Practical rearrangement methods for dynamic double-array dictionaries

Shunsuke Kanda*, Yuma Fujita, Kazuhiro Morita, and Masao Fuketa

Graduate School of Advanced Technology and Science
Tokushima University, Tokushima, Japan

Abstract

Double-array structures have been widely used to implement dictionaries with string keys. Although the space efficiency of dynamic double-array dictionaries tends to decrease with key updates, we can still maintain high efficiency using existing methods. However, these methods have practical problems of time and functionality. This paper presents several efficient rearrangement methods to solve these problems. Through experiments using real-world datasets, we demonstrate that the proposed rearrangement methods are much more practical than existing methods.
Keywords: Double-array trie; Dynamic dictionary; Data structure; String processing

1 Introduction

An edge-labeled tree called a *trie* [1, 2] is widely used to store a set of strings. A trie is constructed by merging common string prefixes. Further, registered strings are extracted by concatenating the edge labels of root-to-leaf paths. Since there are nodes corresponding to each string, these strings can be mapped to unique identifiers. In addition to a simple exact matching operation, the trie can support powerful prefix-based operations used in specific applications such as natural language processing and information retrieval [3, 4, 5]. Therefore, many applications use tries to implement dictionaries with string keys.

Currently, a *double array (DA)* [6, 7] is widely used to implement tries. The DA structure supports extremely fast retrieval in practical spaces. Many studies have investigated static and dynamic DA tries. Most studies of static DA tries attempted to improve space efficiency and retrieval speed [8, 9, 10, 11, 12]. Static DA tries are typically used for dictionaries without frequent key updates (i.e., insertions and deletions), such as *Darts* [13] and *Darts-clone* [14]. With respect to dynamic DA tries, there are two main objectives. The first objective is to improve update time [15, 16, 17, 18]. Although the original DA trie [6] has been slow and unstable, a recent DA trie [19] enables update times that are close to

* *Corresponding author*: Shunsuke Kanda, Graduate School of Advanced Technology and Science, Tokushima University, Minamijosanjima 2-1, Tokushima 770-8506, Japan. Email: shnsk.knd@gmail.com, Telephone: +819043393422

those of a hashing dictionary, such as the C++ standard library's `std::unordered_map`. Therefore, there have been some applications using dynamic DA tries such as a full-text search engine [20] and text-stream processing [21]. The second objective is to improve space efficiency. A DA structure can include empty elements. Although its space efficiency depends on the *load factor* (i.e., the proportion of non-empty elements to the total elements), as in a hash table [22], existing methods [23, 24, 25, 26] can maintain a high load factor.

In this paper, we describe how the methods related to the second objective have practical problems of time and functionality. We found that maintaining a high load factor with the methods does not have many advantages. While the simplest solution is to rearrange a DA structure with arbitrary timing, simply applying these methods requires a significant amount of time. Therefore, we propose several rearrangement methods and evaluate their practical performance through experiments using read-world datasets. The experimental results demonstrated that the proposed methods provide significantly faster rearrangement. In addition, the proposed rearrangement methods can shorten basic dictionary operation runtimes. In other words, the proposed methods can also contribute to the first objective.

The remainder of this paper is organized as follows. In Section 2, we describe the data structure of DA trie dictionaries and discuss existing improvement methods for dynamic dictionaries. In Section 3, we propose our practical rearrangement methods for dynamic DA dictionaries. In Section 4, we present our evaluation of the proposed methods experimentally using real-world datasets. In Section 5, we introduce related studies pertaining to dynamic dictionaries. Moreover, we show the values of the DA dictionaries and the proposed methods. Section 6 presents conclusions and suggestions for future work.

2 Double-array trie dictionaries

This section describes the structure of the DA trie dictionary and related methods. In this paper, the number of trie nodes is denoted as n. String keys are drawn from a finite alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. Functions $\lfloor a \rfloor$ and $\lceil a \rceil$ denote the largest integer not greater than a and the smallest integer not less than a, respectively. For example, $\lfloor 2.4 \rfloor = 2$ and $\lceil 2.4 \rceil = 3$. Note that the base of the logarithm is 2 throughout this paper.

2.1 Data structure

A DA structure [6] represents a trie using BASE and CHECK arrays, where BASE$[s]$ and CHECK$[s]$ correspond to node s. The BASE and CHECK element corresponding to node s is denoted as *element* s. Assuming the existence of an edge from internal node s to node t with label c, the DA satisfies the following equations:

$$\text{BASE}[s] \oplus c = t \text{ and } \text{CHECK}[t] = s. \tag{1}$$

In other words, the DA can find child t from node s with label c by the following two steps: the child is obtained by $t \leftarrow \text{BASE}[s] \oplus c$; and we can identify whether the child exists by comparing CHECK$[t]$ to s. Parent s of node t is simply obtained by CHECK$[t]$. The most significant advantage of DAs is the extremely fast node-to-node traversal.

BASE and CHECK can include *empty elements* to satisfy Eq. (1). Here, EMPTY$[s] \in \{1, 0\}$ indicates whether element s is empty. As empty elements arise in the internal BASE and CHECK elements, EMPTY$[0]$ = EMPTY$[N-1] = 0$ always holds, where N is the length of the array. Note that element 0 always corresponds to the root. A set of addresses of empty elements is denoted as $R = \{0 \leq s < N \mid \text{EMPTY}[s] = 1\}$; i.e., $N = n + m$ holds, where $m = |R|$ is the number of empty elements. The load factor of BASE and CHECK is denoted as $\alpha = n/N$ $(0 \leq \alpha \leq 1)$.

2.2 Dictionary implementation

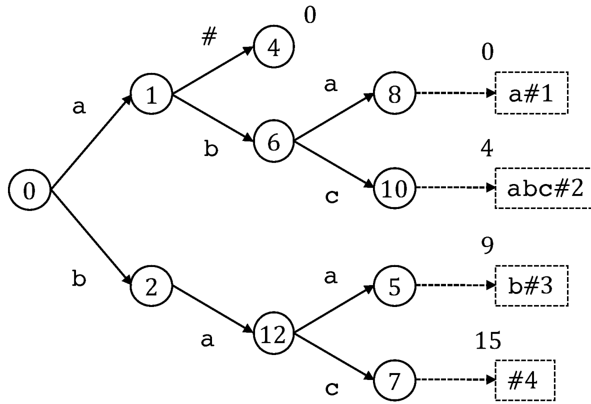
A *minimal-prefix trie (MP-trie)* [6, 7, 27] is often used to implement dictionaries. The MP-trie maintains only minimal prefixes to identify each key as nodes and the remaining suffixes as strings separately. The separated strings are stored in a TAIL array. Links to TAIL are kept in the BASE values of leaves. Since the number of nodes is fewer compared to a normal trie, the MP-trie can implement a compact DA dictionary. Moreover, sequential search on TAIL can improve retrieval speed. In dynamic dictionaries, useless spaces can arise in TAIL due to key insertions and deletions. We refer to these spaces as *empty spaces* to differentiate them from the empty elements of BASE and CHECK. The load factor of TAIL is denoted as $0 \leq \beta \leq 1$.

Figure 1 shows an example of an MP-trie dictionary and the DA representation for key-value pairs (`a#`, 0), (`abaa#`, 1), (`abcabc#`, 2), (`baab#`, 3), and (`bac#`, 4) from the alphabet $\{\texttt{\#} = 0, \texttt{a} = 1, \texttt{b} = 2, \texttt{c} = 3\}$. The terminator `#` (basically, the ASCII zero code) is added such that each leaf corresponds to each key. The DA dictionary uses a LEAF array such that LEAF$[s] \in \{1, 0\}$ indicates whether element s exists for a leaf node or not. If node s has a link to TAIL, the link is provided by BASE$[s]$ in LEAF$[s] = 1$. For example, the link to TAIL$[9..10]$ = `b#` is provided by BASE$[5] = 9$ in LEAF$[5] = 1$. Note that CHILD and SIB are introduced in Section 2.3. In Figure 1, the empty elements and spaces are shaded. The load factors are $\alpha = 10/13 = 0.77$ and $\beta = 13/17 = 0.76$. The empty elements are chained using the BASE and CHECK values, as explained in Section 2.3.

The example dictionary embeds associated values in a fixed-length space on TAIL after the terminator, e.g., TAIL$[11] = 3$. If a key is registered without TAIL, the associated value is embedded in the corresponding leaf BASE element, e.g., BASE$[4] = 0$. Embedding is the most practical dictionary implementation using dynamic DA tries. This is because node addresses change frequently during updates. On the other hand, static tries can implement mapping keys to unique identifiers using RANK/SELECT operations [12].

Basic operations. Here, we define three basic dictionary operations: SEARCH, INSERT, and DELETE. In addition, we describe implementation examples for the dictionary shown in Figure 1.

SEARCH(*key*) returns the associated value if *key* is registered. This is implemented by traversing root-to-leaf nodes and by checking TAIL. Suppose that SEARCH(`abcabc#`) = 2 is performed. First, child 1 of root 0 is found as BASE$[0] \oplus \texttt{a} = 0 \oplus 1 = 1$ and CHECK$[1] = 0$. Similarly, nodes 6 and 10 are found as follows: BASE$[1] \oplus \texttt{b} = 4 \oplus 2 = 6$, CHECK$[6] = 1$, BASE$[6] \oplus \texttt{c} = 9 \oplus 3 = 10$, and CHECK$[10] = 6$. Equation LEAF$[10] = 1$ means that the other characters can be found in TAIL starting from BASE$[10] = 4$. From TAIL$[4..7]$ = `abc#`, the key is registered and the value TAIL$[8] = 2$ is returned.



	0	1	2	3	4	5	6	7	8	9	10	11	12
BASE	0	4	13	9	0	9	9	15	0	11	4	3	4
CHECK	0	0	0	11	1	12	1	12	6	3	6	9	2
EMPTY	0	0	0	1	0	0	0	0	0	1	0	1	0
LEAF	0	0	0	0	1	1	0	1	1	0	1	0	0
CHILD	a	#	a	?	?	?	a	?	?	?	?	?	a
SIB	?	b	a	?	b	c	#	a	c	?	a	?	a

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TAIL	a	#	1	?	a	b	c	#	2	b	#	3	?	?	?	#	4

Figure 1: MP-trie dictionary and the corresponding DA

INSERT(key, val) registers key and associates val to the key. This is implemented by defining new nodes for the key and adding a new suffix to TAIL. If collisions of elements occur, they are solved by relocating the elements. Suppose that INSERT(abccb#, 5) is performed. A part of the resulting dictionary is shown in Figure 2. First, new branches labeled a and c from node 10 are added. Then, a new leaf node labeled a is defined and linked to TAIL[5]. Consequently, TAIL[4] becomes empty. At the same time, another new leaf node labeled c is also defined. The suffix b# and value 5 are appended to the end of TAIL, and the node forms a link to TAIL[17].

