Knowl Inf Syst (2017) 51:1023–1042
DOI 10.1007/s10115-016-0999-8

REGULAR PAPER

Compressed double-array tries for string dictionaries supporting fast lookup

Shunsuke Kanda[1] · **Kazuhiro Morita**[1] · **Masao Fuketa**[1]

Received: 4 June 2016 / Revised: 31 August 2016 / Accepted: 26 September 2016 /
Published online: 4 October 2016

Abstract A string dictionary is a basic tool for storing a set of strings in many kinds of applications. Recently, many applications need space-efficient dictionaries to handle very large datasets. In this paper, we propose new compressed string dictionaries using improved double-array tries. The double-array trie is a data structure that can implement a string dictionary supporting extremely fast lookup of strings, but its space efficiency is low. We introduce approaches for improving the disadvantage. From experimental evaluations, our dictionaries can provide the fastest lookup compared to state-of-the-art compressed string dictionaries. Moreover, the space efficiency is competitive in many cases.

Keywords Trie · Double-array · Compressed string dictionaries · Data management · String processing and indexing

1 Introduction

In the advanced information society, huge amounts of data are represented as strings such as documents, web pages, URLs, genome data. For that reason, many researchers have tackled to propose efficient algorithms and data structures for handling string data. The data structures include a *string dictionary* for storing a set of strings. It implements mapping strings to identifiers (basically, integer IDs), that is, it has to support two retrieval operations: `lookup` returns the ID of a given string, and `access` returns the string of a given ID. As the mapping is very useful for string processing and indexing, the string dictionary is a basic tool in many kinds of applications for natural language processing, information retrieval, semantic web graphs, bioinformatics, geographic information systems and so on. On the other hand, recently, there are many real examples that the size of string dictionaries becomes critical

✉ Shunsuke Kanda
shnsk.knd@gmail.com

1 Department of Information Science and Intelligent Systems, Tokushima University, Minamijosanjima 2-1, Tokushima 770-8506, Japan

problems for very large datasets [28]. That is to say, many applications need compressed string dictionaries.

A popular data structure to implement the string dictionary is a *trie* [14,23] that is an edge-labeled tree. As strings are registered on root-to-leaf paths by merging the common prefixes, it contributes to data compression and can support powerful prefix-based operations such as enumeration of all strings included as prefixes of a given string. The operations can be useful in specific applications such as stemmed searches [5] and auto-completions [6] in natural language dictionaries.

There are many researches about space-efficient tries. In particular, trie representations using succinct labeled trees [3,7,30,31] and XBW [13] provide good space efficiency. However, their node-to-node traversals are slow because many bit operations are used for random memory access, that is, the `lookup` and `access` operations become slow. To solve this problem for static compressed string dictionaries, Grossi and Ottaviano [18] present a new data structure inspired in the path decomposition trie [12]. It enables to support fast traversal by reducing the number of random memory accesses.

As for other state-of-the-art works, Martínez-Prieto et al. [28] introduce and practically evaluate static compressed string dictionaries based on some techniques. In short, the dictionaries based on Front-Cording [36] provide good performances in time/space trade-off. Their `access` operations are fast especially. The dictionaries based on hashing [10] are good choices if fast `lookup` is needed. Arz and Fischer [4] propose Lempel–Ziv (LZ) compressed string dictionaries that adapt the LZ78 parsing [43] to `lookup` and `access` operations. The dictionaries are effective for datasets containing many often repeated substrings.

We focus on a *double-array (DA) trie* proposed by Aoe [1]. DA is a popular trie representation supporting the fastest node-to-node traversal. It is used in many applications at present such as MeCab,[1] Groonga[2] and so on. String dictionaries using the DA trie can support fast `lookup` and `access`, but the scalability is a problem for large datasets because DA is a pointer-based data structure. Although several compressed DA tries are proposed [15,21,40], we cannot adopt them to the string dictionaries because they cannot support `access` instead of compression.

This paper proposes a new compressed DA trie supporting fast `lookup` and `access` operations by using different approaches with previous compressed DA tries. In addition, this paper shows the advantages of our string dictionaries from experimental evaluations for real datasets. Compared to the original DA trie, our data structure can implement string dictionaries in half or smaller space. Compared to other state-of-the-art compressed string dictionaries, our dictionary can provide the fastest `lookup`. Moreover, the space efficiency is competitive in many cases.

The rest of the paper is organized as follows. Section 2 provides basic definitions and introduces related data structures. Section 3 proposes a new compressed DA trie which can implement the string dictionary. Section 4 proposes a technique for improving operation speed of our data structure. Section 5 shows experimental evaluations. Section 6 concludes the paper and indicates our future works. In addition, we provide the source code at https://github.com/kamp78/cda-tries for the reader interested in further comparisons.

[1] Yet another Part-of-Speech and Morphological Analyzer at http://taku910.github.io/mecab/.

[2] An open-source full-text search engine and column store at http://groonga.org/.

2 Preliminaries

This section introduces data structures on which our research is related, after we give basic definitions as follows.

We denote an array A that consists of n elements $A[0]A[1]\dots A[n-1]$ as $A[0,n)$ and the array fragment $A[i,j+1)$ that consists of the elements $A[i]A[i+1]\dots A[j]$ as $A[i,j]$. Notation $(a)_2$ denotes a binary representation of value a, and $|(a)_2|$ denotes the code length, that is, the bits needed to represent a. For example, $(9)_2 = 1001$ and $|(9)_2| = 4$. 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 $\lceil 2.4 \rceil = 3$. The base of logarithm is 2 throughout the paper.

2.1 Succinct data structures

Given a bit array B, we define two basic operations: $\texttt{rank}(B,i)$ returns the number of 1s in $B[0,i)$, and $\texttt{select}(B,i)$ returns the position of the $i+1$ th occurrence of 1 in B. Suppose $B[0,8) = [00100110]$, $\texttt{rank}(B,6) = 2$ and $\texttt{select}(B,1) = 5$.

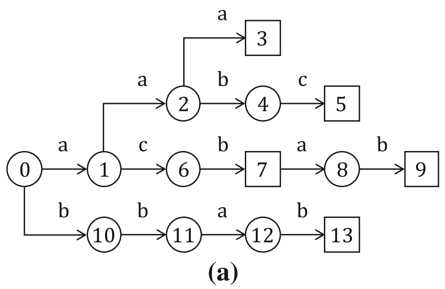
As these operations are at the heart of many compressed data structures, several practical implementations are proposed [17,22,32]. Our string dictionaries will use the implementation that Okanohara and Sadakane [32] introduce as the *verbative*. For $B[0,n)$, the verbative supports $\texttt{rank}$ in $O(1)$ and $\texttt{select}$ in $O(\log n)$ using extra $o(n)$ bits.

2.2 String dictionaries and tries

Strings are drawn from a finite alphabet Σ of size σ. A string dictionary is a data structure that stores a set of strings, $\mathcal{S} \subset \Sigma^*$. Dictionary $\mathcal{S}$ supports two primitive operations:

- $\texttt{lookup}(q)$ returns the ID if $q \in \mathcal{S}$.
- $\texttt{access}(i)$ returns the string with ID $i \in [0, |\mathcal{S}|)$.

Trie. A trie [14,23] is an edge-labeled tree structure that is well used to implement the string dictionary. Figure 1a shows an example of the trie. The trie is built by merging common prefixes of strings and by giving a character on each edge. Strings are registered on the root-to-leaf paths. When a string is the prefix of another one, it terminates on an internal node. To identify terminal nodes, we define a bit array $\texttt{TERM}$ in which $\texttt{TERM}[s] = 1$ iff node s is terminal. For example, we define $\texttt{TERM}[0,14) = [00010101010001]$ for the trie of Fig. 1a, and $\texttt{TERM}[7] = 1$ denotes that internal node 7 is the terminal for "acb".



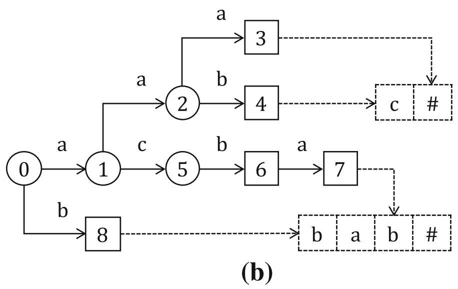


Fig. 1 Tries for $\mathcal{S}$ = {"aaa", "aabc", "acb", "acbab", "bbab"}. The *square nodes* denote terminals of strings. **a** Trie. **b** MP-trie

The trie can carry out `lookup` and `access` as follows. For `lookup`(q), we traverse nodes from the root with characters of q. If reached node s is terminal, that is, $\mathtt{TERM}[s] = 1$, the string ID is returned by $\mathtt{rank}(\mathtt{TERM}, s) \in [0, |\mathcal{S}|)$. For `access`$(i)$, we obtain the terminal node s corresponding to the ID i by $\mathtt{select}(\mathtt{TERM}, i)$. The string is extracted by traversing nodes from node s in reverse and by concatenating the characters on the path.

