of the database size, we build three databases of 10 MB, 100 MB, and 1 GB using the PROTEIN data and TREC data, respectively. Here, we repeat the test 100 times with randomly selected queries whose length is 3–18 and present the average result. To test the effect of the query length, we vary the query length as follows: 3, 6, 9, 12, 15, and 18. We note that the minimum query length is 3, which is the same as $n = 3$. Using PROTEIN-1G and TREC-1G, we repeat the test 50 times with randomly selected queries and present the average result.

We compare the query performance of four indexes. The first one is the n-gram index where a query string is split into overlapping n-grams (by the 1-sliding technique). The second one is the n-gram index where a query string is split into disjoint n-grams. In this way, we are able to improve the query performance of the n-gram index for exact-match queries as explained in Sect. 2.1. Hereafter, we call this n-gram index as the *n-gram-disjoint* index. The third one is the n-gram/2L-m index. The fourth one is the n-gram/2L-v index (only for the TREC databases).

Experiments for comparing with the compact suffix array

We compare the query performance of the n-gram/2L index with that of Makinen's compact suffix array [17] stored in disk. The suffix array is also widely used for text search. Nevertheless, it has been pointed out as a problem that the size of the suffix array tends to be large. In recent years, a number of approaches that reduce its size by compression have been published, and Makinen's compact suffix array (simply, the CSA) is one whose size is similar to that of the n-gram/2L index.

For the experiments, we modify the CSA into a disk-based index structure because the CSA is a memory-based one. When building the index, the CSA requires a large amount of memory (e.g., 10 GB of main-memory for 1 GB of text) and a large number of random accesses on temporary arrays in main memory, which would cause extreme overhead if it were run in a disk environment. Hence, we modify the index building algorithm so as to use relatively small memory (e.g., 1 GB of main-memory for 1 GB of text) and access temporary arrays on disk as sequential as possible. After building the index, we save the final arrays on disk. When processing queries, we access those arrays as if they were in main memory.

To compare the query performance, we measure the number of page accesses and the wall clock time while varying the query length as follows: 3, 6, 9, 12, 15, 18, 21, 24, and 27. Using PROTEIN-1G and TREC-1G, we repeat the test 50 times with randomly selected queries and present the average result.

6.2 Results of the experiments

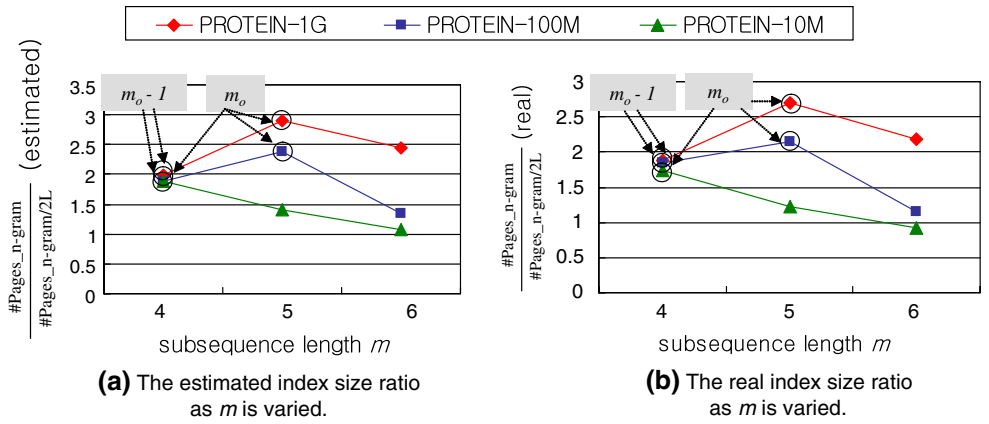
6.2.1 Index size

Figure 14 shows the estimated and real index size ratios as the database size and length of m-subsequences are varied in the PROTEIN databases. These results indicate that the size of the n-gram/2L index is significantly reduced compared with that of the n-gram index. Figure 14b shows that the size of the n-gram/2L index, when the length of m-subsequences is set to be m_o, is reduced by up to 1.7 times in PROTEIN-10M, by up to 2.2 times in PROTEIN-100M, and by up to 2.7 times in PROTEIN-1G compared with that of the n-gram index. Similarly, the size of the n-gram/2L index, when $m = (m_o - 1)$, is reduced by up to 1.85 times in PROTEIN-100M and by up to 1.88 times in PROTEIN-1G compared with that of the n-gram index. We use $(m_o - 1)$ as m to optimize the query performance as mentioned in Sect. 4.3.1.

We note that the estimated m_o in Fig. 14a is identical to the real m_o in Fig. 14b. For PROTEIN-10M, PROTEIN-100M, and PROTEIN-1G, both estimated m_o and real m_o are 4, 5, and 5, respectively. Besides, the real index size ratio is as close as 86–98% of the estimated one, showing a small amount of errors (2–14%) in estimation. These results indicate that the analysis in Sect. 4.2 is indeed correct.