DELETE(key) removes key from the dictionary. This is implemented by removing the nodes that correspond to the key. Suppose that DELETE(abcabc#) is performed. A part of the resulting dictionary is shown in Figure 3. First, node 10 corresponding to the key is removed by emptying the element. At the same time, node 6 becomes a new leaf for key abaa# as the structure of the minimal-prefix keys is changed. The new suffix aa# and value 1 are appended to the end of TAIL by transferring the edge label a and TAIL[0..2]. This process is completed when BASE[6] maintains the link to TAIL[17] and element 8 becomes empty. As a result, empty elements 8 and 10, and empty spaces TAIL[0..2] and TAIL[4..8] are formed.

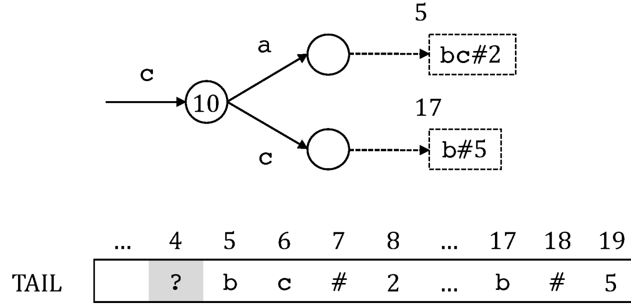


Figure 2: Result of INSERT(abccb#, 5) in the dictionary shown in Figure 1

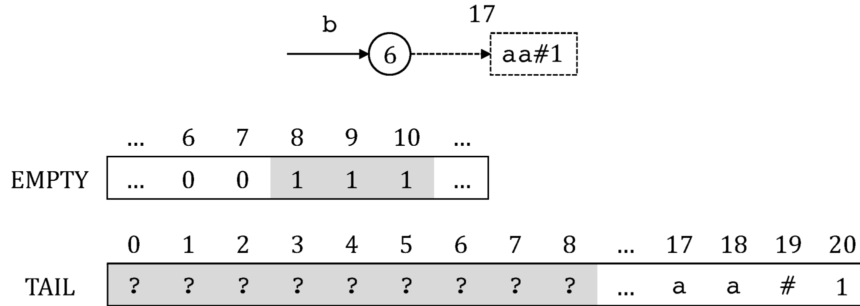


Figure 3: Result of DELETE(abcabc#) in the dictionary shown in Figure 1

2.3 Fast update methods

Two bottlenecks are typically observed when updating DA structures. One is the time taken to scan the empty elements, and the other is the time taken for enumeration of the edge labels. Below we present well-used methods to improve these bottlenecks.

Empty-link method. When new nodes are inserted, the empty elements must be searched to locate the nodes. The original DA [6] performs this search by scanning BASE and CHECK elements linearly in $O(N)$ time; however, this time turns into a critical problem for a large dictionary. Therefore, general implementations use the *empty-link method (ELM)* [15, 26]. ELM builds a doubly circular linked list of empty elements called an *empty list.*

Let $R = \{r_1, r_2, \ldots, r_m\}$. The ELM builds the empty list using empty BASE and CHECK elements as follows:

$$\textsc{base}[r_i] = \begin{cases} r_{i+1} & (1 \leq i < m) \\ r_1 & (i = m) \end{cases} \text{ and } \textsc{check}[r_i] = \begin{cases} r_{i-1} & (1 < i \leq m) \\ r_m & (i = 1) \end{cases}.$$

In other words, the successor and predecessor of r_i are obtained from BASE$[r_i]$ and CHECK$[r_i]$, respectively. The ELM can scan R in $O(m)$ time when the first empty element is maintained as the list head. For example, in Figure 1, empty elements 3, 9, and 11 are scanned as BASE$[3] = 9$, BASE$[9] = 11$, and BASE$[11] = 3$. The ELM utilizes empty elements and thus does not have any disadvantages in terms of space efficiency.

When updating an empty list, the doubly circular linkage can support the adding and

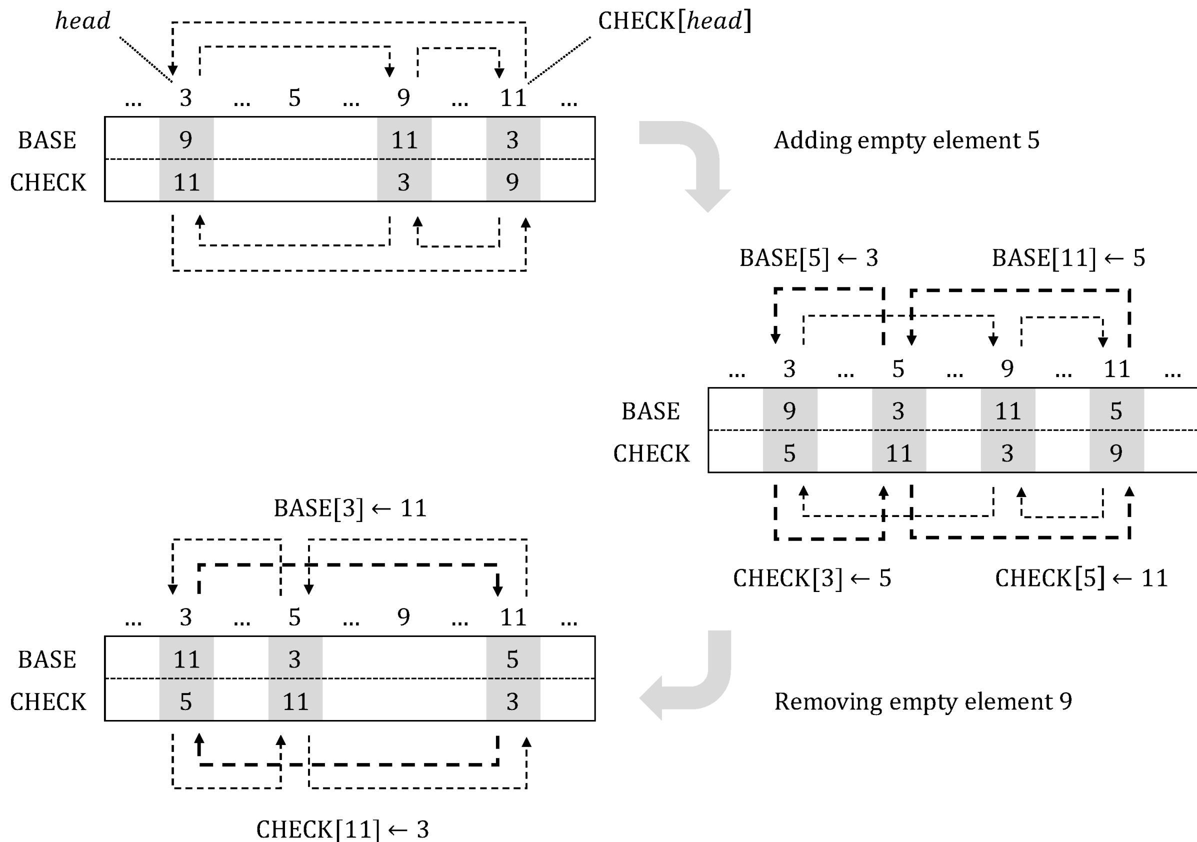


Figure 4: Examples of updates with the ELM

removing of an empty element in constant time as follows. Adding a new empty element is implemented by inserting the element between the first and last elements of the list. Removing an empty element is achieved by re-chaining the previous and next elements. Figure 4 shows an example of updating a DA where $R = \{3, 9, 11\}$. In this example, element 3 is the first empty element and is kept in the variable *head*. The last element, or element 11, is given by $\textsc{check}[head]$. When a new empty element 5 is added, it is inserted between the first and last elements as $\textsc{base}[5] \leftarrow 3$, $\textsc{check}[3] \leftarrow 5$, $\textsc{base}[11] \leftarrow 5$, and $\textsc{check}[5] \leftarrow 11$. Note that the resulting empty elements are scanned in the order of 3, 9, 11, and 5. In other words, the order of $r_1, r_2, \ldots, r_m$ does not correspond to the address order of BASE and CHECK. When the empty element 9 is removed from the DA, the previous element $\textsc{check}[9] = 3$ and the next element $\textsc{base}[9] = 11$ are re-chained as $\textsc{base}[3] \leftarrow 11$ and $\textsc{check}[11] \leftarrow 3$. The updates are performed in constant time.

Node-link method. Let $\textsc{Edges}(s)$ be an operation returning a set of edge labels from node s. For example, $\textsc{Edges}(1) = \{\texttt{\#}, \texttt{b}\}$ in Figure 1. A simple implementation performs $\textsc{Edges}(s)$ in $O(\sigma)$ time by checking the children from node s for all characters in Σ. The time required for this implementation is not significant as $\sigma \leq 256$ when byte characters are used. However, the time required can become a critical bottleneck since EDGES is often called during key updates.

This problem can be solved by the *node-link method (NLM)* [16]. NLM uses additional CHILD and SIB arrays such that $\textsc{child}[s]$ stores the edge label between node s and its

first child, and $\text{SIB}[s]$ stores the edge label between the next sibling of node s and its parent. The NLM can obtain the first child and the next sibling of node s in constant time by $\text{BASE}[s] \oplus \text{CHILD}[s]$ and $\text{BASE}[\text{CHECK}[s]] \oplus \text{SIB}[s]$, respectively. In other words, $E \leftarrow \text{EDGES}(s)$ is performed in $O(|E|)$ time. For example, Figure 1 shows CHILD and SIB for the trie. In this example, $\text{EDGES}(1) = \{\texttt{\#}, \texttt{b}\}$ is performed as follows. The first edge label # is given by $\text{CHILD}[1] = \texttt{\#}$; the first child 4 is calculated by $\text{BASE}[1] \oplus \text{CHILD}[1] = 4 \oplus \texttt{\#} = 4$; the second edge label b is given by $\text{SIB}[4] = \texttt{b}$; and the second child 6 is calculated by $\text{BASE}[1] \oplus \text{SIB}[4] = 4 \oplus \texttt{b} = 6$. It is evident that node 6 is the last child from $\text{SIB}[6] = \texttt{\#} = \text{CHILD}[1]$. However, the NLM involves a trade-off between update time and space efficiency on account of its use of $2N\lceil \log \sigma \rceil$ additional bits.

2.4 Improvement of space efficiency

Although repeating INSERT and DELETE reduces load factors α and β, some additional improvement methods have been proposed for each load factor.