We define two operations to traverse nodes: $\mathtt{child}(s, c)$ returns the child of node s with character c, and $\mathtt{parent}(s)$ returns the pair of the parent of node s and the edge character between the nodes. Operations `lookup` and `access` are supported by `child` and `parent`, respectively. That is to say, trie representations have to support the two operations to implement the string dictionary.

Examples. In Fig. 1a, $\mathtt{child}(1, \text{`c'}) = 6$ and $\mathtt{parent}(4) = (2, \text{`b'})$. Operations $\mathtt{lookup}(\text{"acb"}) = 2$ and $\mathtt{access}(2) = \text{"acb"}$ are carried out as follows. For `lookup`, nodes are traversed with query "acb" as $\mathtt{child}(0, \text{`a'}) = 1$, $\mathtt{child}(1, \text{`c'}) = 6$ and $\mathtt{child}(6, \text{`b'}) = 7$. From $\mathtt{TERM}[7] = 1$, the string ID is returned by $\mathtt{rank}(\mathtt{TERM}, 7) = 2$. For `access`, the terminal node is given by $\mathtt{select}(\mathtt{TERM}, 2) = 7$. The edge labels are extracted by $\mathtt{parent}(7) = (6, \text{`b'})$, $\mathtt{parent}(6) = (1, \text{`c'})$ and $\mathtt{parent}(1) = (0, \text{`a'})$. Concatenating the characters in reverse obtains "acb".

Minimal prefix trie. There are several trie variants for compaction. The variants include a *minimal prefix trie (MP-trie)* [2,11] focusing on that the trie cannot merge the suffixes of strings. The MP-trie keeps only minimal prefixes of strings as nodes and the rest suffixes as strings separately. Moreover, Yata et al. [41] introduce that the common suffixes of the separated strings can be unified. Figure 1b shows an example of the MP-trie. From Fig. 1, we can see that the number of nodes is reduced from 14 to 9. Special terminal character '#' (basically, the ASCII zero code) is added at the end of each separated string. Leaf nodes become terminals instead of reduced nodes and have links to the strings.

2.3 Double-arrays

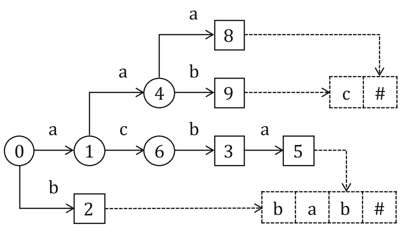
DA [1] represents a trie by using two integer arrays called `BASE` and `CHECK`. Each index corresponds to each node. When the trie has the edge from node s to node t with character c, DA satisfies the following equations[3]:

$$\mathtt{BASE}[s] \oplus \mathtt{code}(c) = t \text{ and } \mathtt{CHECK}[t] = s, \tag{1}$$

where $\mathtt{code}(c) \in [0, \sigma)$ returns the numerical code integer of character c. DA can carry out `child` and `parent` by using the simple equations as follows. For $\mathtt{child}(s, c)$, child t is given by $\mathtt{BASE}[s] \oplus \mathtt{code}(c) = t$ and is returned if $\mathtt{CHECK}[t] = s$. For $\mathtt{parent}(s)$, it is carried out by $(\mathtt{CHECK}[s], \mathtt{char}(\mathtt{BASE}[\mathtt{CHECK}[s]] \oplus s))$, where `char` is an invert function of `code` such that $\mathtt{char}(\mathtt{code}(c)) = c$. DA can provide extremely fast traversal.

DA uses two additional arrays for the MP-trie: a bit array `LEAF` in which $\mathtt{LEAF}[s] = 1$ iff node s is a leaf, and a character array `TAIL` storing separated strings. In $\mathtt{LEAF}[s] = 1$, $\mathtt{BASE}[s]$ has a link from node s to `TAIL`. Figure 2 shows an example of DA representing the MP-trie of Fig. 1b. From this figure, we can see that the node IDs are arranged to satisfy Eq. (1). The arranged nodes can include several invalid IDs such as ID 7. The invalid nodes are identified as empty elements.

[3] Operator $\oplus$ denotes an XOR (exclusive OR) operation. While traditional implementations use a PLUS ($+$), the XOR ($\oplus$) is often substituted in recent ones such as [42] and Darts-clone at https://github.com/s-yata/darts-clone.



(a)

	0	1	2	3	4	5	6	7	8	9
BASE	1	4	0	5	8	2	2		5	4
CHECK		0	0	6	1	3	1		4	4
LEAF	0	0	1	0	0	1	0	0	1	1
TERM	0	0	1	1	0	1	0	0	1	1

	0	1	2	3	4	5
TAIL	b	a	b	#	c	#

(b)

Fig. 2 DA representation of the MP-trie of Fig. 1b. The numerical code integers are code('a') = 0, code('b') = 1 and code('c') = 2. The invert function provides char(0) = 'a', char(1) = 'b' and char(2) = 'c'. The node IDs are arranged to satisfy Eq. (1)

Examples. In Fig. 2, child(1, 'c') = 6 and parent(9) = (4, 'b') are carried out as follows. For child, the child ID is given by BASE[1] $\oplus$ code('c') = 4 $\oplus$ 2 = 6. Node 6 is returned from CHECK[6] = 1. For parent, the parent ID is given by CHECK[9] = 4. The edge character between nodes 4 and 9 is given by char(BASE[4] $\oplus$ 9) = char(8 $\oplus$ 9) = char(1) = 'b'. As a result, the pair (4, 'b') is returned. For the link from node 5 to TAIL[2], this TAIL position is given by BASE[5] = 2 because of LEAF[5] = 1.

Construction algorithm. DA is built by arranging node IDs to satisfy Eq. (1). Let E be a set of edge characters from node s, the child IDs are arranged by using xcheck(E) that returns an arbitrary integer *base* such that nodes *base*$\oplus$code(c) are invalid for each character $c \in E$, that is, the elements are empty. When BASE[s] is defined as BASE[s] $\leftarrow$ xcheck(E), the child IDs t are also defined as $t \leftarrow$ BASE[s] $\oplus$ code(c) and CHECK[t] $\leftarrow s$ for each character $c \in E$. In static construction, DA is built by repeating this process from the root recursively.

Previous compressed DAs. In practice, the space usage of DA is very large because BASE and CHECK use 32- or 64-bit integers to represent node pointers. Several methods are proposed to compress the arrays. The *compact double-array (CDA)* [40] is a useful and popular one. CDA changes the right part of Eq. (1) into CHECK[t] = c. That is to say, each CHECK element is represented in $\log \sigma$ bits by storing characters instead of integers. In practice, CHECK becomes compact because of $\log \sigma = 8$ as byte characters. However, CDA cannot support parent because the CHECK does not indicate parent nodes. Therefore, CDA cannot support access, that is, cannot implement the string dictionary.

Kanda et al. [21] propose another compressed DA, called the *double-array using linear functions (DALF)*, that empirically represents BASE with 8-bit integers. However, this method cannot also support access because it is based on CDA. Although Fuketa et al. [15] also propose a CDA-based compact trie representation, its applications are limited to fixed-length strings such as zip codes.

2.4 Directly addressable codes

Variable-length coding is the main part of data compression [34]. It can represent a fixed-length array of integers using variable-length codes with less space. A problem with the codes is how to directly extract arbitrary integers. Brisaboa et al. [9] propose the *directly addressable codes (DACs)* to solve the problem practically.

DACs implement direct extraction by combining rank with Vbyte coding [35]. Suppose that DACs represent an array of integers P. Given a parameter b, we split $(P[i])_2$ into blocks of b bits, $p_{(i,k_i)}, \ldots, p_{(i,2)}, p_{(i,1)}$ where $k_i = \lceil |(P[i])_2|/b \rceil$. For example in $P[i] = 49$ and

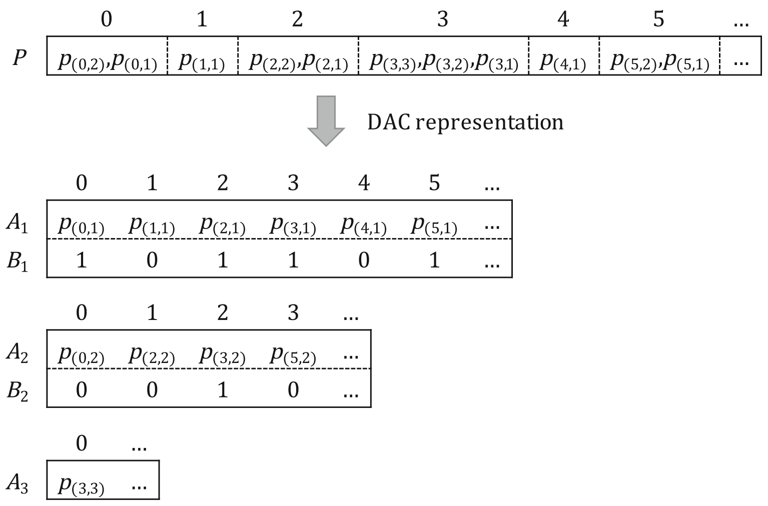

0 1 2 3 4 5 ...
P
p(0,2),p(0,1) p(1,1) p(2,2),p(2,1) p(3,3),p(3,2),p(3,1) p(4,1) p(5,2),p(5,1) ...
DAC representation
A1 p(0,1) p(1,1) p(2,1) p(3,1) p(4,1) p(5,1) ...
B1 1 0 1 1 0 1 ...
A2 p(0,2) p(2,2) p(3,2) p(5,2) ...
B2 0 0 1 0 ...
A3 p(3,3) ...