Figures 15 and 16 show the estimated and real index size ratios as the database size and length of m-subsequences (or v-subsequences) are varied in the TREC databases. As mentioned in Sect. 4.2.2, the index size ratio of the n-gram/2L-v index in Fig. 16b improves over that of the n-gram/2L-m index in Fig. 15b. Figure 15b shows that the size of the n-gram/2L-m index, when the length of m-subsequences is set to be $(m_o - 1)$, is reduced by up to 1.28 times in TREC-10M, by up to 1.68 times in TREC-100M, and by up to 1.88 times in TREC-1G compared with that of the n-gram index. On the other hand, Fig. 16b shows that the size of the n-gram/2L-v index, when the length of v-subsequences is set to be v_o, is reduced by up to 1.44 times in TREC-10M, by up to 1.84 times in TREC-100M, and by up to 2.42 times in TREC-1G compared with that of the n-gram index. Thus, the size of the n-gram/2L-v index is reduced by up to 29% (TREC-1G, $v = 4$) compared with that of the n-gram/2L-m index.

Figures 15 and 16 show that the estimated m_o (or v_o) is identical to the real m_o (or v_o). However, the real index size ratio is shown to be 52–73% of the estimated one, showing a larger amount of errors compared with Fig. 14. It is because the inverted index, in real implementation, stores document identifiers or m-subsequence identifiers in addition to offsets

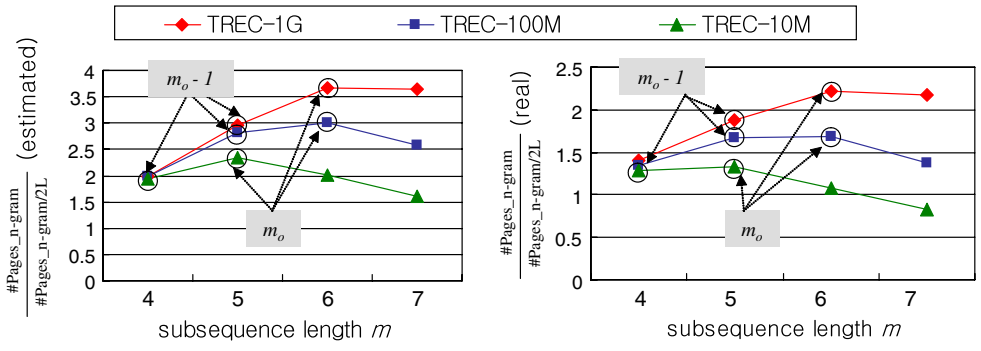


(a) The estimated index size ratio as m is varied.

(b) The real index size ratio as m is varied.

Fig. 14 The estimated and real index size ratio in the PROTEIN databases when using m-subsequences

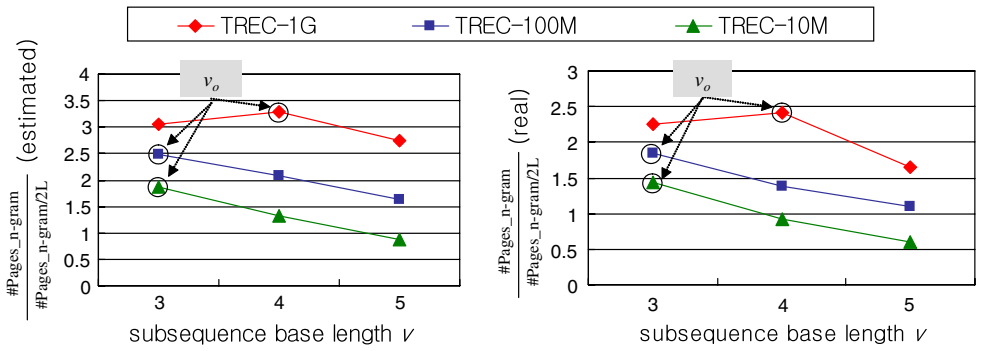
Fig. 15 The estimated and real index size ratio in the TREC databases when using m-subsequences



(a) The estimated index size ratio as m is varied.

(b) The real index size ratio as m is varied.

Fig. 16 The estimated and real index size ratio in the TREC databases when using v-subsequences



(a) The estimated index size ratio as v is varied.

(b) The real index size ratio as v is varied.

while we count only offsets in estimation. The identifier of a document (or an m-subsequence) and offsets at which a term occurs are maintained as a unit in a *posting* (see Fig. 3). Here, the space required for identifiers is relatively larger in the n-gram/2L index because the average number of offsets in a posting is smaller than in the n-gram index, thus making the estimation to deviate from the real one. This difference of the average number of offsets is due to duplication of offsets in the n-gram index and its elimination in the n-gram/2L index. This tendency is more marked in Fig. 15 and 16 than in Fig. 14 because, in TREC data, a collection of magazines or papers, the words or expressions are more frequently repeated.

