Tokushima University

Ph.D. Thesis

Space- and Time-Efficient String Dictionaries

空間効率と時間効率の良い文字列辞書

Author:
Shunsuke Kanda

Supervisor:
Prof. Masao Fuketa

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Graduate School of Advanced Technology and Science
Department of Information Science and Intelligent Systems

March 2018

Abstract

In modern computer science, the management of massive data is a fundamental problem because the amount of data is growing faster than we can easily handle them. Such data are often represented as strings such as documents, Web contents and genomics data; therefore, data structures and algorithms for space-efficient string processing have been developed by many researchers. In this thesis, we focus on a string dictionary that is an in-memory data structure for storing a set of strings. It has been traditionally used to manage vocabulary in natural language processing and information retrieval. The size of the dictionaries is not problematic because of Heaps' Law. However, string dictionaries in recent applications, such as Web search engines, RDF stores, geographic information systems and bioinformatics, need to handle very large datasets. As the space usage of string dictionaries is a significant issue in those applications, it is necessary to develop space-efficient data structures.

If limited to static applications, existing data structures have already achieved very high space efficiency by exploiting succinct data structures and text compression techniques. For example, state-of-the-art string dictionaries can be implemented in space up to 5% of the original dataset size. However, there remain trade-off problems among space efficiency, lookup-time performance and construction costs. For example, Re-Pair that is a powerful compression technique for a text is well used to obtain high space efficiency and fast lookup operations, but the time and space needed during construction are not practical for very large datasets.

In our work, we attempt two approaches to solve the problems. One develops novel string dictionaries based on double-array tries. The double-array trie can support the fastest lookup operation, but its space usage is very large. We propose how to improve the disadvantage. The novel string dictionaries can support very fast lookup operations, although the space usage is somewhat larger compared to existing compressed ones. The other addresses to improve the high construction costs of applying Re-Pair by introducing an alternative compression strategy using dictionary encoding. Our strategy enables lightweight construction, while providing competitive space efficiency and operation speed.

If allowing dynamic operations such as insertion and deletion, there are some sophisticated data structures. While they improve the space efficiency by reducing pointer overheads, they still consume much larger space than the static string dictionaries because

some redundancy for dynamic update is needed. In fact, the number of dynamic applications handling very large string dictionaries is fewer than that of static ones; however, a part of applications as in Web crawler and Semantic Web need to compact dynamic string dictionaries. In this thesis, we propose a new data structure through path decomposition that is a technique for constructing cache-friendly trie structures. Although the path decomposition has been utilized to static dictionary implementation, we adopt it for dynamic dictionary construction with a different approach. From experiments, we show that our data structure can implement the smallest dynamic string dictionary.

We also address problems of dynamic double-array trie dictionaries. The dynamic dictionaries are well-used as in stream text processing and search engines, but the space efficiency tends to decrease with key updates. Although we can still maintain high efficiency using existing methods, these methods have practical problems of time and functionality. In this thesis, we present several practical rearrangement methods to solve these problems. We show that the proposed methods can support significantly faster rearrangement via read-world datasets.

As all our methods are heuristic and practical, we sometimes do not provide the theoretical guarantees; however, we always show the practical performances through careful experiments using large real-world datasets. Furthermore, most of our implementations are provided as open source libraries under the MIT License.

Acknowledgements

I would like to thank everyone who supported me in my research and life at Tokushima University.

First of all, I would like to express the deepest appreciation to my supervisor, Professor Masao Fuketa. He not only supported my research life but also taught many things to me, how to research, how to think, how to behave, and so on. In addition, he provided me many good opportunities. I am truly proud that I could be his first Ph.D. student. I would like also to express my gratitude to Associate Professor, Kazuhiro Morita. He always gave advice to me when I had a trouble. I would like to thank Emeritus Professor, Jun-ichi Aoe. He supported the first half of my laboratory life. I would like to thank Professor Kenji Kita and Professor Masami Shishibori, who are the members of the committee of my thesis.

I would like to express my appreciation to all students in the laboratory. Thanks to them, I was able to live a satisfying research life. I am deeply grateful to my English teacher, Noriko Zotzman. She spent a lot of time to touch up my manuscripts and taught me good English usage. My English skills were improved thanks to her. I would like also to express my gratitude to Professor Samuel Sangkon Lee for kindly checking English usage in the paper published in SPE.

A part of this work was supported by Japan Society for the Promotion of Science. The contents of this thesis are partly based on our papers published in the KAIS, SPE and DBSJ journals and in the proceedings of the 3rd Innovate-Data and 24th SPIRE. I would like to thank the anonymous reviewers for their helpful comments.