BASE and CHECK. Morita et al. [24] proposed a rearrangement method to improve load factor α by packing the arrays. This method eliminates empty elements by relocating the rearmost element and its siblings known as *compression elements* to the empty elements. While rearranging the arrays with each DELETE operation can help to maintain a high load factor α, some empty elements always remain. This is because elimination is not possible if there are fewer empty elements than compression elements. Oono et al. [25] improved Morita's method by focusing on *single nodes* with no siblings. Oono's method applies empty elements and single elements to the relocation addresses. This method can maintain a high load factor α in a normal trie that includes many single nodes; however, it does not work efficiently in an MP-trie as the many single nodes in a normal trie are replaced into strings. Therefore, Yata et al. [26] presented an adaptive method that uses elements with fewer siblings than the rearmost element as well as empty and single elements. Available literature [26] shows that this adaptive method can maintain $\alpha \simeq 1.0$ in MP-tries from experiments using English and Japanese datasets.

Here, we describe a Yata's adaptive procedure, namely PACK. The pseudocode for PACK is shown in Algorithm 1. PACK searches the relocation addresses of the compression elements using EXCHECK, which is shown in Algorithm 2. EXCHECK then returns the BASE value indicating the addresses. Note that EXCHECK in Algorithm 2 uses ELM. Other operations used in PACK and EXCHECK are described as follows.

- $\text{SHELTER}(s, base, E)$ locates elements in $\{u \in U \mid \text{EMPTY}[u] = 0\}$ and their siblings to empty elements in $R - \{u \in U \mid \text{EMPTY}[u] = 1\}$, where $U = \{base \oplus c \mid c \in E\}$. If element s is located, the located address is returned; otherwise, s is returned.

- $\text{MOVE}(s, base, E)$ locates elements $\text{BASE}[s] \oplus c$ to empty elements $base \oplus c$ for all characters $c \in E$.

- $\text{ISTARGET}(base, E)$ returns TRUE if addresses $s \leftarrow base \oplus c$ satisfy the conditions for all characters $c \in E$: $0 < s < N$, $\text{EMPTY}[s] = 1$, and $|\text{EDGES}(\text{CHECK}[s])| < |E|$; otherwise, it returns FALSE.

Algorithm 1 PACK

```
 1: while R ≠ ∅ do
 2:     s ← CHECK[N − 1]
 3:     E ← EDGES(s)
 4:     base ← EXCHECK(E)
 5:     if base = −1 then
 6:         break
 7:     end if
 8:     s ← SHELTER(s, base, E)
 9:     MOVE(s, base, E)
10: end while
```

Algorithm 2 EXCHECK(E) using ELM. E is a set of edge labels.

```
1: r ← head
2: repeat
3:     base ← r ⊕ E[0]                          ▷ E[0] is an arbitrary character in E
4:     if ISTARGET(base, E) = TRUE then
5:         return base
6:     end if
7:     r ← BASE[r]                               ▷ a next empty element
8: until r = head
9: return −1
```

Essentially, PACK improves the load factor α by repeating the following steps. EXCHECK obtains the relocation addresses of the compression elements; SHELTER solves collisions during relocation; and MOVE eliminates empty elements by relocating the compression elements. These steps are depicted in Figure 5. EXCHECK scans R using ELM and checks BASE values calculated from each empty element using ISTARGET. ISTARGET checks whether the received BASE value is appropriate to indicate the relocation addresses in the adaptive method. Since the algorithm is complex, please refer to the original literature [26] for more details.

TAIL. Dorji et al. [23] proposed two methods to improve the load factor β. The first method embeds short suffixes into leaf BASE elements rather than storing them in TAIL.

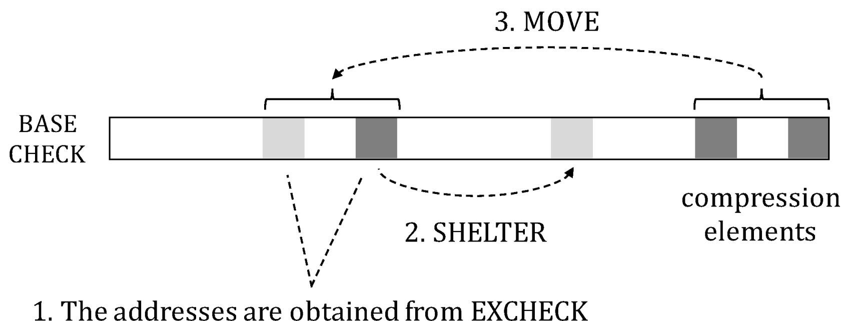


Figure 5: Steps of PACK

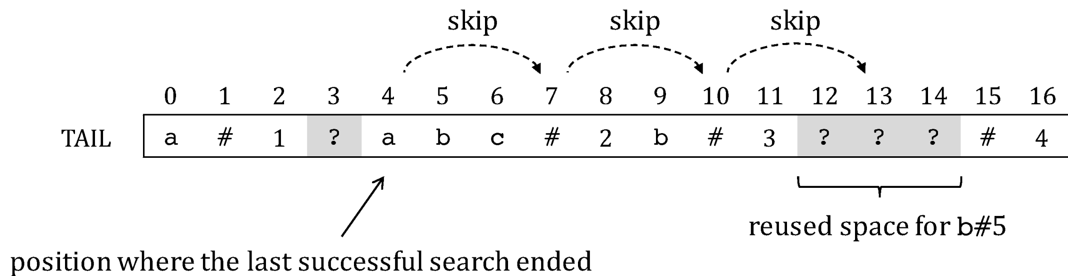


Figure 6: Example of Dorji's second method demonstrating the insertion of suffix `b#` and value 5

In addition to saving TAIL space, this method can reduce the number of TAIL updates. The second method allows reuse of the empty spaces by searching TAIL sequentially. However, simple sequential search requires significant amounts of time; therefore, the method combines three techniques.

1. A suffix longer than the threshold length is appended to the end of TAIL in the same manner as the conventional update.
2. The search begins from the position where the last successful search ended.
3. The search proceeds by skipping every $l-1$ elements until an empty space is found. Here, l denotes the suffix length.

Figure 6 shows an example of the second method. This method marks any empty space with a *white space*; however, such marking is unsuitable for dictionary implementation.

3 Practical rearrangement methods

While the existing methods described in Section 2.4 can maintain high load factors α and β, these methods present practical problems related to time and functionality.

- The combination of DELETE and PACK requires significantly more time compared to simple DELETE since the algorithm is complex and involves many operations. In addition, empty elements are reused during INSERT. Therefore, maintaining the high load factor α by sacrificing DELETE time does not have many advantages.
- Dorji's first method cannot implement trie dictionaries, such as that shown in Figure 1, as the associated values cannot be embedded. In addition, in Dorji's second method, empty marking cannot be used for dictionary implementation since any character can be used to represent associated values. In other words, the third technique cannot be used because the empty spaces are not directly identified. Previous results [23] show that the method presents problems related to time requirements for large datasets. If the third technique is excluded, the problem becomes much more critical.

Algorithm 3 EXCHECK(E) using BLM. E is a set of edge labels.

```
 1: b ← head                                        ▷ head block address
 2: repeat
 3:     r ← HEAD[b]
 4:     repeat
 5:         base ← r ⊕ E[0]            ▷ E[0] is an arbitrary character in E
 6:         if IsTarget(base, E) = TRUE then
 7:             return base
 8:         end if
 9:         r ← BASE[r]                             ▷ next empty element
10:     until b = HEAD[b]
11:     b ← NEXT[b]                                  ▷ next block
12: until b = head
13: return −1
```

To address these problems, the simplest solution is to rearrange the DA when necessary, e.g., when a dictionary is written to a file and the load factors exceed the predefined lower limits. This section proposes some efficient rearrangement methods. Sections 3.1 and 3.2 describe rearranging BASE and CHECK. TAIL is simply rearranged by transferring suffixes for all leaves to a new TAIL.

3.1 Cache-friendly implementation of the ELM

While the ELM supports scanning R in $O(m)$ time, the scan order is random in BASE and CHECK, as shown in Figure 4. Therefore, cache misses can occur frequently in EXCHECK when there are many empty elements. We solve this problem using the *block-link method (BLM)*, which implements empty-linkage in the following two steps: the first step partitions BASE and CHECK into blocks of length L and the second step builds empty-linkages for each block. More precisely, the method builds two types of empty lists for blocks, which include any empty elements, as well as empty elements in each block. We refer to the former as the *block list*. The latter comprises small empty lists obtained using the ELM for each block.

Let R_b be a set of addresses of the empty elements in block b, that is, $R = \bigcup_{b<\lceil N/L \rceil} R_b$. The block list consists of three additional arrays: HEAD[b] keeps an arbitrary address in R_b as the first empty element; NEXT[b] keeps the next linking block address of block b; and PREV[b] keeps the previous linking block address of block b. The block list is also updated in constant time in the same manner as the ELM. The BLM scans R by visiting each block b and by scanning R_b. Since the addresses in R_b are in a constant area of memory, the BLM can improve the cache efficiency of the ELM. Figure 7 shows linkage examples using ELM and BLM with $L = 4$. While ELM scans R in the order of 4, 9, 14, 8, 6, and 11, BLM can scan R in the order of 4, 6, 9, 11, 8, and 14. Algorithm 3 shows EXCHECK using BLM. In Algorithm 3, the first loop visits each block using NEXT, and the second loop scans empty elements in the block using BASE. The algorithm of the second loop is the same as in the ELM version; however, the scan area remains constant.

For the block length, $L = 2^{\lceil \log \sigma \rceil}$ is efficient for the following reasons. For $base \leftarrow r \oplus E[0]$ in EXCHECK, the $base$ is the result of a bitwise XOR operation on the lowest