Table 2 shows that the index size ratio increases as the database size does. Here, the index size ratio is obtained by using $(m_o - 1)$ as the length of m-subsequences or by using v_o as the base length of v-subsequences. This result confirms our analysis in Sect. 4.2. When we use m-subsequences, as the database size is varied by ten fold from 10 MB to 1 GB,

Table 2 The index size ratio as the database size is varied

Data set	10 MB	100 MB	1 GB
PROTEIN	1.734 ($m_o = 4$)[†]	1.847 ($m_o - 1 = 4$)	1.877 ($m_o - 1 = 4$)
TREC	1.281 ($m_o - 1 = 4$)	1.677 ($m_o - 1 = 5$)	1.878 ($m_o - 1 = 5$)
	1.437 ($v_o = 3$)	1.841 ($v_o = 3$)	2.417 ($v_o = 4$)

the index size ratio increases by 2% in the PROTEIN databases, and by 21% in the TREC databases on the average when $m = (m_o - 1)$. Besides, when we use v-subsequences, the index size ratio increases by 29% in the TREC databases on the average when $v = v_o$. Here, we use not $(m_o - 1)$ but m_o as the length of m-subsequences for PROTEIN-10M since m must be longer than n and $(m_o - 1) = 3$ is not longer than $n = 3$ in PROTEIN-10M. [†] The case when $(m_o - 1) = 3$ is not available.

6.2.2 Query performance

Figure 17a shows the query processing time of the n-gram index and n-gram/2L index as the database size is varied for the PROTEIN database. Here, we set the length of m-subsequences to $(m_o - 1)$ to optimize the query performance as mentioned in Sect. 4.3.1. As indicated by the time complexity in Sect. 4.3, the n-gram/2L index significantly improves the query performance compared with the n-gram index. Further, we obtain a larger improvement as the database size gets larger. Figure 17a shows that the improvement in the query performance is 1.37 times in PROTEIN-100 MB and 6.65 times in PROTEIN-1 GB.

Figure 17b and c show the number of page accesses and query processing time as the query length is varied for PROTEIN-1G. We note that they increase at a lower rate in the n-gram/2L index than in the n-gram index as $Len(Q)$ gets longer. In Fig. 17b and c, as $Len(Q)$ is varied from 3 to 18, the number of page accesses for the n-gram/2L index increases only by 27% and the wall clock time only by 53% on the average, while those for the n-gram index increase by 12.0 times and by 32.9 times, respectively. In effect, the wall clock time—when considering queries shorter than six times of n—is improved by up to 13.1 times compared with those of the n-gram index.

Figure 18 shows the query performance in the TREC databases, showing a tendency similar to that in the PROTEIN databases. We observe that the number of page accesses of the n-gram/2L index is improved by up to 3.5 times, and the wall clock time by up to 2.9 times compared with those of the n-gram index.

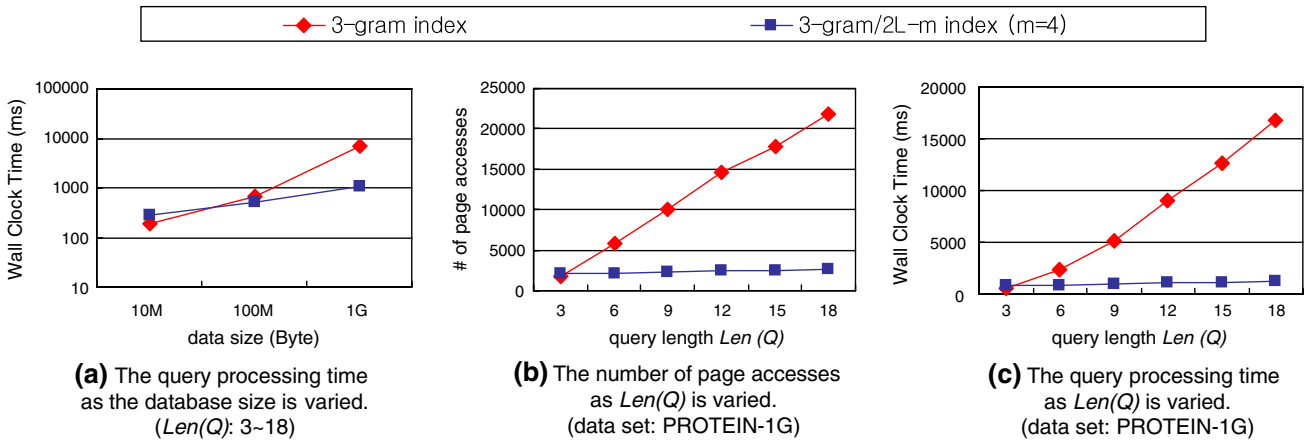
Figure 19 shows the query processing times of the n-gram-disjoint index and the n-gram/2L-m index as the database size is varied for the PROTEIN databases. The n-gram/2L-m index still outperforms the n-gram-disjoint index. We observe that the number of page accesses of the n-gram/2L-m index is improved by up to 2.1 times, and the wall clock time by up to 1.9 times compared with those of the n-gram-disjoint index.