Last but not least, I really thank my parents, brothers, and best friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In modern computer science, the management of massive data is a fundamental problem because the amount of data is growing faster than we can easily handle them. For instance, billions of people are posting on social networking services such as Twitter and Facebook; the number is thousands to tens of thousands per second. Although analyzing and mining such large data are very useful in many real-world applications, the amount of data obviously exceeds the size that we can analyze through naïve data structures in main memory. Use of secondary or tertiary storage can be one of the temporary solutions, but the difference of data processing speed between main and external memories is very large. In addition, there are many situations where memory is limited and a large secondary memory is not at hand, such as routers and smartphones. Therefore, many researchers have investigated space-efficient data structures for in-memory data processing [106].

Such data are often handled as sequences of characters, or *strings*, such as documents, Web contents, genomics data, sensor data, and behavioral history; therefore, many compact data structures for string processing have been developed. This thesis focuses on *string dictionaries* to store a set of strings in main memory and to provide fast query processing. In short, it is an associative array with string keys, such as `std::map<std::string,`*`type`*`>` in C++ or `map[string]`*`type`* in Go. It is a very underlying and familiar data structure, and has been used to store a document vocabulary in classical applications of natural language processing and information retrieval. In general, such applications do not need to manage very large datasets because of *Heaps' Law* [61]. This is a very precise law which states that the vocabulary of a text of n words is of size $\mathcal{O}(n^{\beta})$, where β depends on the particular text. Value β is usually in the range 0.4–0.6 [14]. This means that the space consumption of the dictionary of a terabyte-size collection would be a few megabytes. The dictionary would easily fit in any main memory.

On the other hand, a number of natural language applications such as Web search engines and machine translation systems are not under Heaps' Law, as mentioned in [26, 95]. Also, other recent applications involving Web graphs, resource description framework (RDF) stores, bioinformatics, NoSQL databases, geographic information systems and so

on have to manage very large datasets through string dictionaries. From that background, many space-efficient string dictionaries using a variety of techniques have been proposed, and many sophisticated implementations have been published as open source libraries.

Although the implementations have different features, we can classify them into two classes: *static* and *dynamic*. Static dictionaries do not allow to insert and delete key strings at any time. In other words, a static dictionary is built from an existing lexicographically-sorted dataset at hand, basically offline. In contrast, dynamic ones allow to insert and delete keys flexibly; therefore, they can be applied to more applications. Concerning the dictionary size, static dictionaries considerably outperform dynamic ones. Actually, most of applications handling very large data are satisfied using static string dictionaries because the data are not frequently changed. Nevertheless, a part of applications as in Web crawler and Semantic Web need to compact dynamic string dictionaries. Consequently, many researchers and engineers have investigated efficient data structures for each class as follows.

For static string dictionaries, the most significant literature would be [26] (or its extended journal version [95]), which mentioned the need of space-efficient string dictionaries and presented some compact data structures. On the other hand, some researchers proposed distinct compact data structures [8, 57, 139]. These can implement string dictionaries in space much smaller than classical data structures, while supporting lookup operations within a few microseconds. For example, Martínez-Prieto et al. [95] showed that their state-of-the-art dictionaries can be implemented in space up to 5% of the raw dataset size through careful experiments using large real-world datasets. However, there remain trade-off problems among space efficiency, lookup-time performance and construction costs. For example, a powerful compression technique for a text, called *Re-Pair* [83], is well used to obtain high space efficiency and fast lookup operations, but the time and space needed during construction are not practical for very large datasets.

For dynamic string dictionaries, there are some space-efficient data structures [10, 54, 84, 130, 149]. While these attempt to improve the space efficiency by reducing pointer overheads, they still consume much larger space than the static dictionaries because some redundancy for dynamic update is needed. Note that, concerning a *trie* [48, 79] that is a major tree structure of constructing string dictionaries, several practical compact dynamic representations have been presented. Especially, Poyias et al. [123, 124] recently proposed compact trie representations in asymptotically information-theoretically optimal space while supporting basic tree operations in constant expected time. However, there has been no discussion or evaluation about string dictionary implementation.

As discussed above, there remain some problems both for the static and dynamic dictionaries, whereas the amount of string data will keep growing. In our work, we addressed

the engineering of more efficient data structures to solve the problems. We sometimes do not provide the theoretical guarantees since our data structures are heuristic; however, we always show their practical performances through careful experiments using large real-world datasets. Furthermore, most of our implementations are provided as open source libraries on Kanda's GitHub page at https://github.com/kampersanda under the MIT License.

1.1 Thesis Organization and Our Contributions

This thesis is structured as follows. In Chapter 2, we first show basic notations used throughout the thesis and summarize tools for implementing our dictionaries. We also introduce applications, definitions, and existing data structures of string dictionaries. We then present our contributions, which are summarized below.