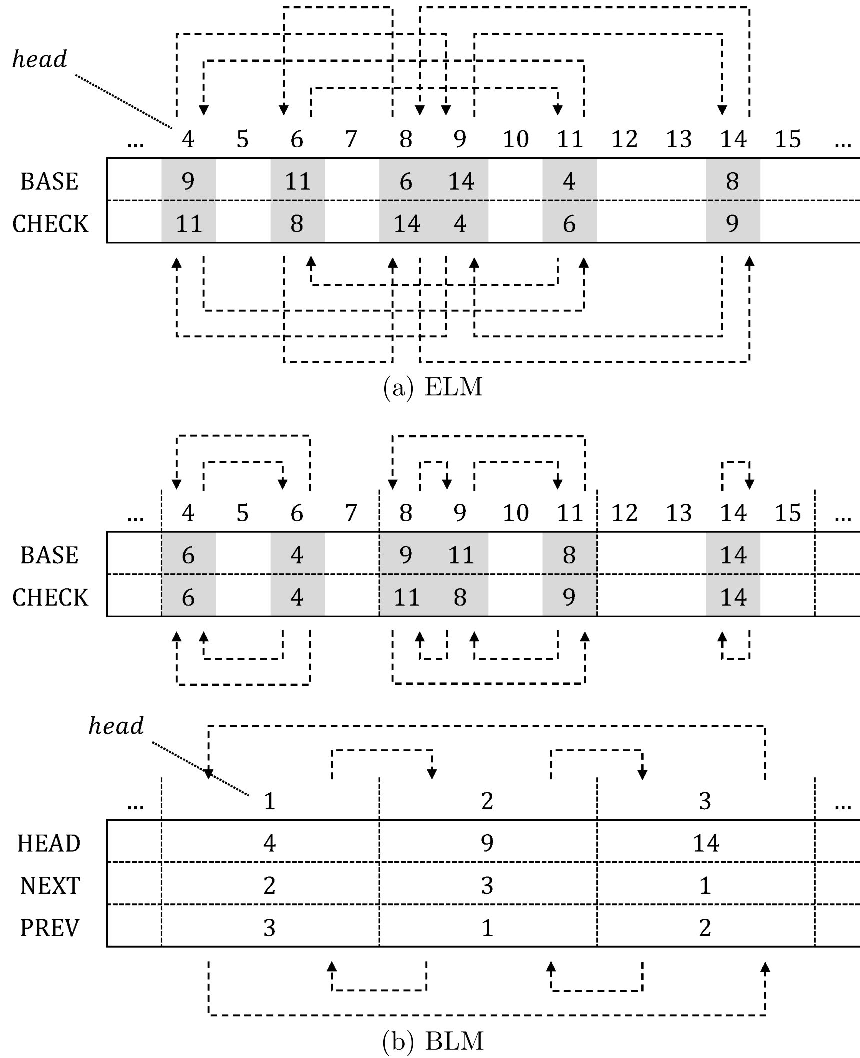


Figure 7: Examples of the ELM and BLM

$\lceil \log \sigma \rceil$ bits of r. In other words, for $L = 2^{\lceil \log \sigma \rceil}$, both r and $base$ are integers indicating the same block address since $\lfloor r/L \rfloor = \lfloor base/L \rfloor$. IsTarget searches relocation addresses using the $base$; thus, addresses $s \leftarrow base \oplus c$ for all characters $c \in E$ are also integers that indicate the same block address. Therefore, when calling IsTarget for arbitrary r_i, there is a strong probability that elements in the block $\lfloor r_i/L \rfloor$ will be present in cache memory. If elements $r_{i+1}, r_{i+2}, \ldots$ are in the same block, cache efficiency can be improved. In the experiments discussed in Section 4, our implementation uses $L = 2^{\lceil \log 256 \rceil} = 256$ for all datasets since $\sigma \leq 256$ is always satisfied when using single-byte characters, while σ is frequently not pre-given in practical dynamic applications. For $L = 256$, the additional HEAD, NEXT, and PREV arrays use only $3 \cdot 32/256 = 0.375$ bits for each BASE or CHECK element when assuming 32-bit integers to indicate addresses. This space is negligibly small compared to the total space of BASE and CHECK using 64 bits for each element.

Related studies. For block linkage, the previous literature [16] has proposed a similar method[1]. This method implements recommendation of appropriate search areas for query keys by classifying blocks. This reduces the average number of visiting empty elements. Other related studies [17, 18] have also proposed methods to reduce the number of visits using different approaches. However, applying such methods to Pack is difficult as they are designed for simple key insertion. Therefore, our experiments did not implement such methods.

3.2 Static reconstruction

Although Pack improves the load factor α by eliminating empty elements, the algorithm is complex and includes many operations. On the other hand, rebuilding the DA dictionary from scratch is a very simple approach. Generally, this is called *static reconstruction* since the DA dictionary is built on the condition that all registered keys are pre-given.

Algorithm 4 shows the static reconstruction pseudocode. In Rebuild, a new dictionary is built from the original dictionary. The new dictionary registers the same keys and associated values. The algorithm is described as follows. Rebuild traverses nodes in the original trie and re-determines the BASE and CHECK values for each node (lines 8–15). To determine a new BASE value, Rebuild uses XCheck(E), which returns an integer *base* such that EMPTY[$base \oplus c$] $= 1$ for all characters $c \in E$. XCheck determines the BASE value by scanning R in the same manner as EXCheck; however, Rebuild does not require to solve node collisions using Shelter. When a leaf is reached, the suffix and associated value are transferred to the new TAIL array (lines 17–23).

In the algorithm, node addresses are redefined in the depth-first order using a stack. Although a queue can be substituted, node addresses are redefined in the breadth-first order. Generally, such node arrangement causes frequent cache misses in node-to-node traversal near leaves. Thus, most public implementations of a static DA trie define node addresses in the depth-first order. While recursive processing can also define node addresses in the depth-first order, the algorithm considers stack overflow for large datasets, especially in the concurrent rearrangement introduced in Section 3.3.

An advantage of Rebuild is that a pair of parent and child nodes can be proximally positioned as it is not necessary to relocate elements to solve collisions. In other words, Rebuild can improve the cache efficiency of node-to-node traversal. In fact, the experimental results in [16] show that static DA dictionaries enable faster retrieval than DA dictionaries built by sequentially inserting random keys. In addition to Search, the cache improvement contributes to improved Insert and Delete as they also retrieve the received key. On the other hand, Rebuild requires a larger working space than Pack as it uses large temporal structures, such as BASE$'$, CHECK$'$, and ST. Section 4.1 provides estimations of each working space.

3.3 Concurrent rearrangement

We can shorten rearrangement time by concurrent processing. The proposed method called the *trie division method (TDM)* enables concurrent rearrangement by dividing a

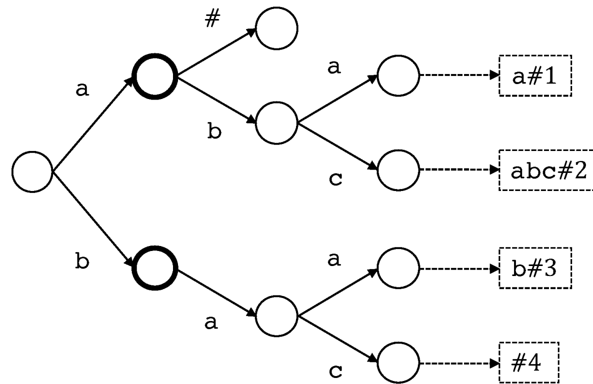
[1] Although the literature [16] is written in Japanese, Yoshinaga and Kitsuregawa [21] have introduced the proposed methods in English.

Algorithm 4 REBUILD

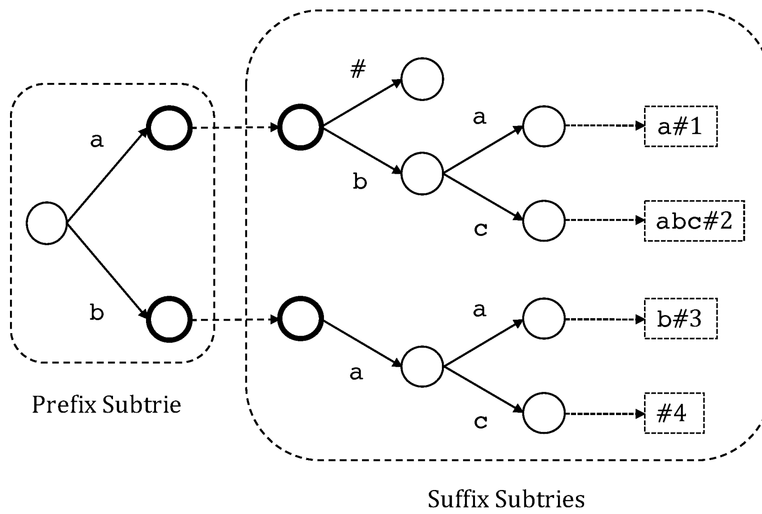
```
1: Create empty arrays BASE′, CHECK′, EMPTY′, LEAF′ and TAIL′        ▷ They are temporal
2: Create an empty stack ST for storing node address pairs
3: Push root pair (0, 0) to ST
4: EMPTY′[0] ← 0
5: while ST is not empty do
6:     Pop node address pair (s, s′) from ST
7:     if LEAF[s] = 0 then
8:         E ← EDGES(s) for the original dictionary
9:         BASE′[s′] ← XCHECK(E) for the new dictionary
10:        for all c ∈ E do
11:            t′ ← BASE′[s′] ⊕ c
12:            CHECK[t′] ← s′
13:            EMPTY′[t′] ← 0
14:            Push child pair (BASE[s] ⊕ c, t′) to ST
15:        end for
16:    else
17:        LEAF′[s′] ← 1
18:        if BASE[CHECK[s]] ⊕ s = # then        ▷ Leaf s has an associated value
19:            BASE′[s′] ← BASE[s]
20:        else
21:            Transfer the suffix and associated value from TAIL[BASE[s]] at the end of TAIL′
22:            BASE′[s′] ← the start address of the suffix on TAIL′
23:        end if
24:    end if
25: end while
26: Swap the temporal arrays BASE′, CHECK′, . . . and the original ones BASE, CHECK, . . .
```

trie into a *prefix subtrie* and *suffix subtries*, as shown in Figure 8. In the figure, the trie is divided based on the highlighted nodes, which are referred to as *division nodes*. The prefix subtrie has a common root for all registered keys. The leaves corresponding to the division nodes maintain links to the suffix subtries. The prefix subtrie is built as a normal trie because the structure with leaves having links overlaps the MP-trie. In the suffix subtries, each root corresponds to each division node. The suffix subtries are built as the MP-trie.

The TDM sets a rule to determine the division nodes based on the features of the registered keys. The important consideration for the rule is that many suffix subtries do not occur. For keys with no special feature in the prefixes, it is only necessary to determine nodes from the root as division nodes. In other words, a key is separated at the first character, as shown in Figure 8b. As the maximum number of suffix subtries is $\sigma = 256$ in practice, concurrent processing will work efficiently. For keys with any feature in the prefixes, such as URLs, particular prefixes are should be preregistered. Then, nodes from the predefined nodes are defined as division nodes. Figure 9 shows an example of a prefix subtrie preregistering `http://` and `https://`. If keys `http://a...`, `http://b...`, and `https://c...` are registered, the nodes are defined from edges with labels `a`, `b`, and `c` as division nodes. The TDM is similar to the technique used in the *DA language models* [28, 29] for static tries with a large alphabet.