Figure 20 shows the query processing times of the n-gram-disjoint index, the n-gram/2L-m index, and the n-gram/2L-v index as the database size is varied for the TREC databases. As shown in Fig. 20, the query performance of the n-gram-disjoint index tends to be close to that of the n-gram/2L-m index. However, the query performance of the n-gram/2L-v is still superior to that of the n-gram-disjoint index. This result indeed indicates the effectiveness of using v-subsequences. Figure 20b shows that the n-gram/2L-v index ($v = 4$) improves the number of page accesses by 2.1 times over the n-gram/2L-m index ($m = 5$) and by 2.9 times over the n-gram-disjoint index. Similarly, Fig. 20c shows that the n-gram/2L-v index ($v = 4$) improves the wall clock time by 2.3 times over the n-gram/2L-m index ($m = 5$) and by 2.0 times over the n-gram-disjoint index.

6.2.3 Comparison with the CSA

Figure 21 shows the number of page accesses and query processing time as the query length is varied for PROTEIN-1G. When the query length is short (e.g., $Len(Q) = 3$), we note that the query performance of the CSA is much poorer than that of the n-gram/2L index. However, as the query length increases, we observe that the query performance of the CSA tends to be close to or better than that of the n-gram/2L index. The reason is that, as $Len(Q)$ gets longer, the CSA reads less position information (i.e., offsets) from the position array, while the n-gram/2L index reads more offsets from more posting lists. The number of offsets that the CSA reads is proportional to the number of query results, which decreases as $Len(Q)$ gets longer.

We also note that the difference between the query performance of the n-gram/2L index and that of the CSA is larger in the wall clock time as in Fig. 21b than in the number of page accesses as in Fig. 21a. This is due to random accesses accompanying disk seeks frequently occurring in the CSA. In the n-gram/2L-index, the offsets to access for processing a query are consecutively stored in each index page and, at the same time, the pages to access for processing a query are consecutively stored on disk for each posting list. Thus, the n-gram/2L-index obtains the effect of sequential access and disk cache. However, in the CSA, the offsets to access for processing a query are scattered over the position array.

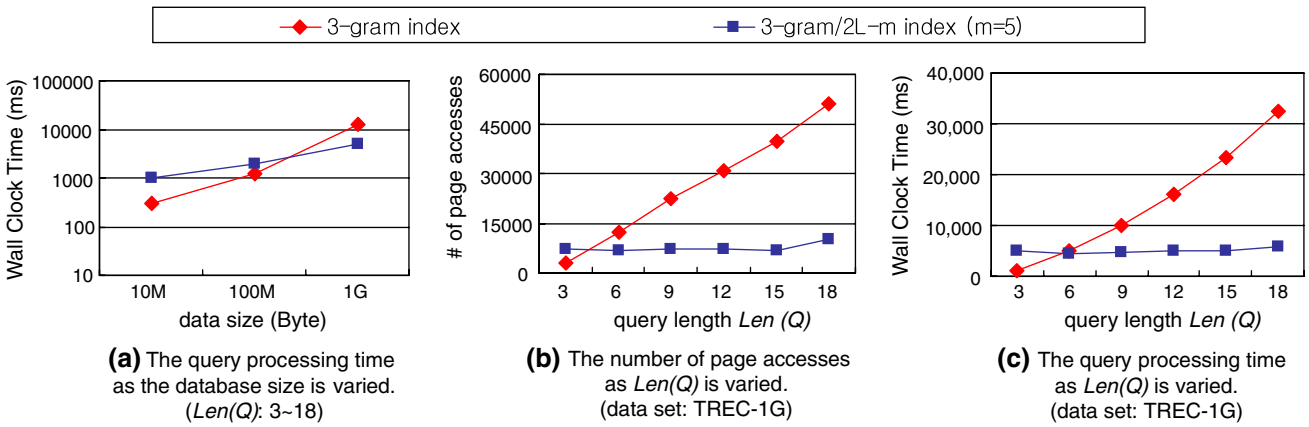


(a) The query processing time as the database size is varied. (*Len(Q)*: 3~18)

(b) The number of page accesses as *Len(Q)* is varied. (data set: PROTEIN-1G)

(c) The query processing time as *Len(Q)* is varied. (data set: PROTEIN-1G)