Fig. 3 Example of a DAC representation for array P

$b = 2$, we split $(49)_2 = 110001$ into $p_{(i,3)} = 11$, $p_{(i,2)} = 00$ and $p_{(i,1)} = 01$. First, arrays A_j store all the j-th blocks for $1 \leq j$ until all blocks are stored. Next, bit arrays B_j are defined such that $B_j[i] = 1$ iff $A_j[i]$ stores the last block. Figure 3 shows an example of a DAC representation.

Let $i_1, i_2, \ldots, i_{k_i}$ denote the path storing $P[i]$, that is, $A_1[i_1] = p_{(i,1)}, A_2[i_2] = p_{(i,2)}, \ldots, A_{k_i}[i_{k_i}] = p_{(i,k_i)}$. We can extract $P[i]$ by following the path and by concatenating the A_j values. The start position i_1 is given by $i_1 = i$, and the after ones $i_2, \ldots, i_{k_i}$ are given by the following;

$$i_{j+1} = \mathtt{rank}(B_j, i_j) \;\; (B_j[i_j] = 1). \tag{2}$$

From $B_j[i_j] = 0$, we can identify that $A_j[i_j]$ stores the last block. For example in Fig. 3, $P[5]$ is extracted by concatenating values $A_1[5] = p_{(5,1)}$ and $A_2[3] = p_{(5,2)}$. The second position 3 is given by $\mathtt{rank}(B_1, 5) = 3$, and we can see that $A_2[3] = p_{(5,2)}$ is the last block from $B_2[3] = 0$.

Let N denote the maximum integer in P, DACs can represent P using arrays $A_1, \ldots, A_L$ and $B_1, \ldots, B_{L-1}$, where $L = \lceil |(N)_2|/b \rceil$. Note that DACs do not use B_L because that A_L stores only the last blocks is trivial. Since A_j is a fixed-length array, extracting an integer in a DAC representation takes $O(L)$ time in the worst case.

An advantage of DACs is the fast extraction. Brisaboa et al. [9] show that DACs can provide faster extraction than other directly extractable variable-length codes in practice. In particular, byte-oriented DACs with $b = 8$ are well used because very fast extraction can be supported. For compressed string dictionaries, HashDAC and RPDAC in [28] apply DACs to array compression. In addition, a construction algorithm introduced in Sect. 3.2 is compatible with the byte-oriented DACs. Therefore, our data structure will use them to compress DA and to maintain the fast operations.

3 New compressed double-array trie

DA's scalability is caused by storing node pointers in BASE and CHECK arrays. General implementation represents the arrays as fixed-length ones with 32- or 64-bit integers. Therefore, their space usages become very large. DACs can represent such arrays using variable-length codes with directly extraction, but representing `BASE` and `CHECK` including many large integers is inefficient in space and time.

We present a new data structure built by the following steps: Step 1 transforms `BASE` and `CHECK` into arrays including many small integers, and Step 2 represents the arrays using DACs. Section 3.1 presents the transformation technique. Section 3.2 shows a construction algorithm to support the transformation. Section 3.3 explains our data structure.

3.1 XOR transformation

It is a technique that compresses an array of integers by using differences between values and indices. It transforms an array of integers P into array P_X such that $P_X[i] = P[i] \oplus i$. We can extract $P[i]$ from $P_X[i]$ as $P_X[i] \oplus i = (P[i] \oplus i) \oplus i = P[i]$ because of $i \oplus i = 0$. Suppose that P is partitioned into blocks of length r that is a power of 2, we give the following theorem for P_X.

Theorem 1 *Integer $P_X[i]$ can be represented in $\log r$ bits for $P[i]$ such that $\lfloor P[i]/r \rfloor = \lfloor i/r \rfloor$.*

Proof When r is a power of 2, $\lfloor i/r \rfloor$ denotes to right shift $(i)_2$ by $\log r$ bits. In $\lfloor P[i]/r \rfloor = \lfloor i/r \rfloor$, $(P[i])_2$ and $(i)_2$ consist of the same bits except for the lowest $\log r$ bits. Therefore, $(P[i] \oplus i)_2$ except for the lowest $\log r$ bits becomes zero, that is, $P_X[i] = P[i] \oplus i$ can be represented in $\log r$ bits. □

Examples. Let $P[23] = 21$ in $r = 4$. Function $\lfloor 23/4 \rfloor = 5$ denotes to right shift $(23)_2 = \mathbf{101}11$ by $\log 4 = 2$ bits as $(5)_2 = \mathbf{101}$. Similarly, $\lfloor 21/4 \rfloor = 5$ denotes to right shift $(21)_2 = \mathbf{101}01$ by 2 bits. Binaries $\mathbf{101}11$ and $\mathbf{101}01$ consist of the same bits except for the lowest 2 bits because of $\lfloor 23/4 \rfloor = \lfloor 21/4 \rfloor$. Therefore, $P_X[23] = 21 \oplus 23 = 2$ can be represented in 2 bits as $\mathbf{101}11 \oplus \mathbf{101}01 = \mathbf{000}10$.

3.2 Construction algorithm

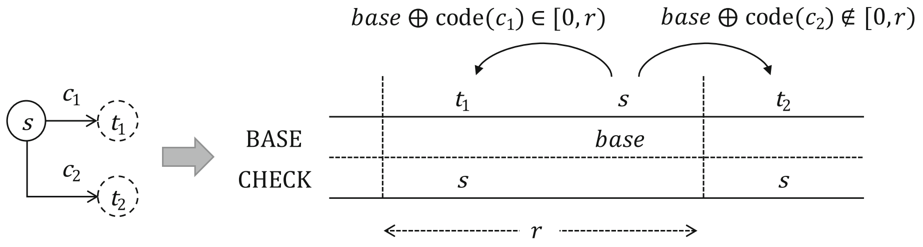
DACs can efficiently represent an array including many b-bit integers because such integers are represented by using only the first array A_1. Let P include many integers satisfying the condition of Theorem 1 in $r = 2^b$, most P_X values are in $\log r = b$ bits. For `BASE` and `CHECK`, the values can be freely determined as long as Eq. (1) is satisfied. Therefore, we can obtain `BASE` and `CHECK` values satisfying the condition in $r = 2^b$.

We present a function $\texttt{ycheck}_r$ that targets to determine `BASE` values satisfying the condition. Let E be a set of edge characters from node s, XCDA defines `BASE` values as $\texttt{BASE}[s] \leftarrow \texttt{ycheck}_r(E, s)$.

Algorithm 1 $\texttt{ycheck}_r(E, s)$

```
1: for $base \leftarrow \lfloor s/r \rfloor \cdot r, (\lfloor s/r \rfloor + 1) \cdot r$ do
2:     if Nodes $base \oplus \texttt{code}(c)$ are invalid for each $c \in E$ then
3:         return $base$                          ▷ $\lfloor base/r \rfloor = \lfloor s/r \rfloor$
4:     end if
5: end for
6: return $\texttt{xcheck}(E)$                    ▷ $\lfloor \texttt{xcheck}(E)/r \rfloor \neq \lfloor s/r \rfloor$
```

Function $\texttt{ycheck}_r(E, s)$ targets to determine $\texttt{BASE}[s]$ such that $\lfloor \texttt{BASE}[s]/r \rfloor = \lfloor s/r \rfloor$. This loop searches such $\texttt{BASE}[s]$ satisfying Eq. (1) on the block $\lfloor s/r \rfloor$. If the loop cannot find it, $\texttt{BASE}[s]$ is determined in the same manner as the conventional algorithm.