(a) Without the TDM



(b) With the TDM

Figure 8: Examples of the TDM for the MP-trie in Figure 1

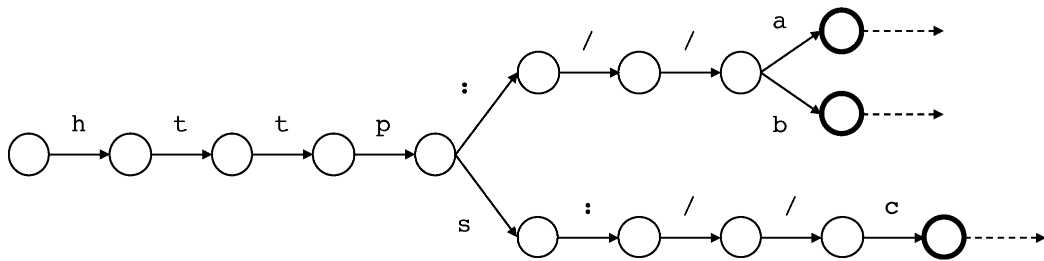


Figure 9: Example of a prefix subtrie preregistering `http://` and `https://`

Since the TDM can implement dictionary operations by traversing nodes in the order of prefix and suffix subtries, no major changes to the algorithms are required; however, there are two important minor changes for INSERT and DELETE in the prefix subtrie. For INSERT, if the key registration is performed within the prefix subtrie, the value is embedded in a leaf of the prefix subtrie. For example, in Figure 9, when INSERT(ht#, 1) is performed, we append a leaf with the label # from the node reached by ht, and we embed the value 1 to the BASE value. For DELETE, if a suffix subtrie becomes empty, only the corresponding division node is removed. Predefined nodes are not removed for future insertions. Here, suppose that DELETE(https://c...) is performed in Figure 9. As the suffix subtrie becomes empty, only the division node with the label c is removed and the nodes predefined with labels s:// are not removed.

4 Experimental evaluation

This section analyzes the practical performance of various combinations of rearrangement methods using real-world datasets. In these experiments, we assessed the rearrangement times and working spaces for some configuration dictionaries. Furthermore, we measured the runtimes of SEARCH after rearrangement to evaluate the cache improvement from REBUILD. The source code is available at `https://github.com/kamp78/ddd`.

4.1 Settings

We carried out the experiments using Quad-core Intel Core i7 4.0 GHz, with 16 GB RAM (L2 cache: 256 KB; L3 cache: 8 MB). The methods were implemented in C++ and compiled using Apple LLVM version 8 (clang-8) with optimization -O3. The runtimes were measured using `std::chrono::duration_cast`. To measure the working spaces, we used the `/usr/bin/time` command and referred to the *maximum resident set size*. We used `std::thread` for the TDM with eight threads.

Datasets. The dictionaries were built from the following real-world datasets.

- GEONAMES: Geographic names on the *asciiname* column from the GeoNames dump (`http://download.geonames.org/export/dump/allCountries.zip`).
- ENWIKI: All page titles from the English Wikipedia of February 2015 (`https://dumps.wikimedia.org/enwiki/`).
- NWC: Japanese word *n*grams in the Nihongo Web Corpus 2010 (`http://dist.s-yata.jp/corpus/nwc2010/ngrams/word/over999/filelist`).
- INDOCHINA: URLs of a 2004 crawl of the country domains of Indochina performed by UbiCrawler [30] (`http://data.law.di.unimi.it/webdata/indochina-2004/indochina-2004.urls.gz`).

Table 1 summarizes each dataset. Here, *Size* is the raw size in MiB, *Keys* is the number of different string keys, *Ave. length* is the average number of characters per string including a terminator, *Chars* is the number of different characters in the dataset,

Table 1: Datasets

	Size	Keys	Ave. length	Chars	Nodes	Singles	Subtries
GEONAMES	101.2	6,784,722	15.6	96	12,216,182	19.1	81
ENWIKI	227.2	11,519,354	20.7	199	27,308,602	36.6	108
NWC	439.4	20,722,756	22.2	180	60,554,010	42.5	115
INDOCHINA	612.9	7,414,866	86.7	98	22,572,605	55.2	33

Table 2: Possible combinations of rearrangement methods

	ELM	ELM + NLM	BLM	BLM + NLM
PACK	PE	PEN	PB	PBN
REBUILD	RE	REN	RB	RBN
	ELM + TDM	ELM + NLM + TDM	BLM + TDM	BLM + NLM + TDM
PACK	PET	PENT	PBT	PBNT
REBUILD	RET	RENT	RBT	RBNT

Nodes is the number of nodes in the MP-trie, *Singles* is the percentage of single nodes, and *Subtries* is the number of subtries with the TDM. All datasets were encoded in UTF-8.

Data structures. For rearrangement methods, we evaluated all possible combinations shown in Table 2 (16 patterns in total). This section refers to each combination as the name composed of the initial letters. PE and PEN are the conventional methods, and the others are the proposed methods.

For the TDM, since all keys in INDOCHINA start with the prefix `http://`, we preregistered the prefix. In the other datasets, we separated a key at the first character, as shown in Figure 8b. Table 3 shows the top 10 results related to the frequency of appearance of characters for each divided point. For the NWC and INDOCHINA datasets, the frequency of a particular character is very large. The ASCII code 227 is the first byte character generally used to represent Japanese letters in UTF-8. The ASCII code 119 is the letter `w` of `http://www...`. Thus, a particular suffix subtrie becomes very large. Note that such imbalance is also an evaluation item.

We prepared ten patterns of load factors for each dictionary by randomly inserting all keys and by randomly deleting the keys. Figure 10 shows the load factors α and β of PE for each dataset before rearrangement. The dictionaries become increasingly sparse. In addition, Figure 11 shows that the TAIL length also becomes increasingly large, similar to the example shown in Figure 3. The length of BASE and CHECK, or N, was nearly unchanged because the arrays can shrink only when the rearmost element is removed.

Implementation details. To measure the rearrangement performance precisely, it is important that no extra memory allocation is made for expanding arrays, especially for the working space. For TAIL, both PACK and REBUILD simply transfer non-empty spaces to a new array. Given an original TAIL whose length is M and load factor is β, we can estimate the new TAIL length as $M \cdot \beta$. Therefore, we only require initial memory

Table 3: Top 10 results in frequency of appearance of characters for each divided point

	GEONAMES		ENWIKI		NWC		INDOCHINA	
Rank	ASCII	%	ASCII	%	ASCII	%	ASCII	%
1	83	9.7	83	8.7	**227**	**51.6**	**119**	**62.9**
2	66	8.3	67	7.2	229	10.6	115	4.0
3	75	7.2	65	6.9	230	8.1	116	3.1
4	77	6.9	77	6.6	60	7.4	99	2.9
5	67	6.8	84	6.0	231	5.1	109	2.6
6	76	5.4	76	5.6	232	4.6	114	2.4
7	80	5.4	66	5.5	228	4.3	105	2.2
8	84	4.8	80	5.2	233	3.4	100	2.0
9	65	4.6	68	4.1	40	0.7	112	2.0
10	72	4.4	82	3.9	226	0.5	97	1.8

allocation.

On the other hand, we cannot accurately estimate the number of elements of BASE and CHECK after rearrangement since α depends on the trie configuration and the order of the defining nodes. PACK does not require this consideration because the node rearrangement is performed within the original arrays, that is, within N elements. However, REBUILD must determine the initial memory allocation size of the temporal structures. In the experiments, we reserved an additional 1024 elements for unknown empty elements as α was always improved to greater than 0.99 in the preliminary experiments. In other words, we initially reserved $N \cdot \alpha + 1024$ elements for BASE$'$ and CHECK$'$. Here, $N \cdot \alpha$ denotes the number of nodes ($= n$). Although the number is heuristic, extra memory allocation was never required in these experiments. For stack ST, we initially reserved sufficient $N \cdot \alpha$ units since this is necessary to traverse nodes.

We present the estimations of each working space before the final results are shown in Section 4.2. PACK rearranges nodes in $O(N)$ space. REBUILD rearranges nodes in $O(N(1 + \alpha))$ space because the original node components use $O(N)$ space and the new node components use $O(N \cdot \alpha)$ space. For TAIL rearrangement, both of PACK and REBUILD use $O(M) + O(M \cdot \beta) = O(M(1 + \beta))$ space. The above is summarized as follows. PACK uses $O(N + M(1 + \beta))$ space and REBUILD uses $O(N(1 + \alpha) + M(1 + \beta))$ space. Here, $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$. Although the working space of PACK is obviously smaller than that of REBUILD, the difference becomes small according to decreasing α.

4.2 Results

Figures 12, 13, and 14 show the experimental results for rearrangement times, working spaces, and search times, respectively. The y-axis in Figure 12 uses a logarithmic scale of 2. The search times were measured for all registered keys in each dictionary at random and were averaged over 10 runs. Figure 12 omits the results of combinations using REBUILD and BLM (i.e., RB, RBN, RBT, and RBNT) as there were no distinct differences compared to the NLM versions. Figure 13 omits the results of combinations using BLM (i.e., PB, PBN, PBT, PBNT, RB, RBN, RBT, and RBNT) since BLM requires only negligibly small space. Figure 14 omits the results of combinations using NLM and BLM (except

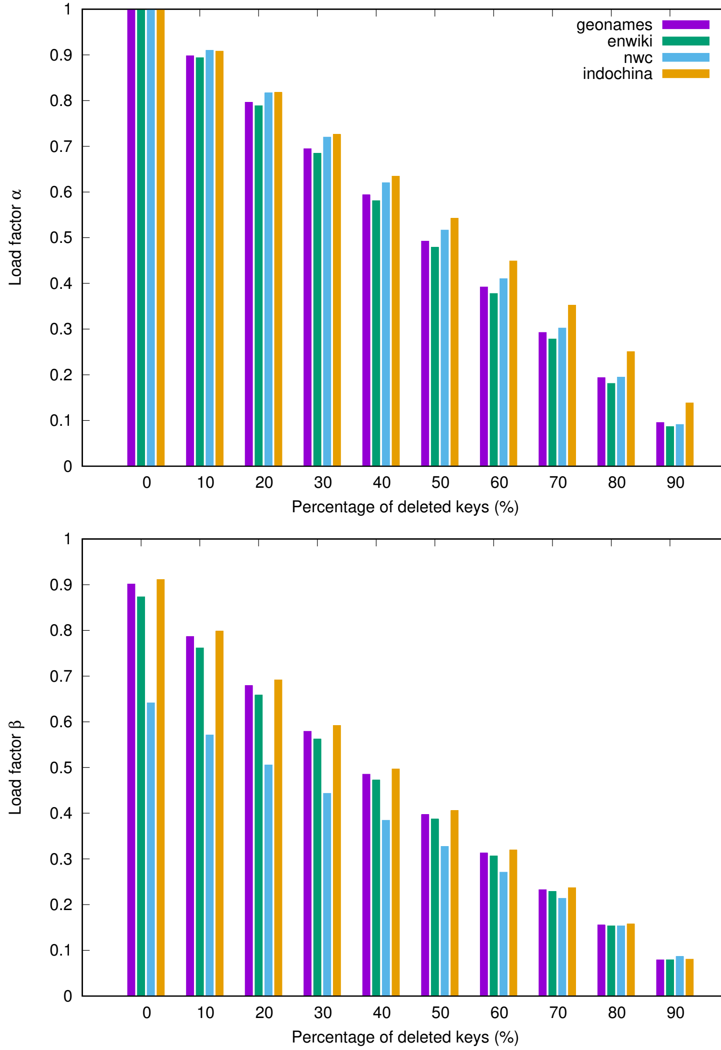


Figure 10: Load factors of PE for each dataset

PE, PET, RE, and RET) as these methods are not related to Search. In all cases, the load factors were improved to $\alpha \simeq 1.0$ and $\beta = 1.0$ after rearrangement.