Fig. 17 The query performance for the PROTEIN databases

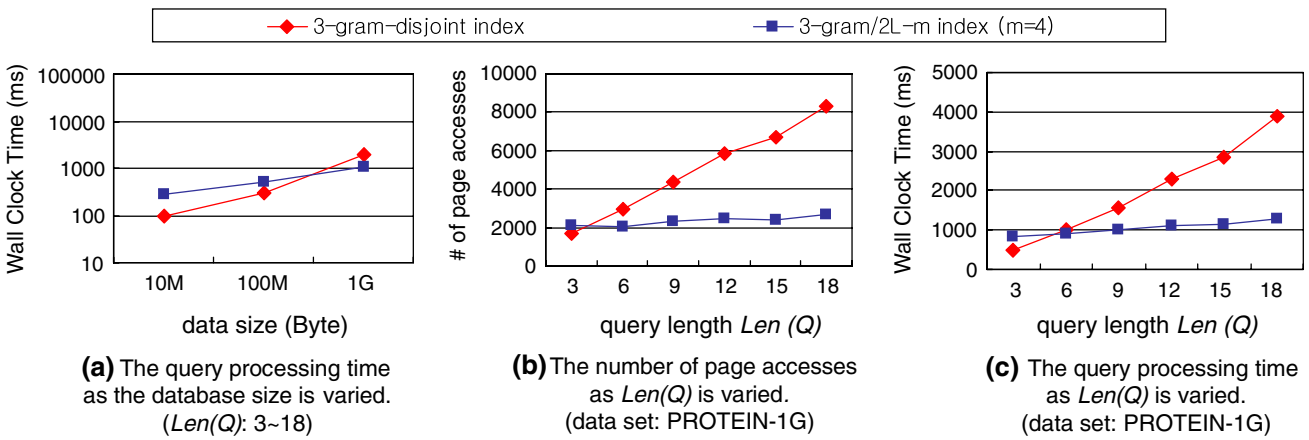


(a) The query processing time as the database size is varied. (*Len(Q)*: 3~18)

(b) The number of page accesses as *Len(Q)* is varied. (data set: TREC-1G)

(c) The query processing time as *Len(Q)* is varied. (data set: TREC-1G)

Fig. 18 The query performance for the TREC databases

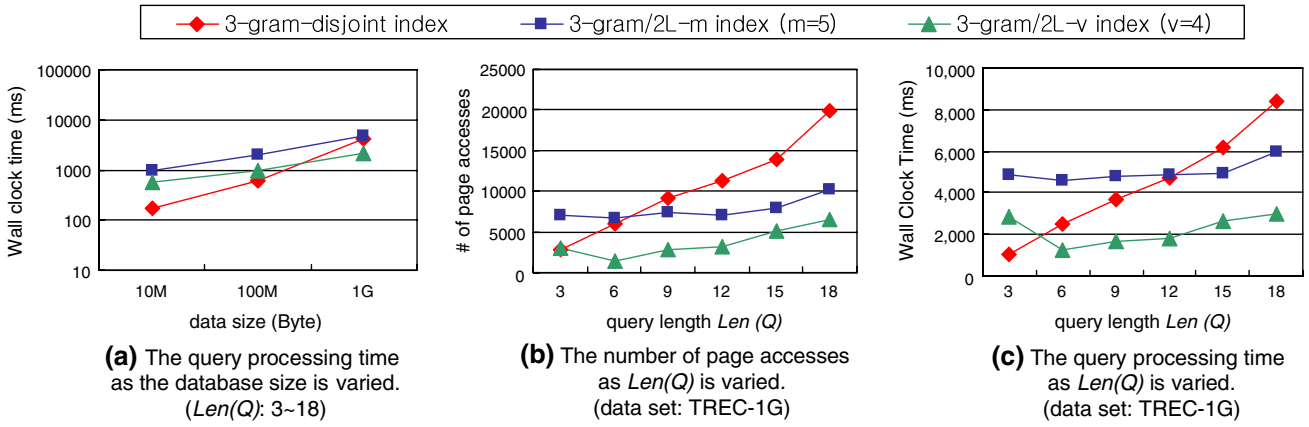


(a) The query processing time as the database size is varied. (*Len(Q)*: 3~18)

(b) The number of page accesses as *Len(Q)* is varied. (data set: PROTEIN-1G)

(c) The query processing time as *Len(Q)* is varied. (data set: PROTEIN-1G)

Fig. 19 The query performance for the PROTEIN databases when using the *n*-gram-disjoint index and the *n*-gram/2*L*-*m* index

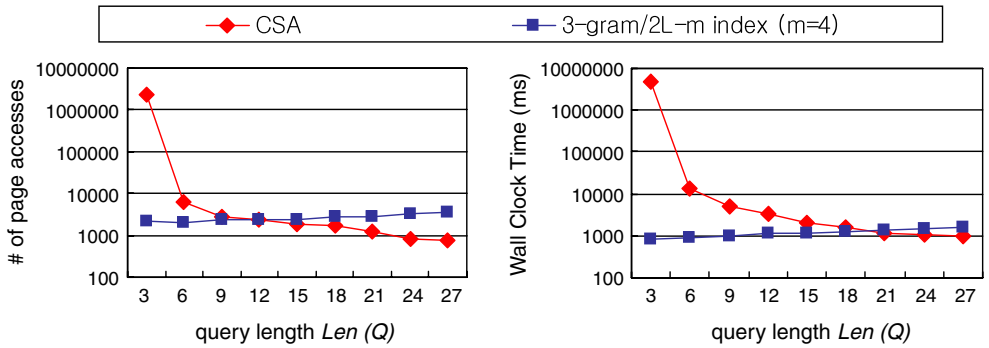


(a) The query processing time as the database size is varied. (*Len(Q)*: 3~18)

(b) The number of page accesses as *Len(Q)* is varied. (data set: TREC-1G)

(c) The query processing time as *Len(Q)* is varied. (data set: TREC-1G)

Fig. 20 The query performance for the TREC databases when using the n-gram-disjoint index and the n-gram/2L-v index



(a) The number of page accesses as *Len(Q)* is varied. (data set: PROTEIN-1G)

(b) The query processing time as *Len(Q)* is varied. (data set: PROTEIN-1G)

Fig. 21 Query performance for PROTEIN-1G when using the CSA and the n-gram/2L-m index

Thus, the CSA cannot take advantage of sequential access and disk cache.