$base \oplus \mathrm{code}(c_1) \in [0, r)$
$base \oplus \mathrm{code}(c_2) \notin [0, r)$
s
c_1
t_1
c_2
t_2
BASE
CHECK
t_1
s
t_2
$base$
s
s
r

Fig. 4 The relation between node s and its children t_1 and t_2. Suppose that $\mathtt{BASE}[s] = base$ satisfies $\lfloor base/r \rfloor = \lfloor s/r \rfloor$, $\lfloor \mathtt{CHECK}[t_1]/r \rfloor = \lfloor s/r \rfloor = \lfloor t_1/r \rfloor$ is also satisfied in $\mathtt{code}(c_1) \in [0, r)$

Function $\mathtt{ycheck}_r(E, s)$ is effective for characters c such that $\mathtt{code}(c) \in [0, r)$ as the following reason. Let t be the child of node s with such character c, the following equation is satisfied;

$$\lfloor \mathtt{BASE}[s]/r \rfloor = \lfloor (\mathtt{BASE}[s] \oplus \mathtt{code}(c))/r \rfloor = \lfloor t/r \rfloor. \quad (3)$$

When $\lfloor \mathtt{BASE}[s]/r \rfloor = \lfloor s/r \rfloor$ is satisfied, Eq. (3) and the right part of Eq. (1) give $\lfloor s/r \rfloor = \lfloor t/r \rfloor = \lfloor \mathtt{CHECK}[t]/r \rfloor$. That is to say, we only have to search $\mathtt{BASE}[s]$ such that $\lfloor \mathtt{BASE}[s]/r \rfloor = \lfloor s/r \rfloor$ in order to obtain $\mathtt{BASE}[s]$ and $\mathtt{CHECK}[t]$ satisfying the condition of Theorem 1 (see Fig. 4).

In practice, $\sigma \leq 256$ always holds because byte characters are used to edge labels. Therefore, $\mathtt{ycheck}_r$ can obtain BASE and CHECK values satisfying the condition in $r = 2^8 = 256$. In other words, the function is compatible with the byte-oriented DACs with $b = 8$. The effectiveness will be shown in Sect. 5.

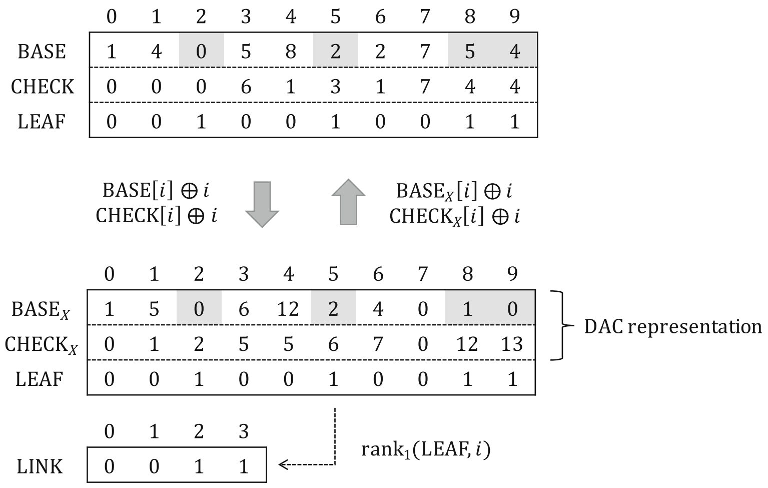
3.3 Data structure

We call our data structure the *XOR-compressed double-array (XCDA)*. Let $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ be arrays such that $\mathtt{BASE}_X[i] = \mathtt{BASE}[i] \oplus i$ and $\mathtt{CHECK}_X[i] = \mathtt{CHECK}[i] \oplus i$, respectively. XCDA is built by representing $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ using the byte-oriented DACs. From Sect. 3.2, $\mathtt{ycheck}_r$ can provide $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ including many 8-bit integers. Therefore, XCDA can provide compact trie representations.

On the other hand, it is necessary to discuss how to represent empty elements and TAIL links. General DAs represent empty elements by using invalid values such as negative integers. The links are determined randomly corresponding to TAIL positions. These $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ values become large when using the XOR transformation. Therefore, XCDA represents the values as follows.

- As $\mathtt{CHECK}[t] = s$ means that the parent of node t is node s, inequation $s \neq t$ always holds. We can consider $\mathtt{CHECK}[i] = i$ as empty elements. The $\mathtt{CHECK}_X$ values always become zero because of $\mathtt{CHECK}_X[i] = \mathtt{CHECK}[i] \oplus i = i \oplus i = 0$. If $\mathtt{BASE}[s]$ is empty, $\mathtt{CHECK}[s]$ is also empty. Therefore, we do not have to identify whether BASE elements are empty. XCDA sets $\mathtt{BASE}[i] = i$ for empty elements.
- XCDA represents TAIL links by using the first array A_1 and an additional array LINK. Suppose $\mathtt{BASE}[s] = pos$ in $\mathtt{LEAF}[s] = 1$, $\mathtt{BASE}_X[s]$ stores the lowest b bits of $(pos)_2$ and $\mathtt{LINK}[\mathtt{rank}(\mathtt{LEAF}, s)]$ stores the rest bits. XCDA supports fast extraction of TAIL links because only A_1 and LINK are used.

Examples. Figure 5 shows an example of XCDA for the DA of Fig. 2. The shaded elements denote TAIL links. Except for the links, $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ are built by using the XOR



	0	1	2	3	4	5	6	7	8	9
BASE	1	4	0	5	8	2	2	7	5	4
CHECK	0	0	0	6	1	3	1	7	4	4
LEAF	0	0	1	0	0	1	0	0	1	1

	0	1	2	3	4	5	6	7	8	9
BASE_X	1	5	0	6	12	2	4	0	1	0
CHECK_X	0	1	2	5	5	6	7	0	12	13
LEAF	0	0	1	0	0	1	0	0	1	1

	0	1	2	3
LINK	0	0	1	1

Fig. 5 The transformed arrays in $b = 2$ from the DA of Fig. 2

transformation. For example, $\mathtt{CHECK}_X[3]$ is transformed by $\mathtt{CHECK}[3] \oplus 3 = 6 \oplus 3 = 5$. Empty $\mathtt{BASE}_X[7]$ and $\mathtt{CHECK}_X[7]$ become zero by setting $\mathtt{BASE}[7] = 7$ and $\mathtt{CHECK}[7] = 7$. For the TAIL link $\mathtt{BASE}[9] = 4$, the lowest b bits of $(\mathtt{BASE}[9])_2 = (4)_2 = 100$ and the rest bits are stored in $\mathtt{BASE}_X[9]$ and $\mathtt{LINK}[\mathtt{rank}(\mathtt{LEAF}, 9)] = \mathtt{LINK}[3]$, respectively. Let $b = 2$, $\mathtt{BASE}_X[9] = 00$ and $\mathtt{LINK}[3] = 1$. XCDA is built by representing the $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ using DACs.

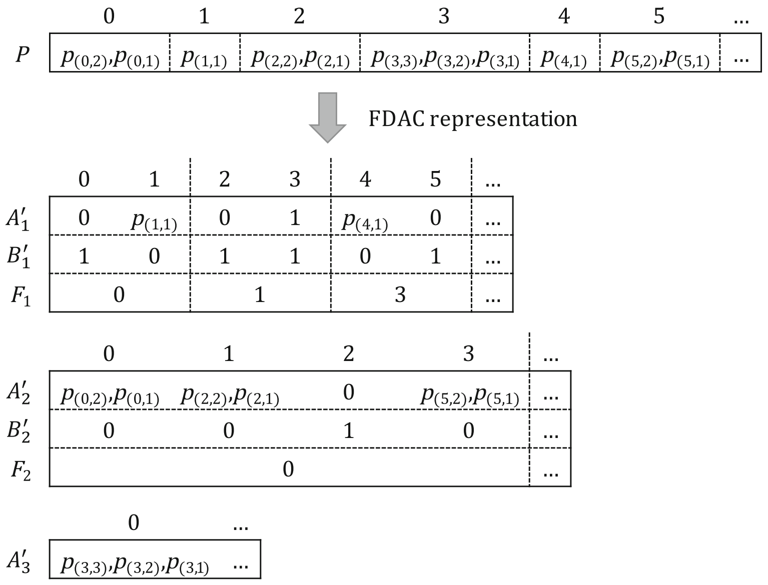
It is very easy to extract original BASE and CHECK values from the XCDA. Value $\mathtt{CHECK}[3] = 6$ is extracted by $\mathtt{CHECK}_X[3] \oplus 3 = 5 \oplus 3 = 6$. From $\mathtt{BASE}[7] = 0$, we can identify that this element is empty. From $\mathtt{LEAF}[9] = 1$, the link $(\mathtt{BASE}[9])_2 = (4)_2 = 100$ is extracted by concatenating values $\mathtt{LINK}[\mathtt{rank}(\mathtt{LEAF}, 9)] = \mathtt{LINK}[3] = 1$ and $\mathtt{BASE}_X[9] = 00$.