Pack time observations. First, we focus on the results obtained with enwiki at a deletion rate of 50%. Here, the differences are obvious. For the conventional methods, PE uses 137 s and PEN does not improve the time since Edges time is not the main problem. On the other hand, PB and PBN significantly improve the time to 7 s and 4 s, respectively. Thus, the cache inefficiency of ELM and the usefulness of BLM are obvious. TDM also provides large improvements. PET, PENT, PBT, and PBNT are 35, 42, 6, and 9 times faster compared to each method without TDM, respectively. When comparing the best cases of the conventional and proposed methods, PBNT performs Pack in 0.5 s that is 260 times faster than PEN.

We also focus on the results obtained with indochina at a deletion rate of 50%. The improvement rates obtained by BLM are obviously smaller compared to the other datasets. When comparing ELM and BLM, PB is 1.5 times faster than PE, PBN is 2.1 times faster than PEN, PBT is 1.4 times faster than PET, and PBNT is 1.9 times faster than

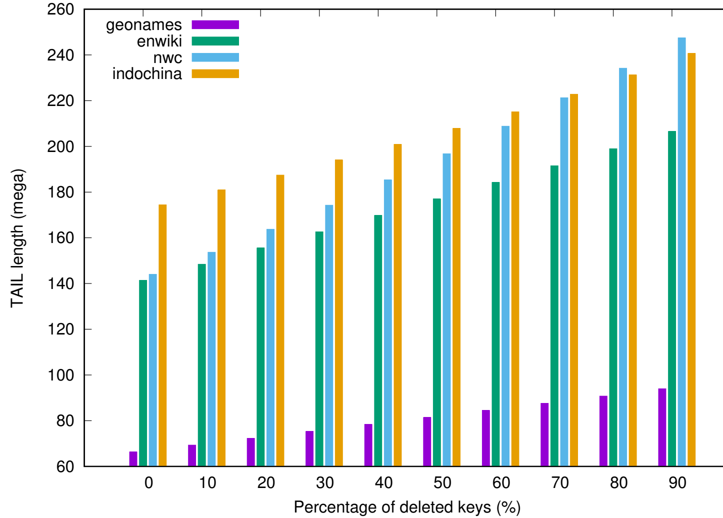


Figure 11: TAIL lengths of PE for each dataset

PENT. These results are the outcome of the rate of single nodes. When PACK relocates a single element where $|E| = 1$ in Algorithm 1, EXCHECK can directly select an empty element $r \leftarrow head$ in Algorithm 2 or $r \leftarrow \text{HEAD}[head]$ in Algorithm 3 as the relocation address. In other words, EXCHECK does not need to scan R as ISTARGET always returns TRUE. Therefore, the improvement rates obtained by BLM with INDOCHINA are smaller compared to those with ENWIKI. However, PACK is performed quickly.

Overall, PBNT always provides the best results. The largest improvement rates from PEN to PBNT with the GEONAMES, ENWIKI, NWC, and INDOCHINA datasets are 218, 260, 32, and 3 times, respectively. For NWC and INDOCHINA, the ratios are relatively small, since TDM did not work efficiently as seen from the imbalances in Table 3. As for the absolute runtimes, PBNT performs PACK in up to 0.9, 0.5, 3.0, and 1.3 s with the GEONAMES, ENWIKI, NWC, and INDOCHINA datasets, respectively. Therefore, the proposed PACK is useful under any conditions, while the conventional methods require more than a min under many conditions.

REBUILD time observations. In all cases, the simplest implementation, namely RE, is the slowest, and the most complex implementation, namely RENT, is the fastest. At a deletion rate of 0% with the GEONAMES, ENWIKI, NWC, and INDOCHINA datasets, the differences are 8.4, 9.2, 3.4, and 2.4 times, respectively. When comparing RET and REN, RET is faster than REN with GEONAMES and ENWIKI; however, the results are reversed with NWC and INDOCHINA. In other words, the advantage of NLM outperforms that of TDM due to the imbalances. Relative to the absolute runtimes, RENT performs REBUILD in up to 0.4, 0.8, 4.2, and 2.5 s with the GEONAMES, ENWIKI, NWC, and INDOCHINA datasets, respectively. Therefore, the enhanced REBUILD is very useful under any conditions, similar to PACK.

Comparison of PACK and REBUILD times. Here, we compare PBNT and RENT, which provide the best performance. With the smallest dataset, namely GEONAMES, RENT is often faster as the number of nodes is small. With ENWIKI and NWC, the results

are reversed at a deletion rate of approximately 35–40%. With INDOCHINA, the result is reversed at a deletion rate of approximately 50% as PACK is performed quickly. As shown in Figure 10, these reversals occur in approximately $0.5 \leq \alpha \leq 0.6$.

In conclusion, REBUILD is a good choice for small datasets, such as GEONAMES. For large datasets, PACK and REBUILD are more effective according to load factor α. It is difficult to accurately estimate the best threshold load factor; however, both will provide high performance in approximately $0.5 \leq \alpha \leq 0.6$.

Working space observations. All the results appear to be based on estimations. We have not presented the results obtained by NLM since the additional space of $2N\lceil\log\sigma\rceil$ bits is simply included. When comparing PE and RE, PE is 1.44, 1.45, 1.67, and 1.35 times smaller than RE at a deletion rate of 0% with the GEONAMES, ENWIKI, NWC, and INDOCHINA datasets, respectively. The ratio of NWC is relatively large because the number of nodes is large, as shown in Table 1. On the other hand, the ratios are close to 1.0 according to α. At a deletion rate of 90% with the GEONAMES, ENWIKI, NWC, and INDOCHINA datasets, the ratios are 1.04, 1.04, 1.05, and 1.05 times, respectively.

It is worth noting that TDM contributes to the reduction of working space. TDM rearranges small subtries in given threads; thus, the temporal spaces for each process are small. When comparing PE and PET with GEONAMES and ENWIKI, TDM reduces the working spaces by up to 1.26 and 1.23 times, respectively. When comparing RE and RET with GEONAMES and ENWIKI, TDM reduces the working spaces by up to 1.19 and 1.23 times, respectively. On the other hand, the reduction ratios obtained with NWC and INDOCHINA are relatively small due to the imbalances. When comparing PE and PET with NWC and INDOCHINA, TDM reduces the working spaces by up to 1.04 and 1.05 times, respectively. When comparing RE and RET with NWC and INDOCHINA, TDM reduces the working spaces by up to 1.08 and 1.19 times, respectively.

In conclusion, PACK is efficient for saving working space. TDM also contributes to the saving. For sparse dictionaries, REBUILD using TDM is a good choice.

SEARCH time observations. There are distinct differences between PACK and REBUILD since REBUILD can contribute to the cache improvement of node-to-node traversal. When comparing PE and RE, RE performs SEARCH up to 1.5, 1.8, 2.1, and 2.9 times faster than PE with the GEONAMES, ENWIKI, NWC, and INDOCHINA datasets, respectively. The result obtained with INDOCHINA is better because the number of node-to-node traversals is large. The obtained results are also related to update times since both INSERT and DELETE retrieve the received key. Therefore, it is possible to use REBUILD to improve all dictionary operations.

Synthetic evaluation. Finally, we present a synthetic evaluation of the proposed rearrangement methods. PACK and REBUILD should be used as the situation demands. If the focus is on saving the working space, PACK is a better choice. REBUILD is a better choice to improve the runtimes of dictionary operations. On the other hand, the working space of REBUILD is close to that of PACK when the dictionary is sparse. In terms of rearrangement time, REBUILD is more useful for sparse dictionaries for any datasets.

In all aspects, BLM and TDM provide many advantages without any disadvantage. Therefore, these methods can be applied as new default components for dynamic DA dic-

tionary implementations. Note that it is necessary to determine appropriate pre-registered prefixes when using TDM. The results show that it would be better to preregister the ASCII code 227 when storing Japanese keywords and add `http://www` to the preregistered prefixes when storing URLs.