Figure 22 shows the query performance for TREC-1G, showing a tendency similar to that for PROTEIN-1G. In conclusion, the n-gram/2L index outperforms the CSA when $Len(Q)$ is short (i.e., less than 15–20), and the CSA is similar to or better than the n-gram/2L index when $Len(Q)$ is long (i.e., more than 15–20). That is, there exists some crossing point. This result coincides with that of the experiments done in work by Puglisi et al. [24], in which they proposed a memory-based compressed n-gram index, compared it with the compressed suffix array in a memory environment, and found that their compressed n-gram index is better for short query strings and worse for long query strings than the compressed suffix array.

7 Related work

In this section, we briefly introduce prior art related to the n-gram/2L index—especially, ones on the inverted index, the n-gram index, and the two-level indexing approach. Further, we also introduce the suffix array (or the suffix tree), which is often compared with the inverted index.

Inverted index

Due to explosive use of text search engines such as Google [4], a significant number of research results on the inverted index, which is the essential component of the text search engine, have been published over the last decade. On the issue of reducing the index size, most research focused on compressing posting lists. Compression of posting lists is typically done by efficiently encoding gaps between document identifiers [26], but in some cases is done by using Lempel-Ziv parsing [10]. On the issue of improving scalability and query throughput, research has been done on building a distributed inverted index over a number of machines [4,19]. This scheme is essential for commercial systems that handle very large scale databases. A detailed survey on the inverted index can be found in the paper by Zobel and Moffat [34].

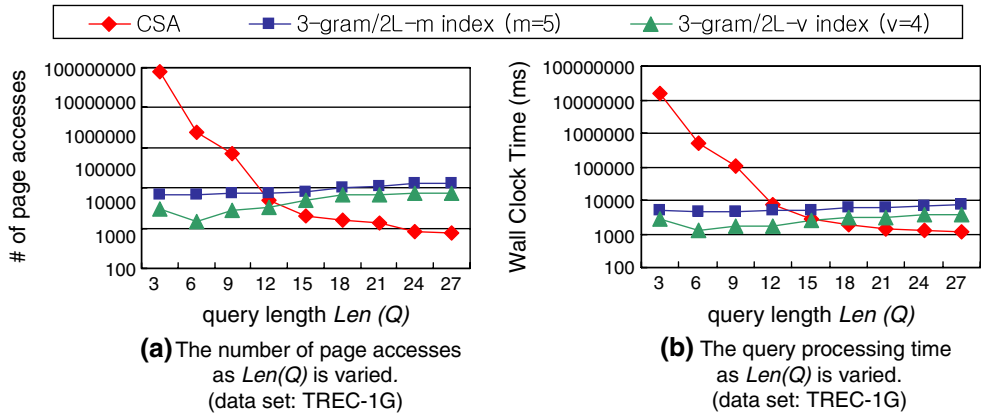


Fig. 22 Query performance for TREC-1G when using the CSA and the n-gram/2L-v index

n-Gram index

There have been a number of efforts to use the n-gram index for various applications. The n-gram index has two major advantages: language-neutral and error-tolerant. Since it is language-neutral, the n-gram index is often used in information retrieval for Asian languages such as Korean [15], Japanese [33], and Chinese [7], where extraction of words is not simple. Since it is error-tolerant, the n-gram index is often used in approximate string matching, where the query processing algorithm performs the following two steps: (1) finding candidate documents (or substring) by searching the n-gram index with n-grams extracted from a query string (the filtration step); (2) doing refinement in order to find final results by using online algorithms (the refinement step). Approximate string matching algorithms using the n-gram index are classified into two categories depending on the filtration method [21]: (1) the algorithms that find documents in which some substrings of a query appear *without errors* as candidate results [1,27,31]; (2) those that find candidate documents in which some substrings of a query appear *with a few errors* as candidate results [12,23]. n-gram/2L-approximation [12], which is a variation of the n-gram/2L index proposed for approximate searching by the authors, belongs to this category. More results on approximate string matching can be found in the paper by Navarro et al. [21].

Two-level indexing approach

There have been some studies that use the two-level indexing approach. The block addressing scheme is often used for reducing the size of the inverted index. This scheme divides a text database into blocks and stores block offsets where the n-gram appears instead of character offsets [2,16]. For query processing, it first searches for blocks that a query appears, and then, performs online search over the retrieved blocks for finding exact positions. Although it performs query processing in two steps, it does not use a real two-level index. Some other approaches use the two different kinds of data structures for indexing. For example, Cao et al. [5] uses a hash table as the first-level index and a signature tree as the second-level index for biological sequences.