4 Improvement for fast operations

Section 3 introduces techniques to transform BASE and CHECK into $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ including many small integers, respectively. XCDA represents $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ by using DACs. On the other hand, all $\mathtt{BASE}_X$ and $\mathtt{CHECK}_X$ values are not represented in b bits because of Eq. (1). While DACs extract such values by using rank in constant time, many bit operations are used in practice. Therefore, the retrieval speed of XCDA using DACs is not competitive to that of DA using plain pointers. This section presents new pointer-based DACs called *Fast DACs (FDACs)*, supporting directly extraction without rank.

4.1 Pointer-based fast DACs

For simplicity, we introduce FDACs corresponding to DACs in Sect. 2.4. More precisely, $P[i]$ is extracted through the same path, $i_1, i_2, \ldots, i_{k_i}$. Figure 6 shows an example of a FDAC representation. In this figure, as Fig. 3, $P[5]$ is extracted by following the 5 and 3 positions on the first and second arrays, respectively. Such FDACs consist of the following arrays:



0 1 2 3 4 5 ...
P
p(0,2),p(0,1) p(1,1) p(2,2),p(2,1) p(3,3),p(3,2),p(3,1) p(4,1) p(5,2),p(5,1) ...
FDAC representation
A'1 0 p(1,1) 0 1 p(4,1) 0 ...
B'1 1 0 1 1 0 1 ...
F1 0 1 3 ...
A'2 p(0,2),p(0,1) p(2,2),p(2,1) 0 p(5,2),p(5,1) ...
B'2 0 0 1 0 ...
F2 0 ...
A'3 p(3,3),p(3,2),p(3,1) ...

Fig. 6 Example of a FDAC representation corresponding to the DACs of Fig. 3. We assume the DACs with $b = 1$ and the FDACs with $b_1 = 1, b_2 = 2$ and $b_3 = 3$, that is, $r_1 = 2$ and $r_2 = 4$

- Arrays $A'_1, A'_2, \ldots, A'_L$ with $b_1, b_2, \ldots, b_L$ bit integers, where $b_1 = b, b_2 = 2 \cdot b, \ldots, b_L = L \cdot b$.
- Bit arrays $B'_1, B'_2, \ldots, B'_{L-1}$ including the same bits as $B_1, B_2, \ldots, B_{L-1}$ in Sect. 2.4.
- Arrays $F_1, F_2, \ldots, F_{L-1}$ whose each element corresponds to each block, assuming that A'_j and B'_j are partitioned into blocks of length $r_j = 2^{b_j}$.

On the path $i_1, i_2, \ldots, i_{k_i}$, values $A'_j[i_j]$ for $1 \leq j < k_i$ indicate the next positions i_{j+1} by keeping the results of $\texttt{rank}(B'_j, i_j)$, and value $A'_{k_i}[i_{k_i}]$ keeps $P[i]$ directly. In order that $A'_j[i_j]$ can indicate i_{j+1} in b_j bits, arrays F_j keep the results of rank for each head of blocks on A'_j, as $F_j[x] = \texttt{rank}(B'_j, r_j \cdot x)$. Arrays A'_j store the differences as $A'_j[i_j] = \texttt{rank}(B'_j, i_j) - F_j[\lfloor i_j/r_j \rfloor]$. Each element of A'_j can be represented in $b_j = \log r_j$ bits because $A'_j[i_j] \in [0, r_j)$ is always satisfied. FDACs change Eq. (2) into Eq. (4);

$$i_{j+1} = A'_j[i_j] + F_j[\lfloor i_j/r_j \rfloor] \ \ (B'_j[i_j] = 1). \tag{4}$$

We explain how to carry out the extraction using the example of Fig. 6. When $P[5]$ is extracted, the first position 5 of A'_1 is given in the same manner as DACs. From $B'_1[5] = 1$, we can see that the second position exists. While DACs get the second position by $\texttt{rank}(B_1, 5) = 3$, FDACs can get it without rank as $A'_1[5] + F_1[\lfloor 5/r_1 \rfloor] = A'_1[5] + F_1[2] = 0 + 3 = 3$. Thanks to $F_1[2]$ keeping $\texttt{rank}(B'_1, r_1 \cdot 2) = \texttt{rank}(B'_1, 4) = 3$, $A'_1[5]$ can represent the results of rank in $b_1 = 1$ bit. We can see that $A'_2[3]$ directly keeps $P[5]$ because of $B'_2[3] = 0$, and the extraction is done.

FDACs can represent an array of integers P when every integer can be represented in any arrays $A'_1, \ldots, A'_L$. Although the extraction time of FDACs is equal to that of DACs in $O(L)$, FDACs can follow the path $i_1, \ldots, i_{k_i}$ at high speed without rank.

On the other hand, the space efficiency becomes low when using arrays $A'_2, \ldots, A'_L$ frequently, because each A'_j element uses $j \cdot b$ bits, while each A_j element uses b bits.

Fortunately, we can obtain many $\texttt{BASE}_X$ and $\texttt{CHECK}_X$ values represented in A'_1 because of $\texttt{ycheck}_r$. Therefore, FDACs are excellent with XCDA.

Byte-oriented FDACs. We do not have to separately manage A'_j and B'_j because B'_j does not use `rank`. Therefore, FDACs can improve the cache efficiency of DACs by allocating $A'_j[i]$ and $B'_j[i]$ on contiguous space. The byte-oriented FDACs define $b_1 = 7, b_2 = 15, \ldots$ so that $A'_j[i]$ and $B'_j[i]$ are represented on the same byte space.

4.2 Code arrangement

Function $\texttt{ycheck}_r$ works for characters c such that $\texttt{code}(c) \in [0, 128)$ when using the byte-oriented FDACs with $b_1 = 7 = \log 128$ bits. There are no problems for ASCII characters because $\sigma \leq 128$ always holds. On the other hand, byte characters given by splitting multi-byte ones such as UTF-8 in Japanese and Chinese often satisfy $128 < \sigma$. This subsection introduces a technique to utilize $\texttt{ycheck}_r$ for the byte-oriented FDACs.

We improve `code` into $\texttt{code}_F$ such that $\texttt{code}_F(c) \in [0, \sigma)$ returns the order number of character c when sorting characters in the string dictionary by frequency in descending order. That is to say, $\texttt{code}_F$ returns integers in $[0, r)$ for the top r characters of appearance frequency in the dictionary. Most characters are empirically represented as $\texttt{code}_F(c) \in [0, 128)$ because character frequency of real datasets is biased.

Suppose that a string dictionary is built from all page titles of the Japanese Wikipedia of Jan. 2015.[4] The character encoding is UTF-8. While the dictionary satisfies $\sigma = 189$, 99.7 % characters c in the dictionary are represented as $\texttt{code}_F(c) \in [0, 128)$.

5 Experimental evaluations

This section analyzes practical performances of XCDAs on real-world datasets. We compare XCDAs with other string dictionaries and give evaluations of our data structure in practice.

5.1 Setting

We carried out the experiments on Quad-Core Intel Xeon 2×2.4 GHz, 16 GB RAM. All string dictionaries were implemented in C++. They were compiled using Apple LLVM version 7.0.2 (clang-700.1.81) with optimization -O3.
Datasets. We used the following real datasets of several types:

- `geonames`: Geographic names on the *asciiname* column from the geonames dump.[5]
- `nwc-2010`: Japanese word $n$grams in the Nihongo Web Corpus 2010.[6]
- `jawiki-titles`: All page titles from the Japanese Wikipedia of Jan. 2015.
- `enwiki-titles`: All page titles from the English Wikipedia of Feb. 2015.
- `uk-2005`: URLs of a 2005 crawl by the UbiCrawler [8] on the `.uk` domain.[7]
- `gene-DNA`: All substrings of 12 characters found in the gene-DNA dataset from Pizza&Chili Corpus.[8]

[4] https://dumps.wikimedia.org.

[5] http://download.geonames.org/export/dump/allCountries.zip.

[6] http://dist.s-yata.jp/corpus/nwc2010/ngrams/word/over999/filelist.

[7] http://data.law.di.unimi.it/webdata/uk-2005/uk-2005.urls.gz.

[8] http://pizzachili.dcc.uchile.cl/texts/dna/dna.gz.