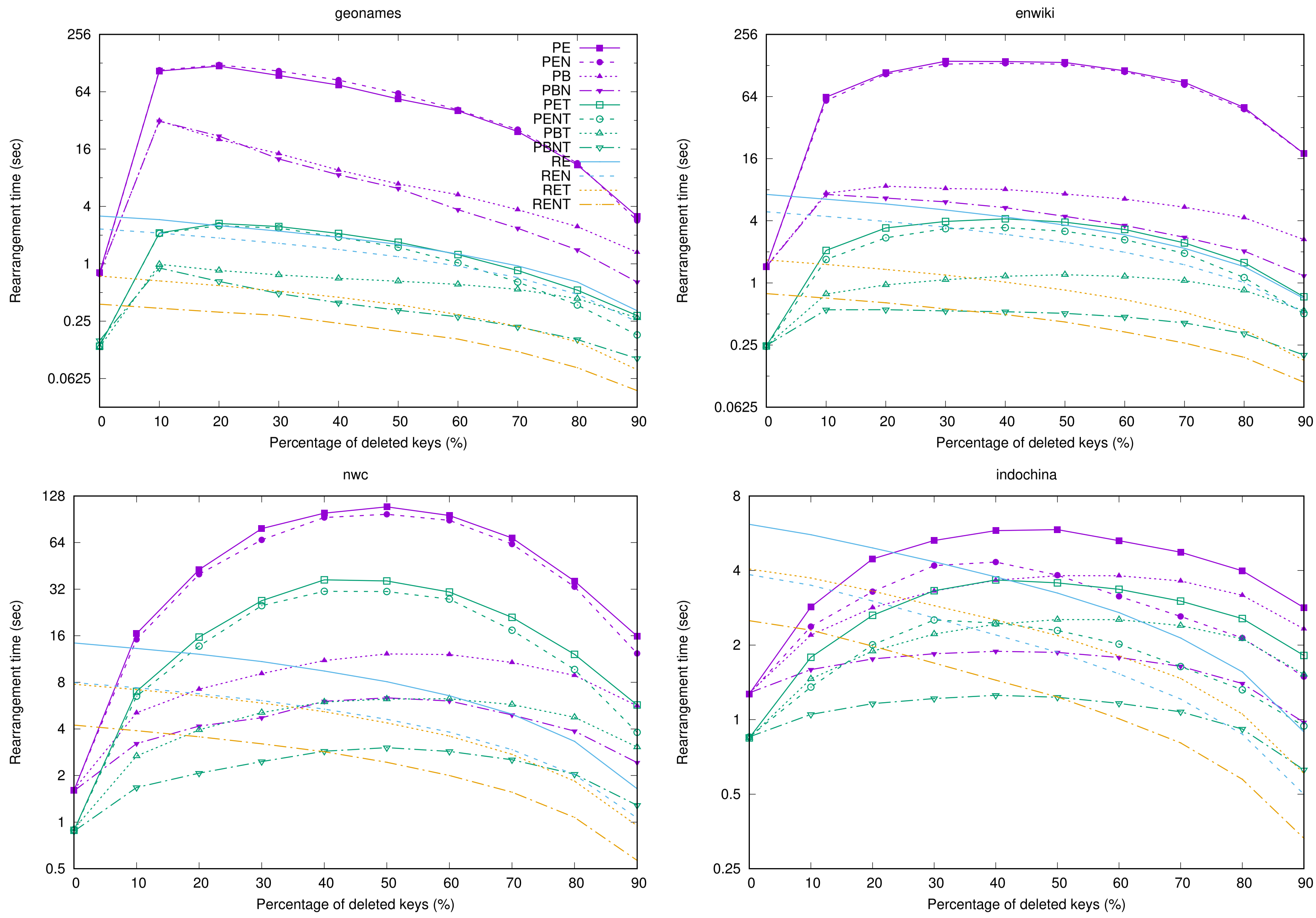


Figure 12: Experimental results of rearrangement time

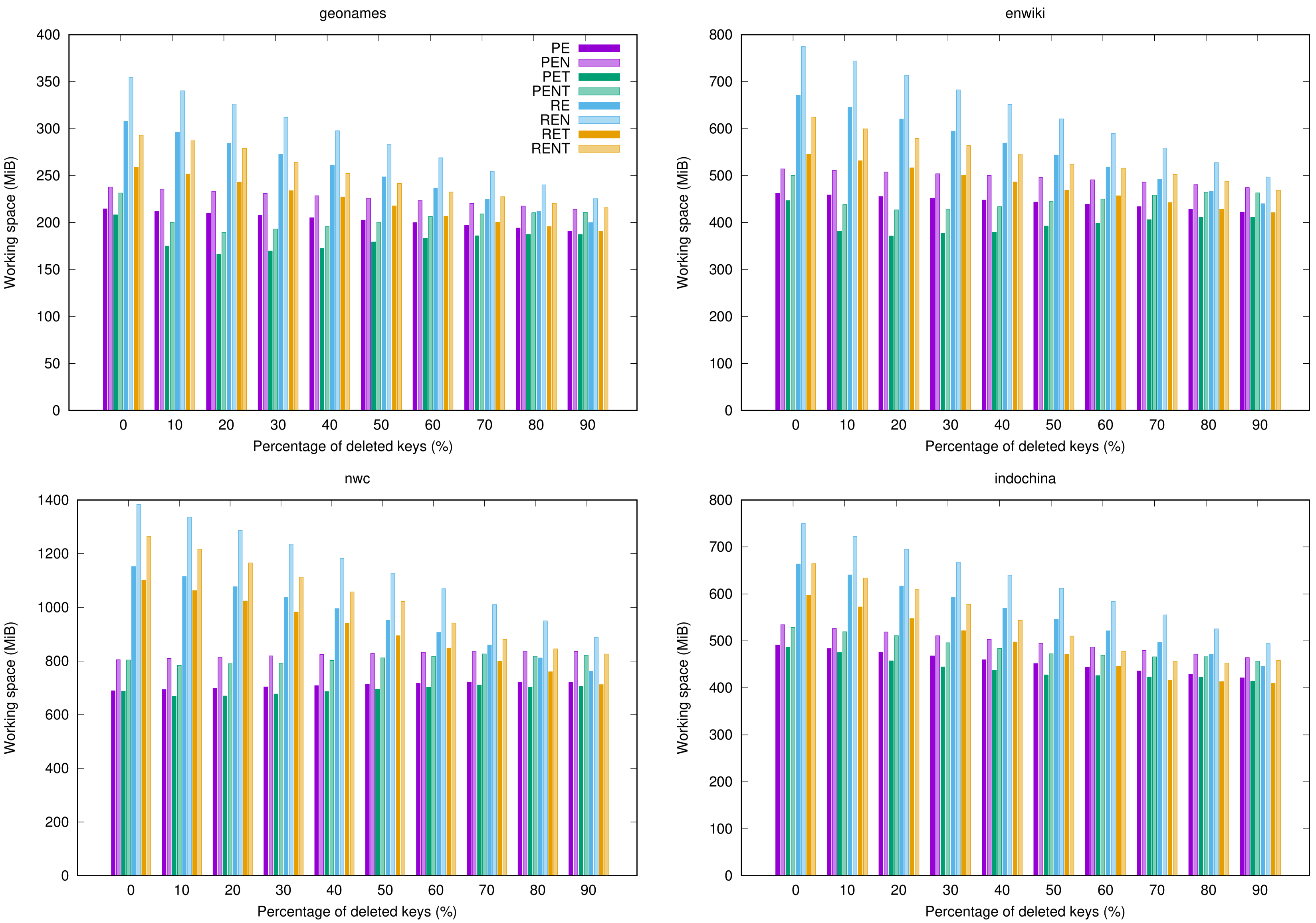


Figure 13: Experimental results of working space

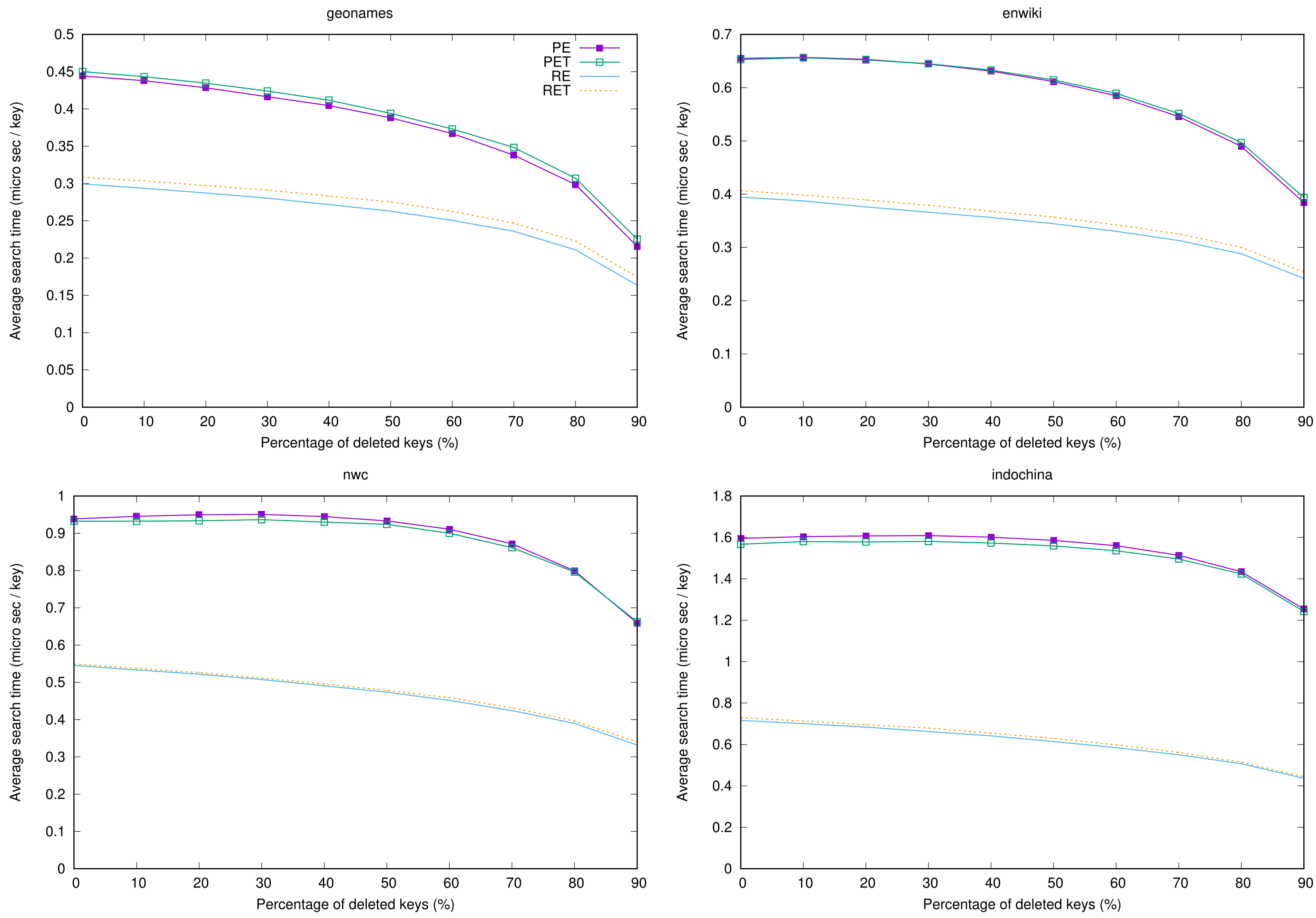


Figure 14: Experimental results of SEARCH time

5 Related work

In this section, we summarize studies related to dynamic trie dictionaries and demonstrate the value of DA dictionaries. For major practical implementations, the classical approach is to use a *ternary search tree (TST)* [31], which combines the attributes of binary search trees and digital search tries. It performs SEARCH in $O(\log |K| + k)$ time using three pointers and one character for each node, where $|K|$ is the number of keys and k is the query length. Compared to DAs that support $O(k)$ time using two pointers, TSTs are not competitive. Heinz et al. [32] proposed the *burst trie* that achieves high space efficiency by selectively collapsing chains of trie nodes into small buckets of strings that share a common prefix. Askitis and Sinha [33] proposed the *HAT-trie* that improves the burst trie using cache-conscious hash tables. Other dictionary structures include the *adaptive radix tree* [34] and the *lexicographic tree* [35]. In a recent study, Mavlyutov et al. [36] concluded that the HAT-trie is an efficient data structure for managing URI data from comparative experiments involving various data structures; however, they did not evaluate DA dictionaries.

Yoshinaga [19] provided experimental results of a comparison of DA and related dictionaries for text and binary datasets. In this study, state-of-the-art DA dictionaries were implemented using the *Cedar* library. The related dictionaries were implemented using various open source software, such as *Hat-Trie* [37] and *Sparsehash* [38]. The results showed that the Cedar library excels in terms of space efficiency and search time. Moreover, its insertion time approached that of the hashing dictionaries and outperformed other trie-based dictionaries, with the exception of *Judy trie SL* [39]. Considering that trie dictionaries can support powerful prefix-based operations required in many applications [3, 4, 5], DA tries offer great value as a dictionary structure. For such DA dictionaries, repeating updates commonly reduces space efficiency. Therefore, the proposed methods and the results given in Section 4 can contribute significantly to useful implementations of dynamic dictionaries.