Suffix array (or suffix tree)

For text search, the suffix array is also widely used. The suffix array is inherently a memory-based index structure while the inverted index is a disk-based one. The suffix array is suitable for relatively small databases [34]. It has been pointed out as a problem that the size of the suffix array tends to be large. But, in recent years, a number of approaches that reduce its size by compression have been reported. Most of them compress the index structure based on text entropy [8,17], but some others do that based on Lempel-Ziv parsing [11]. In spite of compression, the suffix array might not always fit in main memory. Some approaches touched the issue of using the suffix tree on disk [9], but there is very little work on this issue yet. More work on the suffix array can be found in the survey paper by Navarro and Makinen [22].

8 Conclusions

In this paper, we have proposed the n-gram/2L index that significantly reduces the size and improves the query performance compared with the n-gram index. The novelty of our approach lies in finding the redundancy of the position information that exists in the n-gram index and eliminating that redundancy. To eliminate the redundancy, we construct the inverted index in two steps: (1) extracting $n-1$ overlapping m-subsequences (or v-subsequences) from documents and building the back-end index; and (2) extracting n-grams from those subsequences and building the front-end index. Here, v-subsequences, which are variable-length

subsequences, allow us to further enhance the size and the query performance of the n-gram/2L index by exploiting words in extracting subsequences for natural language documents.

We have theoretically analyzed the properties of the n-gram/2L index. First, we have formally proven in Lemma 2 that the redundancy of the position information that exists in the n-gram index is due to a non-trivial MVD. Then, we have proven in Lemma 3 and Theorem 2 that our index is derived by the relational normalization process that decomposes the n-gram index into 4NF. Second, we have analyzed the space complexity and proposed the model for determining the optimal length of m (i.e., m_o) minimizing the index size. Since the space complexity of our index is $O(|\mathbb{S}|(\text{avg}_{n\text{gram}} + \text{avg}_{\text{doc}}))$ and that of the n-gram index is $O(|\mathbb{S}|(\text{avg}_{n\text{gram}} \times \text{avg}_{\text{doc}}))$, the reduction of the index size becomes more marked as the database size gets larger. Third, we have analyzed the time complexity. Since the time complexity is shown to be the same as the space complexity, the improvement of the query performance becomes more marked as the database size gets larger. Besides, we have found out that we can speed up query processing by small sacrifice in the index size (i.e., by using $(m_o - 1)$ as the length of m-subsequences.) Fourth, we have shown that the query processing time increases only very slightly as the query length gets longer by using Eq. (9).

We have performed extensive experiments for the size and query performance of the n-gram/2L index varying the data set, database size, query length, m-subsequences length, and v-subsequences length. We have used $(m_o - 1)$ as the length of m-subsequences to speed up query processing. Experimental results using real text and protein databases of 1 GB show that the size of the n-gram/2L index is reduced by up to 1.9 (TREC-1G, $m = 5$ and PROTEIN-1G, $m = 4$) times and, at the same time, the query performance—when considering queries shorter than six times of n—is improved by up to 2.9 (TREC-1G, $m = 5$)–13.1 (PROTEIN-1G, $m = 4$) times compared with those of the n-gram index. For text databases, we have shown that using v-subsequences further improves the size and query performance for text databases. Experimental results show that the size of the n-gram/2L-v index is reduced by up to 29% (TREC-1G, $v = 4$) and, at the same time, the query performance is improved by up to 2.27 (TREC-1G, $v = 4$) times compared with those of the n-gram/2L-m index (TREC-1G, $m = 5$). We have also compared the query performance of the n-gram/2L index with that of Makinen's CSA [17] stored in disk. Experimental results show that the n-gram/2L index outperforms the CSA when $Len(Q)$ is short (i.e., less than 15–20), and the CSA is similar to or better than the n-gram/2L index when $Len(Q)$ is long (i.e., more than 15–20).

Overall, these results indicate that the n-gram/2L index is a new structure that can replace the n-gram index in many applications including information retrieval.

Acknowledgment A preliminary version of this paper has appeared in the proceedings of the 31th International Conference on Very Large Data Bases held in Trondheim, Norway, in August/September 2005 [13]. We have significantly extended this version with the new notion of the v-subsequence, algorithms, and extensive experiments including comparison with the compact suffix array [17]. We would like to thank the anonymous reviewers for their incisive comments that made the paper more complete and readable. This work was supported by the Korea Science and Engineering Foundation (KOSEF) and the Korean Government (MOST) through the NRL Program (No. R0A-2007-000-20101-0).