Static Compressed Double-Array Tries In Chapter 3, we propose new string dictionaries based on *double-array tries* [2]. The double array is a data structure well-used to represent a trie because its node-to-node traversal speed is the fastest; however, its space usage is much larger compared to recent compressed data structures because it uses two pointers for each node. Although Yata et al. [142] proposed a compact form of the double array, it still uses one pointer for each node. By solving the problem, we propose more compressed data structures supporting fast lookup operations.

The contents of this chapter are based on [70, 74].

Static Dictionary Compression Using Dictionary Encoding In Chapter 4, we attempt an alternative compression strategy to string dictionaries, instead of an existing strategy using the Re-Pair compression. Re-Pair is a powerful compression tool and can provide fast decoding, but the compression incurs large construction costs in practice although it is theoretically executed in linear time and space over the length of a given text. Although we can choose other lightweight compression techniques such as Huffman coding [66] and online grammar compression [96, 97], Re-Pair is the most popular compression tool at present because of the excellent compression rate. To solve this problem, we apply *dictionary encoding* to string dictionary compression.

The contents of this chapter are based on [71, 73]

Dynamic Path-Decomposed Tries In Chapter 5, we propose a new data structure of space-efficient dynamic string dictionaries, namely a *dynamic path-decomposed trie*. It is based on a trie formed by *path decomposition* [46] that is a trie transformation technique. The path decomposition was proposed to construct cache-friendly trie structures and is

utilized in static applications at present [57, 64], whereas we use it for dynamic dictionary construction with a different approach.

The contents of this chapter are based on [72]

Practical Rearrangement of Dynamic Double-Array Tries In Chapter 6, we investigate practical rearrangement of dynamic double-array trie dictionaries. Although the space efficiency of dynamic double-array trie dictionaries tends to decrease with key updates, we can still maintain high efficiency using existing methods [99, 115, 143]. However, these methods have practical problems of time and functionality. This chapter presents some efficient rearrangement methods to solve these problems.

The contents of this chapter are based on [75]

Chapter 2

String Dictionaries

In this chapter, we first introduce the basic notation used throughout this thesis in Section 2.1. We then describe tools for implementing our data structures in Section 2.2. In Sections 2.3 and 2.4, we show applications and definitions of string dictionaries, respectively. Section 2.5 finally summarizes selected data structures of the string dictionaries.

2.1 Basic Notation

In all cases, our *arrays* start at position zero. An array A of n elements $A[0]A[1]\dots A[n-1]$ is denoted as $A[0,n)$, or $A[0,n-1]$. The length of the array $A[0,n)$ is $|A| = n$. Although we sometimes call arrays *sequences*, there is no difference between those.

A *string* is an array of elements drawn from a finite universe, called the *alphabet*. Alphabets are usually denoted as $\Sigma = \{0, 1, \dots, \sigma - 1\}$, where σ is the *alphabet size*. The alphabet elements are called *characters* or *symbols*. The set of all the strings over Σ is denoted as Σ^*. Given a string $x[0,n)$, a substring from the i-th character to the j-th is denoted by $x[i,j]$ or $x[i,j+1)$. Particular cases of substrings are *prefixes*, of the form $x[0,i)$, and *suffixes*, of the form $x[i,n)$. A special case of particular importance is when Σ is $\{\mathtt{0},\mathtt{1}\}$. In this case, the string is called a *bit string* or a *bit array*.

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 $\lceil 2.4 \rceil = 3$. 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, $(9)_2 = \mathtt{1001}$ and $|(9)_2| = 4$. The base of the logarithm is 2 throughout this paper, i.e., $\log a = \log_2 a$.

2.2 Tools

This section describes major tools for implementing compact data structures, related to our work. Note that we assume that the tools are in static cases, not supporting to dynamically update own structures.

2.2.1 Fully Indexable Dictionaries

A *fully indexable dictionary (FID)* is a key component of many compact data structures. Given a bit array B, the FID supports the following operations:

- $\textsc{Rank}_b(B, i)$ returns the number of $b \in \{\mathtt{0}, \mathtt{1}\}$ bits in $B[0, i)$
- $\textsc{Select}_b(B, i)$ returns the position of the $i + 1$ th occurrence of $b \in \{\mathtt{0}, \mathtt{1}\}$ in B.

Given a bit array $B[0, 8) = \mathtt{00100110}$, the operations are performed as $\textsc{Rank}_{\mathtt{0}}(B, 6) = 4$, $\textsc{Rank}_{\mathtt{1}}(B, 6) = 2$, $\textsc{Select}_{\mathtt{0}}(B, 2) = 3$, and $\textsc{Select}_{\mathtt{1}}(B, 1) = 5$.