Table 1 Information about datasets

	Size (MB)	$\lvert\mathcal{S}\rvert$	Ave. length	σ	# of nodes	$\lvert$`TAIL`$\rvert$
`geonames`	106.1	6,784,722	15.6	96	11,378,833	8,733,434
`nwc-2010`	460.8	20,722,756	22.2	180	52,047,795	548,133
`jawiki-titles`	33.9	1,518,205	22.3	189	3,516,248	5,234,145
`enwiki-titles`	238.2	11,519,354	20.7	199	25,749,451	23,108,877
`uk-2005`	2855.5	39,459,925	72.4	103	117,568,967	289,826,785
`gene-DNA`	198.5	15,265,943	13.0	16	20,688,222	39,244

Table 1 summarizes the information about each dataset: the raw size in MB, number of different strings, average number of characters per string when including a terminator, number of different characters used in the dictionary, number of nodes and length of `TAIL` in the MP-trie.

Data structures. We compared performances of XCDAs to previous DA tries and state-of-the-art compressed string dictionaries. For XCDAs, there are four patterns as follows:

- *XCDA-x* using the byte-oriented DACs and `xcheck`.
- *XCDA-y* using the byte-oriented DACs and $\texttt{ycheck}_{256}$.
- *FXCDA-x* using the byte-oriented FDACs and `xcheck`.
- *FXCDA-y* using the byte-oriented FDACs and $\texttt{ycheck}_{128}$.

For previous DA tries, we tested the original DA [1], CDA [40] and DALF [21], representing the MP-trie. Note that CDA and DALF cannot support `access`. For DALF parameters, we chose $x = 8$, $bsize = 512$, $\alpha = 128$ and $gain = 1.0$ in common with the experiments in [21]. DALF represents the MP-trie using `LEAF` and `LINK` in the same manner as Sect. 3.3 because the `BASE` consists of 8-bit integers. We implemented `xcheck` using fast algorithms in [29]. We used $\texttt{code}_F$ for all structures because there are no disadvantages. Our library at https://github.com/kamp78/cda-tries packs these implementations and shows more technical details.

As for the state of the art, *Cent* is the centroid path-decomposed trie and *Cent-rp* is the Re-Pair [26] compressed one, from [18]. We also tested *PFC*, *HTFC-rp* and *HashDAC-rp* from [28]. PFC is a plain Front-Coding dictionary. HTFC-rp is a Hu-Tucker [20] Front-Coding dictionary compressed by using Re-Pair. HashDAC-rp is a hashing dictionary compressed by using Re-Pair and DACs. For the Front-Coding dictionaries, we chose bucket size 8 as the best space/time trade-off in the same manner as [18] and [4]. In addition, we tested bucket sizes 2 and 4 for HTFC-rp in order to observe faster operations. For HashDAC-rp, Martínez-Prieto et al. [28] evaluate 5 load factors. Since their performances do not change significantly, we chose load factor $\alpha = 0.5$ supporting the fastest `lookup`. While LZ-compressed string dictionaries [4] are effective for synthetic datasets containing many often repeated substrings, Cent-rp outperforms the LZ-dictionaries for real datasets from previous experiments. Therefore, our experiments did not include the LZ-dictionaries. Cent and Cent-rp were implemented by using `path_decomposed_tries`.[9] PFC, HTFC-rp and HashDAC-rp were implemented by using `libCSD`.[10]

[9] https://github.com/ot/path_decomposed_tries.

[10] https://github.com/migumar2/libCSD.

Table 2 Experimental results about percentages of values on each level in DACs and FDACs

	1st	2nd	3rd	4th
(a) `geonames`				
XCDA-x	86.04	13.58	0.38	0.00
XCDA-y	**88.94**	10.67	0.39	0.00
FXCDA-x	78.21	21.16	0.63	–
FXCDA-y	**83.88**	15.45	0.68	–
(b) `nwc-2010`				
XCDA-x	92.07	7.72	0.21	0.00
XCDA-y	**93.74**	6.05	0.21	0.00
FXCDA-x	87.32	12.33	0.35	–
FXCDA-y	**90.89**	8.76	0.36	–
(c) `jawiki-titles`				
XCDA-x	88.20	11.51	0.29	0.00
XCDA-y	**90.75**	8.97	0.28	0.00
FXCDA-x	81.00	18.55	0.45	–
FXCDA-y	**86.00**	13.48	0.52	–
(d) `enwiki-titles`				
XCDA-x	88.62	10.98	0.39	0.01
XCDA-y	**90.81**	8.79	0.39	0.01
FXCDA-x	82.37	17.01	0.62	–
FXCDA-y	**86.42**	12.94	0.65	–
(e) `uk-2005`				
XCDA-x	92.88	7.02	0.10	0.00
XCDA-y	**94.25**	5.66	0.10	0.00
FXCDA-x	88.00	11.83	0.17	–
FXCDA-y	**90.44**	9.40	0.16	–
(f) `gene-DNA`				
XCDA-x	94.04	5.90	0.06	0.00
XCDA-y	**94.55**	5.38	0.06	0.00
FXCDA-x	90.09	9.80	0.11	–
FXCDA-y	**90.80**	9.09	0.11	–

The highest percentages in DACs and FDACs are denoted in bold font

5.2 Results

We first observe DACs and FDACs using `xcheck` and $\texttt{ycheck}_r$. Next, we evaluate the practical performance of our data structure in static string dictionaries.

For construction algorithms. Table 2 shows the percentages of values for each level of DACs and FDACs using `xcheck` and $\texttt{ycheck}_r$. In DACs, A_j can represent $8 \cdot j$-bit integers and the maximum level is 4. In FDACs, A'_1, A'_2 and A'_3 can represent 7-, 15- and 32-bit integers, respectively. Each column represents the percentages of represented values in each level, that is, the sum of percentages in each row becomes 100 %.

From the table, the 1st level for all cases can represent many values, while values on the 2nd or deeper levels always arise to satisfy Eq. (1). Function $\texttt{ycheck}_r$ provides better results than xcheck, especially in FDACs whose allocation of the 1st level is smaller. Therefore, $\texttt{ycheck}_r$ can contribute to improvement of our data structure.

For string dictionaries. Tables 3, 4 and 5 show the experimental results about the construction time, percentage of compression ratio between the data structure and the raw data sizes, and average running times of lookup and access. The best results within each category are denoted in bold font. To measure the running times of lookup, we chose 1 million random strings from each dataset. The running times of access were measured for 1 million IDs corresponding to the random strings. Each test was averaged on 10 runs. We could not build HashDAC-rp for uk-2005 because the construction complexity exceeded the memory resources of our computational configuration. Moreover, it did not complete the construction on gene-DNA in 6 h; hence, we had to kill the process.

When comparing the construction algorithms in the new DA tries, using $\texttt{ycheck}_r$ slightly outperforms using xcheck because more values are represented in the 1st level. Function $\texttt{ycheck}_r$ provides better compression ratios in all cases because they obediently depend on the percentages in Table 2. The lookup and access times also depend largely on the percentages; therefore, $\texttt{ycheck}_r$ provides faster operations in most cases. There are no significant problems in construction. Thus, using $\texttt{ycheck}_r$ is a better choice.

When comparing the new DA tries using $\texttt{ycheck}_r$, FXCDA-y provides faster operations than XCDA-y in all cases because of removing rank and improving cache efficiency. For the compression ratios, FXCDA-y is superior in nwc-2010, uk-2005 and gene-DNA, while XCDA-y is superior in geonames, jawiki-titles and enwiki-titles. Although FDACs use more space in the 2nd or deeper levels to embed rank information, the 1st level A'_1 consists of 7-bit integers, less space than 8-bit integers on A_1 in DACs, because A'_1 and B'_1 are not managed separately. Therefore, FXCDA-y becomes compact when the percentage in the 1st level is high. On all aspects, FXCDA-y excels in the new DA tries. In what follows, we compare it to other data structures.

Compared to the previous DA tries, FXCDA-y is 1.7–2.6 times smaller than DA and solves the problem that we cannot apply the previous DA tries to the compressed string dictionaries. CDA always provides the fastest lookup because of improvement of the cache efficiency from CHECK compaction, but the scalability of BASE is a problem. DALF provides competitive compression ratios, but the lookup becomes slow for large datasets such as uk-2005 because of technological factors as follows. DALF is built by arranging nodes in breadth-first order, while general DA tries are built by arranging nodes in depth-first order. In DALF, cache misses can occur frequently in parent–child traversal, that is, the lookup can become slow especially for a long query in a large trie. On the other hand, FXCDA-y supports stable and fast lookup and access.

Compared to Cent and PFC not compressed by Re-Pair, FXCDA-y provides competitive or smaller space except for Cent on gene-DNA. Moreover, it provides the fastest lookup. The running time of FXCDA-y is up to 3 and 2 times faster than those of Cent and PFC, respectively. For access, PFC is the fastest, while FXCDA-y is faster than Cent. For the construction cost, PFC is much faster, yet both FXCDA-y and Cent are practical as static string dictionaries.