References

1. Baeza-Yates, R., Navarro, G.: A practical q-gram index for text retrieval allowing errors. CLEI Electron. J. **1**(2), (1998)
2. Baeza-Yates, R., Navarro, G.: Block addressing indices for approximate text retrieval. J. Am. Soc. Inf. Sci. **51**(1), 69–82 (2000)
3. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. ACM Press (1999)
4. Barroso, L.A., Dean, J., Holzle, U.: Web search for a planet: the google cluster architecture. IEEE Micro **23**(2), 22–28 (2003)
5. Cao, X., Li, S.C., Tung, A.K.H.: Indexing DNA sequences using q-grams. In: Proc. Int'l Conf. on Database Systems for Advanced Applications (DASFAA). Beijing, pp. 4–16 (2005)
6. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 4th edn. Addison Wesley (2003)
7. Gao, J., Goodman, J., Li, M., Lee, K.: Toward a unified approach to statistical language modeling for Chinese. ACM Trans. Asian Lang. Inf. Process. (TALIP) **1**(1), 3–33 (2002)
8. Grossi, R., Vitter, J.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. 32nd ACM Symposium on Theory of Computing (STOC), pp. 397–406 (2000)
9. Karkkainen, J., Rao, S.: 7. Full-text indexes in external memory. In: Algorithms for Memory Hierarchies pp. 149–170 (2003)
10. Karkkainen, J., Sutinen, E.: Lempel-Ziv index for q-grams. Algorithmica **21**(1), 137–154 (1998)
11. Karkkainen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string mathcing. In: Proc. 3rd South American Workshop on String Processing (WSP), pp. 141–155 (1996)
12. Kim, M., Whang, K., Lee, J.: n-gram/2L-approximation: a two-level n-gram inverted index structure for approximate string matching. J. Comput. Systems Sci. Eng. (2007) (to appear)
13. Kim, M., Whang, K., Lee, J., Lee, M.: n-Gram/2L: a space and time efficient two-level n-gram inverted index structure. In: Proc. the 31th Int'l Conf. on Very Large Data Bases (VLDB), Trondheim, pp. 325–336 (2005)
14. Kukich, K.: Techniques for automatically correcting words in text. ACM Comput Surv **24**(4), 377–439 (1992)
15. Lee, J.H., Ahn J.S.: Using n-grams for korean text retrieval. In: Proc. Int'l Conf. on Information Retrieval. ACM SIGIR, Zurich, pp. 216–224 (1996)
16. Lehtinen, O., Sutinen, E., Tarhio, J.: Experiments on block indexing. In: Proc. 3rd South American Workshop on String Processing pp. 183–193 (1996)
17. Makinen, V.: Compact suffix array. In: Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 305–319 (2000)
18. Mayfield, J., McNamee, P.: Single N-gram stemming. In: Proc. Int'l Conf. on Information Retrieval. ACM SIGIR, Toronto, pp. 415–416 (2003)

19. Melnik, S., Raghavan, S., Yang, B., Garcia-Molina, H.: Building a distributed full-text index for the Web. ACM Trans. Inf. Systems **19**(3), 217–241 (2001)
20. Miller, E., Shen, D., Liu, J., Nicholas, C.: Performance and scalability of a large-scale *N*-gram based information retrieval system. J. Digital Inf. **1**(5), 1–25 (2000)
21. Navarro, G., Baeza-Yates, R., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Eng Bull **24**(4), 19–27 (2001)
22. Navarro, G., Makinen, V.: Compressed full-text indexes. Technical report TR/DCC-2006-6, Department of Computer Science, University of Chile, (2006). (accepted to ACM Computing Surveys)
23. Navarro, G., Sutinen, E., Tanninen, J., Tarhio, J.: Indexing text with approximate q-grams. In: Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 350–363 (2000)
24. Puglisi, S., Smyth, W., Turpin, A.: Inverted files versus suffix arrays for locating patterns in primary memory. In: Proc. 13th Symposium on String Processing and Information Retrieval (SPIRE), Glasgow, pp. 122–133 (2006)
25. Ramakrishnan, R.: Database Management Systems. McGraw-Hill, New York (1998)
26. Scholer, F., Williams, H.E., Yiannis, J., Zobel, J.: Compression of inverted indexes for fast query evaluation. In: Proc. Int'l Conf. on Information Retrieval, ACM SIGIR, Tampere, pp. 222–229 (2002)
27. Sutinen, E., Tarhio, J.: Filtration with *q*-samples in approximate string matching. In: Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM), pp. 50–63 (1996)
28. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Vol. I. Computer Science Press, USA (1988)
29. Whang, K., Lee, M., Lee, J., Kim, M., Han, W.: Odysseus:a high-performance ORDBMS tightly-coupled with IR features. In: Proc. 21st IEEE Int'l Conf. on Data Engineering (ICDE), Tokyo, pp. 1104–1105, (2005) (this paper received the Best Demonstration Award)
30. Williams, H.E.: Genomic information retrieval. In: Proc. 14th Australasian Database Conferences, Adelaide, pp. 27–35 (2003)
31. Williams, H.E., Zobel, J.: Indexing and retrieval for genomic databases. IEEE Trans. Knowl. Data Eng. **14**(1), 63–78 (2002)
32. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn., Morgan Kaufmann (1999)
33. Yasushi, O., Masajirou, I.: A new character-based indexing method using frequency data for Japanese documents. In: Proc. Int'l Conf. on Information Retrieval, pp. 121–129. ACM SIGIR, Seattle (1995)
34. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput Surv **38**(2), (2006)