Finally, we discuss recent active studies of dynamic trie representations in compressed space. Several studies have provided asymptotic worst-case results [40, 41, 42]. Poyias and Raman [43] have proposed a practical succinct trie representation named *m-Bonsai*, which is a variant of the *Bonsai* data structure [44]. m-Bonsai can represent a dynamic trie with $(1+\epsilon)n(\log \sigma + O(1))$ bits in information-theoretically optimal space, where n is the number of nodes, σ is the alphabet size, and $\epsilon > 0$ is a constant parameter. Moreover, node-to-node traversal is performed in $O(1)$ expected time; however, this study does not address dictionary implementation.

6 Conclusions

In this study, we have discussed practical rearrangement approaches of dynamic DA dictionaries and proposed efficient methods namely BLM, REBUILD, and TDM. Experimental results demonstrated that the proposed methods can support much faster rearrangement compared to existing methods. The proposed rearrangement methods require a few seconds for large datasets; thus, the time will not become a problem in practical applications. In addition, REBUILD contributes to the improvement of all dictionary operations. Therefore, many dynamic DA implementations can provide efficient rearrangement operations

and upgrade performance related to both space and time.

On the other hand, all practical dynamic dictionaries require very larger space compared to recent static compressed dictionaries [12, 45, 46, 47, 48]. With respect to DA dictionaries, BASE and CHECK use many bits since they are pointer-based arrays. Therefore, we will investigate efficient compression methods for BASE and CHECK in dynamic DAs.

Acknowledgment

We would like to thank Samuel Sangkon Lee for kindly checking English usage in the manuscript. We would like to thank the anonymous reviewers for their helpful comments.

References

[1] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960. doi: 10.1145/367390.367400.

[2] Donald E Knuth. *The art of computer programming, 3: sorting and searching*. Addison Wesley, Redwood City, CA, USA, 2nd edition, 1998.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern information retrieval*, volume 463. Addison Wesley, Boston, MA, USA, 2nd edition, 2011.

[4] Holger Bast, Christian W Mortensen, and Ingmar Weber. Output-sensitive autocompletion search. *Information Retrieval*, 11(4):269–286, 2008. doi: 10.1007/s10791-008-9048-x.

[5] Taku Kudo, Toshiyuki Hanaoka, Jun Mukai, Yusuke Tabata, and Hiroyuki Komatsu. Efficient dictionary and language model compression for input method editors. In *Proc. 1st Workshop on Advances in Text Input Methods (WTIM)*, pages 19–25, 2011.

[6] Jun-ichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, 15(9):1066–1077, 1989. doi: 10.1109/32.31365.

[7] Jun-ichi Aoe, Katsushi Morimoto, and Takashi Sato. An efficient implementation of trie structures. *Software: Practice and Experience*, 22(9):695–721, 1992. doi: 10.1002/spe.4380220902.

[8] Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, Toru Sumitomo, and Jun-ichi Aoe. A compact static double-array keeping character codes. *Information Processing & Management*, 43(1):237–247, 2007. doi: 10.1016/j.ipm.2006.04.004.

[9] Huidan Liu, Minghua Nuo, Long-Long Ma, Jian Wu, and Yeping He. Compression methods by code mapping and code dividing for chinese dictionary stored in a double-array trie. In *Proc. 5th International Joint Conference on Natural Language Processing (IJCNLP)*, pages 1189–1197, 2011.

[10] Masao Fuketa, Hiroya Kitagawa, Takuki Ogawa, Kazuhiro Morita, and Jun-ichi Aoe. Compression of double array structures for fixed length keywords. *Information Processing & Management*, 50(5):796–806, 2014. doi: 10.1016/j.ipm.2014.04.004.

[11] Shunsuke Kanda, Masao Fuketa, Kazuhiro Morita, and Jun-ichi Aoe. A compression method of double-array structures using linear functions. *Knowledge and Information Systems*, 48(1):55–80, 2016. doi: 10.1007/s10115-015-0873-0.

[12] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Compressed double-array tries for string dictionaries supporting fast lookup. *Knowledge and Information Systems*, Online first, 2016. doi: 10.1007/s10115-016-0999-8.

[13] Taku Kudo. Darts: Double-array trie system. `http://chasen.org/~taku/software/darts/`, 2008.

[14] Susumu Yata. Darts-clone: A clone of darts (double-array trie system). `https://github.com/s-yata/darts-clone`, 2011.

[15] Kazuhiro Morita, Masao Fuketa, Yoshihiro Yamakawa, and Jun-ichi Aoe. Fast insertion methods of a double-array structure. *Software: Practice and Experience*, 31(1): 43–65, 2001. doi: 10.1002/1097-024x(200101)31:1<43::aid-spe356>3.0.co;2-r.

[16] Susumu Yata, Masahiro Tamura, Kazuhiro Morita, Masao Fuketa, and Jun-Ichi Aoe. Sequential insertions and performance evaluations for double-arrays (in Japanese). In *Proc. 71st National Convention of Information Processing Society of Japan (IPSJ)*, pages 1263–1264, 2009.

[17] Tomoya Murayama and Hisatoshi Mochizuki. Proposal to fast construction method of double-array by free-space management focused on node's transitions (in Japanese). In *Proc. 78th National Convention of Information Processing Society of Japan (IPSJ)*, pages 1579–1580, 2016.

[18] Sho Seshake and Hisatoshi Mochizuki. Proposal to construction method of double array with short transition pattern (in Japanese). In *Proc. 15th Forum on Information Technology (FIT)*, pages 89–90, 2016.

[19] Naoki Yoshinaga. Cedar: C++ implementation of efficiently-updatable double-array trie. `http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/`, 2014.

[20] Brazil Inc. Groonga: An open-source fulltext search engine and column store. `http://groonga.org/`, 2017.

[21] Naoki Yoshinaga and Masaru Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In *Proc. 24th International Conference on Computational Linguistics (COLING)*, pages 1091–1102, 2014.

[22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, Cambridge, MA, USA, 3rd edition, 2009.

[23] Tshering C Dorji, El-sayed Atlam, Susumu Yata, Mahmoud Rokaya, Masao Fuketa, Kazuhiro Morita, and Jun-ichi Aoe. New methods for compression of MP double array by compact management of suffixes. *Information Processing & Management*, 46(5):502–513, 2010. doi: 10.1016/j.ipm.2009.08.004.

[24] Kazuhiro Morita, Akihiro Tanaka, Masao Fuketa, and Jun-ichi Aoe. Implementation of update algorithms for a double-array structure. In *Proc. IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, pages 494–499, 2001. doi: 10.1109/ICSMC.2001.969862.

[25] Masaki Oono, El-Sayed Atlam, Masao Fuketa, Kazuhiro Morita, and Jun-ichi Aoe. A fast and compact elimination method of empty elements from a double-array structure. *Software: Practice and Experience*, 33(13):1229–1249, 2003. doi: 10.1002/spe.545.

[26] Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, and Jun-ichi Aoe. An efficient deletion method for a minimal prefix double array. *Software: Practice and Experience*, 37(5):523–534, 2007. doi: 10.1002/spe.778.

[27] John A Dundas. Implementing dynamic minimal-prefix tries. *Software: Practice and Experience*, 21(10):1027–1040, 1991. doi: 10.1002/spe.4380211004.

[28] Makoto Yasuhara, Toru Tanaka, Jun-ya Norimatsu, and Mikio Yamamoto. An efficient language model using double-array structures. In *Proc. Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 222–232, 2013.

[29] Jun-ya Norimatsu, Makoto Yasuhara, Toru Tanaka, and Mikio Yamamoto. A fast and compact language model implementation using double-array structures. *ACM Transactions on Asian and Low-Resource Language Information Processing*, 15(4): 27, 2016. doi: 10.1145/2873068.

[30] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8): 711–726, 2004. doi: 10.1002/spe.587.

[31] Jon L Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 97, pages 360–369, 1997.

[32] Steffen Heinz, Justin Zobel, and Hugh E Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002. doi: 10.1145/506309.506312.

[33] Nikolas Askitis and Ranjan Sinha. Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal*, 19(5):633–660, 2010. doi: 10.1007/s00778-010-0183-9.

[34] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proc. IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, 2013. doi: 10.1109/ICDE.2013.6544812.

[35] Marcin Wylot, Philippe Cudre-Mauroux, and Paul Groth. Tripleprov: Efficient processing of lineage queries in a native RDF store. In *Proc. the 23rd international conference on World Wide Web (WWW)*, pages 455–466, 2014. doi: 10.1145/2566486.2568014.

[36] Ruslan Mavlyutov, Marcin Wylot, and Philippe Cudre-Mauroux. A comparison of data structures to manage URIs on the web of data. In *European Semantic Web Conference*, pages 137–151, 2015. doi: 10.1007/978-3-319-18818-8_9.

[37] Daniel C Jones. Hat-trie. `https://github.com/dcjones/hat-trie`, 2011.

[38] Google Inc. Sparsehash. `https://github.com/sparsehash/sparsehash`, 2005.

[39] Doug Baskins. *Judy IV Shop Manual*, 2002.

[40] Rajeev Raman and Srinivasa Rao Satti. Succinct dynamic dictionaries and trees. In *International Colloquium on Automata, Languages, and Programming*, pages 357–368, 2003. doi: 10.1007/3-540-45061-0_30.

[41] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Linked dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica*, 71(4): 969–988, 2015. doi: 10.1007/s00453-013-9836-6.

[42] Diego Arroyuelo, Pooya Davoodi, and Srinivasa Rao Satti. Succinct dynamic cardinal trees. *Algorithmica*, 74(2):742–777, 2016. doi: 10.1007/s00453-015-9969-x.

[43] Andreas Poyias and Rajeev Raman. Improved practical compact dynamic tries. In *Proc. 22nd International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 324–336. Springer, 2015. doi: 10.1007/978-3-319-23826-5_31.

[44] John J Darragh, John G Cleary, and Ian H Witten. Bonsai: a compact representation of trees. *Software: Practice and Experience*, 23(3):277–291, 1993. doi: 10.1002/spe.4380230305.

[45] Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):Article 10, 2010. doi: 10.1145/1868237.1868248.

[46] Julian Arz and Johannes Fischer. LZ-compressed string dictionaries. In *Proc. Data Compression Conference (DCC)*, pages 322–331, 2014. doi: 10.1109/DCC.2014.36.

[47] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *ACM Journal of Experimental Algorithmics*, 19(1):Article 1.8, 2014. doi: 10.1145/2656332.

[48] Miguel A Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. Practical compressed string dictionaries. *Information Systems*, 56: 73–108, 2016. doi: 10.1016/j.is.2015.08.008.