Compared to Cent-rp, HTFC-rp and HashDAC-rp, FXCDA-y is larger because of the powerful Re-Pair compression. FXCDA-y is up to 2.6, 1.8 and 1.5 times larger than Cent-rp, HTFC-rp and HashDAC-rp, respectively. On the other hand, FXCDA-y can provide much faster lookup because the compressions pose a decrease in speed. The running time of FXCDA-y is up to 3.1 and 2.0 times faster than those of Cent-rp and HashDAC-rp, respectively. Compared to HTFC-rp, FXCDA-y is up to 5.5 times faster in bucket size 8.

Table 3 Experimental results about string dictionaries for `geonames` and `nwc-2010`

	Constr. (s)	Cmpr. (%)	`lookup` (μs/str)	`access` (μs/ID)
(a) `geonames`				
New DA tries				
XCDA-x	5.7	51.8	1.12	1.52
XCDA-y	5.8	**51.2**	1.10	1.51
FXCDA-x	**5.5**	55.1	0.96	1.32
FXCDA-y	5.7	52.8	**0.93**	**1.29**
Previous DA tries				
DA	**4.9**	95.8	0.61	**0.95**
CDA	5.0	63.7	**0.49**	–
DALF	9.5	**52.8**	0.80	–
State-of-the-art dictionaries				
Cent	13.6	51.5	2.01	2.13
Cent-rp	33.8	**31.5**	2.10	2.17
PFC	**0.6**	60.5	1.61	**0.47**
HTFC-rp (2)	51.5	59.0	2.39	0.82
HTFC-rp (4)	211.9	42.7	2.80	1.14
HTFC-rp (8)	125.1	34.4	3.50	1.79
HashDAC-rp	298.9	48.0	**1.28**	0.92
`(b) nwc-2010`				
New DA tries				
XCDA-x	16.9	36.6	1.92	2.58
XCDA-y	17.0	36.2	1.91	2.59
FXCDA-x	**16.0**	37.3	1.61	2.22
FXCDA-y	16.2	**35.7**	**1.57**	**2.20**
Previous DA tries				
DA	**13.7**	92.4	1.00	**1.58**
CDA	14.7	58.5	**0.83**	–
DALF	35.4	**34.2**	1.88	–
State-of-the-art dictionaries				
Cent	39.7	42.2	2.73	2.89
Cent-rp	76.4	**16.9**	2.66	2.81
PFC	**1.9**	38.2	2.10	**0.51**
HTFC-rp (2)	201.3	49.3	3.04	0.88
HTFC-rp (4)	343.2	30.7	3.34	1.14
HTFC-rp (8)	423.2	21.5	3.77	1.63
HashDAC-rp	1456.6	29.1	**1.72**	1.08

For smaller bucket sizes, FXCDA-y maintains faster `lookup`, while the compression ratio becomes competitive. In addition, using the Re-Pair compression devotes large construction costs. Therefore, the speed differences can overcome the disadvantage of FXCDA-y in space.

We finally remark that an advantage of our data structure is to support the fastest `lookup` in compressed string dictionaries. Its construction cost is also practical in static string dictionaries. Our data structure is useful in applications emphasizing response speed for string

Table 4 Experimental results about string dictionaries for `jawiki-titles` and `enwiki-titles`

	Constr. (s)	Cmpr. (%)	`lookup` (µs/str)	`access` (µs/ID)
(a) `jawiki-titles`				
New DA tries				
XCDA-x	1.5	53.5	0.85	1.24
XCDA-y	1.5	**53.0**	0.83	1.22
FXCDA-x	**1.4**	55.9	0.70	1.04
FXCDA-y	1.5	54.0	**0.66**	**1.02**
Previous DA tries				
DA	**1.3**	100.3	0.52	**0.90**
CDA	**1.3**	69.1	**0.40**	–
DALF	2.6	**56.1**	0.61	–
State-of-the-art dictionaries				
Cent	3.5	92.0	1.57	1.81
Cent-rp	9.9	**32.4**	1.67	1.89
PFC	**0.2**	61.0	1.35	**0.49**
HTFC-rp (2)	22.5	57.0	2.12	0.85
HTFC-rp (4)	43.4	41.0	2.68	1.32
HTFC-rp (8)	69.5	32.6	3.62	2.25
HashDAC-rp	110.0	35.3	**1.33**	0.85
(b) `enwiki-titles`				
New DA tries				
XCDA-x	12.6	50.6	1.58	2.09
XCDA-y	12.8	**50.1**	1.56	2.10
FXCDA-x	**12.1**	52.8	1.33	**1.82**
FXCDA-y	12.5	51.1	**1.31**	**1.82**
Previous DA tries				
DA	**11.0**	98.1	0.82	**1.31**
CDA	11.3	65.7	**0.67**	–
DALF	23.3	**51.5**	1.39	–
State-of-the-art dictionaries				
Cent	24.5	52.4	2.40	2.48
Cent-rp	73.5	**31.6**	2.62	2.65
PFC	**1.2**	59.6	1.97	**0.62**
HTFC-rp (2)	930.1	56.9	2.87	1.00
HTFC-rp (4)	712.3	40.8	3.40	1.50
HTFC-rp (8)	936.7	32.6	4.49	2.51
HashDAC-rp	780.7	41.0	**1.66**	1.31

queries, and such applications exist in large numbers. For example, *inverted indexes*, used in search engines and so on, handle the dictionaries to find the positions in a text from a keyword composed of natural languages [5]. While this literature shows that the dictionary size does not become a critical problem from Heaps' law [19], Martínez-Prieto et al. [28] show the significance of compressed natural language dictionaries because the size on Web collections becomes far more than a gigabyte. Search engines requiring fast and effective

Table 5 Experimental results about string dictionaries for `uk-2005` and `gene-DNA`

	Constr. (s)	Cmpr. (%)	`lookup` (μs/str)	`access` (μs/ID)
(a) `uk-2005`				
New DA tries				
XCDA-x	72.0	25.4	3.42	4.36
XCDA-y	73.3	25.3	3.41	4.29
FXCDA-x	**70.2**	25.6	**2.66**	**3.50**
FXCDA-y	71.7	**25.2**	2.70	3.54
Previous DA tries				
DA	**65.9**	43.8	1.95	**2.93**
CDA	68.0	31.5	**1.63**	–
DALF	110.1	**24.1**	6.00	–
State-of-the-art dictionaries				
Cent	129.5	27.7	3.59	4.14
Cent-rp	472.7	**17.5**	4.02	4.47
PFC	**6.1**	37.3	**3.04**	**0.67**
HTFC-rp (2)	5908.9	42.3	5.44	2.05
HTFC-rp (4)	7765.0	26.3	6.39	2.90
HTFC-rp (8)	12,598.4	18.3	7.96	4.41
HashDAC-rp	–	–	–	–
(b) `gene-DNA`				
New DA tries				
XCDA-x	5.5	38.0	1.29	1.65
XCDA-y	4.0	38.0	1.30	1.64
FXCDA-x	5.2	37.8	1.21	**1.33**
FXCDA-y	**3.9**	**37.7**	**1.03**	**1.33**
Previous DA tries				
DA	**4.5**	87.4	0.58	**0.88**
CDA	6.0	55.3	**0.46**	–
DALF	7.8	**33.0**	0.54	–
State-of-the-art dictionaries				
Cent	22.9	21.2	3.24	3.47
Cent-rp	24.4	**14.2**	3.18	3.26
PFC	**1.0**	38.4	**1.68**	**0.42**
HTFC-rp (2)	12.2	43.3	2.01	0.55
HTFC-rp (4)	10.0	27.5	2.23	0.78
HTFC-rp (8)	9.3	20.6	2.38	0.98
HashDAC-rp	–	–	–	–

responses can be supported by the compressed dictionaries with fast `lookup` rather than optional `access`. For other natural language applications, input method editors (IMEs) also handle large dictionaries [24]. In particular, limited configurations such as mobile computers need sophisticated data structures. Prefix-based lookup operations are utilized to build a lattice (or word graph) or implement a suggestion feature for a user input. In more detail, so-called *common-prefix-lookup* operation, which returns all strings included as prefixes of a query, is the most important to report all registered substrings in the input. The operation is often used in natural language processing such as Japanese morphological analyses [25], especially in languages not written with a space between words. In IMEs, the lattice is built by calling it for all suffixes in the input; therefore, `lookup` is constantly carried out and its time is significant. Although `access` (also called *reverse-lookup*) is used to support reconversion, its frequency is less. For other applications, domain name servers map domain names to IP addresses in large numbers and must provide request very fast. Thus, there are many applications requiring fast `lookup` because it is the most primitive operation as a dictionary structure. Our data structure can contribute much to them.

6 Conclusion

We have presented XCDA that a new compressed DA structure. Unlike the previous compressed DAs, XCDA tries can implement compressed string dictionaries supporting fast operations. Our experimental evaluations have shown that our dictionaries can support the fastest `lookup` compared to the state of the art. Moreover, the space efficiency is competitive in many cases.

While we have discussed string dictionaries, DAs can be also used to implement other data structures. For example, they include directed acyclic word graphs [38], deterministic finite automata [16,27], *n* gram language models [37]. XCDA can contribute to their compression. For our future works, we will propose the compression methods using XCDA. In addition, XCDA can use dynamic update algorithms for DA tries [29,33,39]. Therefore, we will also propose dynamic XCDA tries.

References

1. Aoe J (1989) An efficient digital search algorithm by using a double-array structure. IEEE Trans Softw Eng 15(9):1066–1077
2. Aoe J, Morimoto K (1992) An efficient implementation of trie structures. Softw Pract Exp 22(9):695–721
3. Arroyuelo D, Cánovas R, Navarro G, Sadakane K (2010) Succinct trees in practice. In: Proceedings of the 11st meeting on algorithm engineering and experimentation (ALENEX), pp. 84–97
4. Arz J, Fischer J (2014) LZ-compressed string dictionaries. In: Proceedings of the data compression conference (DCC), pp. 322–331
5. Baeza-Yates R, Ribeiro-Neto B (2011) Modern information retrieval, 2nd edn. Addison Wesley, Boston
6. Bast H, Mortensen CW, Weber I (2008) Output-sensitive autocompletion search. Inf Retr 11(4):269–286
7. Benoit D, Demaine ED, Munro JI, Raman R, Raman V, Rao SS (2005) Representing trees of higher degree. Algorithmica 43(4):275–292
8. Boldi P, Codenotti B, Santini M, Vigna S (2004) Ubicrawler: a scalable fully distributed web crawler. Softw Pract Exp 34(8):711–726
9. Brisaboa NR, Ladra S, Navarro G (2013) DACs: bringing direct access to variable-length codes. Inf Process Manag 49(1):392–404
10. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms, 3rd edn. MIT press, Cambridge
11. Dundas JA (1991) Implementing dynamic minimal-prefix tries. Softw Pract Exp 21(10):1027–1040
12. Ferragina P, Grossi R, Gupta A, Shah R, Vitter JS (2008) On searching compressed string collections cache-obliviously. In: Proceedings of the 27th symposium on principles of database systems (PODS), ACM, pp. 181–190
13. Ferragina P, Luccio F, Manzini G, Muthukrishnan S (2009) Compressing and indexing labeled trees, with applications. J ACM 57(1):4
14. Fredkin E (1960) Trie memory. Commun ACM 3(9):490–499
15. Fuketa M, Kitagawa H, Ogawa T, Morita K, Aoe J (2014) Compression of double array structures for fixed length keywords. Inf Process Manag 50(5):796–806

16. Fuketa M, Morita K, Aoe J (2014) Comparisons of efficient implementations for DAWG. In: Proceedings of the 7th international conference on computer science and information technology (ICCSIT)
17. González R, Grabowski S, Mäkinen V, Navarro G (2005) Practical implementation of rank and select queries. In: Poster proceedings of the 4th workshop on experimental and efficient a lgorithms (WEA), pp. 27–38
18. Grossi R, Ottaviano G (2014) Fast compressed tries through path decompositions. ACM J Exp Algorithm 19(1):3–4
19. Heaps HS (1978) Information retrieval: computational and theoretical aspects. Academic Press Inc, Orlando
20. Hu TC, Tucker AC (1971) Optimal computer search trees and variable-length alphabetical codes. SIAM J Appl Math 21(4):514–532
21. Kanda S, Fuketa M, Morita K, Aoe J (2016) A compression method of double-array structures using linear functions. Knowl Inf Syst 48(1):55–80
22. Kim DK, Na JC, Kim JE, Park K (2005) Efficient implementation of rank and elect functions for succinct representation. Proceedings of the 4th international workshop on experimental and efficient algorithms (WEA), LNCS 3503. Springer, New York, pp 315–327
23. Knuth DE (1998) The art of computer programming, 3: sorting and searching, 2nd edn. Addison Wesley, Redwood City
24. Kudo T, Hanaoka T, Mukai J, Tabata Y, Komatsu H (2011) Efficient dictionary and language model compression for input method editors. In: Proceedings of the 1st workshop on advances in text input methods (WTIM), pp. 19–25
25. Kudo T, Yamamoto K, Matsumoto Y (2004) Applying conditional random fields to Japanese morphological analysis. In: Proceedings of the conference on empirical methods in natural language processing (EMNLP), pp. 230–237
26. Larsson NJ, Moffat A (1999) Offline dictionary-based compression. In: Proceedings of the data compression conference (DCC), pp. 296–305
27. Maeda A, Mizushima K (2008) A compressed-array representation of automata and its application to programming language (in Japanese). In: Proceedings of the 49th IPSJ programming symposium, pp. 49–54
28. Martínez-Prieto MA, Brisaboa N, Cánovas R, Claude F, Navarro G (2016) Practical compressed string dictionaries. Inf Syst 56:73–108
29. Morita K, Fuketa M, Yamakawa Y, Aoe J (2001) Fast insertion methods of a double-array structure. Softw Pract Exp 31(1):43–65
30. Munro JI, Raman V (2001) Succinct representation of balanced parentheses and static trees. SIAM J Comput 31(3):762–776
31. Navarro G, Sadakane K (2014) Fully functional static and dynamic succinct trees. ACM Trans Algorithms 10(3):16
32. Okanohara D, Sadakane K (2007) Practical entropy-compressed rank/select dictionary. In: Proceedings of the 9th meeting on algorithm engineering and expermiments (ALENEX), pp. 60–70
33. Oono M, Atlam ES, Fuketa M, Morita K, Aoe J (2003) A fast and compact elimination method of empty elements from a double-array structure. Softw Pract Exp 33(13):1229–1249
34. Salomon D (2008) A concise introduction to data compression. Springer, London
35. Williams HE, Zobel J (1999) Compressing integers for fast file access. Comput J 42(3):193–201
36. Witten IH, Moffat A, Bell TC (1999) Managing gigabytes: compressing and indexing documents and images. Morgan Kaufmann, San Francisco
37. Yasuhara M, Tanaka T, Norimatsu J, Yamamoto M (2013) An efficient language model using double-array structures. In: Proceedings of the conference on empirical methods in natural language processing (EMNLP), pp. 222–232
38. Yata S, Morita K, Fuketa M, Aoe J (2008) Fast string matching with space-efficient word graphs. In: Proceedings of the 4th international conference on innovations in information technology (IIT), pp. 79–83
39. Yata S, Oono M, Morita K, Fuketa M, Aoe J (2007) An efficient deletion method for a minimal prefix double array. Softw Pract Exp 37(5):523–534
40. Yata S, Oono M, Morita K, Fuketa M, Sumitomo T, Aoe J (2007) A compact static double-array keeping character codes. Inf Process Manag 43(1):237–247
41. Yata S, Oono M, Morita K, Sumitomo T, Aoe J (2006) Double-array compression by pruning twin leaves and unifying common suffixes. In: Proceedings of the 1st international conference on computing and informatics (ICOCI), pp. 1–4
42. Yoshinaga N, Kitsuregawa M (2014) A self-adaptive classifier for efficient text-stream processing. In: Proceedings of the 24th international conference on computational linguistics (COLING), pp. 1091–1102

43. Ziv J, Lempel A (1978) Compression of individual sequences via variable-rate coding. IEEE Trans Inf Theory 24(5):530–536

Shunsuke Kanda received B.Sc. and M.Sc. degrees in information science and intelligent systems from Tokushima University, Japan, in 2014 and 2016, respectively. He is currently a Ph.D. student at Tokushima University. He is a student member of the Information Processing Society in Japan. His research interests are data structures for string processing and indexing.

Kazuhiro Morita received B.Sc., M.Sc. and Ph.D. degrees in information science and intelligent systems from Tokushima University, Japan, in 1995, 1997 and 2000, respectively. He had been a research assistant from 2000 to 2006 in information science and intelligent systems, Tokushima University, Japan. He is currently an associate professor in the Department of Information Science and Intelligent Systems, Tokushima University, Japan. His research interests are sentence retrieval from huge text databases, double-array structures and binary search tree.

Masao Fuketa received B.Sc., M.Sc. and Ph.D. degrees in information science and intelligent systems from Tokushima University, Japan, in 1993, 1995 and 1998, respectively. He had been a research assistant and an associate professor from 1998 to 2000 and from 2000 to 2015 in information science and intelligent systems, Tokushima University, Japan, respectively. He is currently a professor in the Department of Information Science and Intelligent Systems, Tokushima University, Japan. He is a member of the Information Processing Society in Japan and the Association for Natural Language Processing of Japan. His research interests are information retrieval and natural language processing.