Let n be the length of B. The operations can be supported in constant time by adding $o(n)$ bits of redundancy to the bit array in theory [67, 103]. Some practical implementations were proposed. The simplest implementation is to partition the bit array into large and small blocks, and to use two auxiliary arrays keeping Rank hints for each block, which called *verbatim* in [114]. It supports Rank in $\mathcal{O}(1)$ time and Select in $\mathcal{O}(\log n)$ time using additional $o(n)$ bits. Other implementations include cache-conscious ones [53], entropy-compressed ones [114], and so on [76, 77, 126]. If a bit array is the main part of own data structure, choice of an FID implementation is significant; otherwise, the verbatim provides enough performance experientially.

2.2.2 Variable Byte Coding

Variable-length coding is a scheme to compress a sequence consisting of many small integers using variable-length codes with less space. Especially, the scheme is convenient for compressing posting lists of inverted indexes; therefore, a number of techniques have been proposed in the area of information retrieval [151]. Here, we introduce *variable byte (Vbyte) coding* [134] that is a simple and useful coding technique.

Given an integer a, Vbyte encodes it into byte codes as follows. First, the integer a is cut into 7-bit chunks, $a = a_1 a_2 \dots a_k$. Each chunk is then stored in the lowest bits of a byte, whose highest bit is $\mathtt{1}$ for $a_1 \dots a_{k-1}$, and $\mathtt{0}$ for a_k. As the Vbyte coding can compress an integer sequence using the byte codes, the encoding and decoding are simple and fast.

2.2.3 Directly Addressable Codes

A problem with variable-length coding is how to access the code of the i-th integer directly. Brisaboa et al. [25] proposed the *directly addressable codes (DACs)* to solve the problem practically. DACs implement direct extraction by combining Rank with a Vbyte encoding scheme. Suppose that DACs represent an array of integers P. Given a parameter b, we split $(P[i])_2$ into blocks of b bits, $p_{(i,k_i)}, \dots, p_{(i,2)}, p_{(i,1)}$ where $k_i = \lceil |(P[i])_2| / b \rceil$. For example in $P[i] = 49$ and $b = 2$, we split $(49)_2 = \mathtt{110001}$ into $p_{(i,3)} = \mathtt{11}$, $p_{(i,2)} = \mathtt{00}$, and $p_{(i,1)} = \mathtt{01}$.

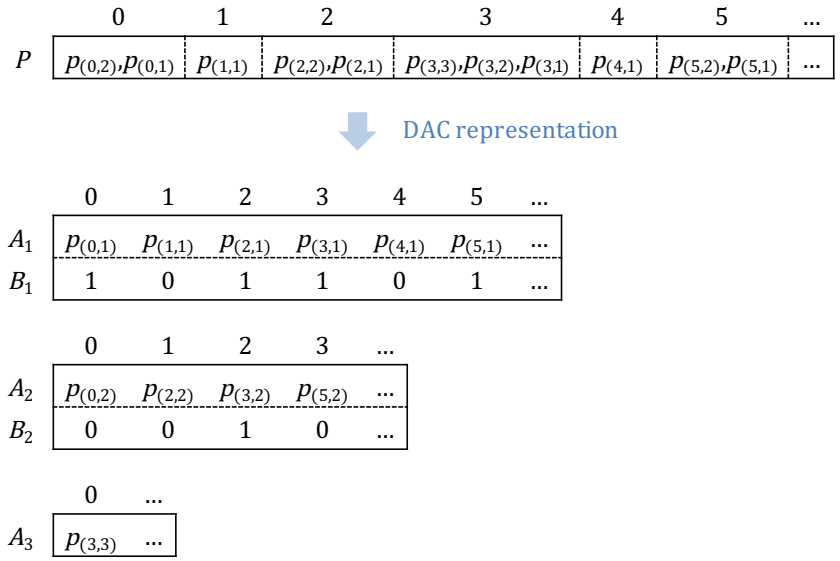


FIGURE 2.1: DAC representation for array P.

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 2.1 shows an example of a DAC representation. Let $i_1, i_2, \ldots, i_{k_i}$ denote the path storing $P[i]$, that is, $A_1[i_1] = p_{(i,1)}$, $A_2[i_2] = p_{(i,2)}$, ..., $A_{k_i}[i_{k_i}] = p_{(i,k_i)}$. 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;

$$i_{j+1} = \text{RANK}(B_j, i_j) \quad (B_j[i_j] = 1). \tag{2.1}$$

From $B_j[i_j] = \texttt{0}$, we can identify that $A_j[i_j]$ stores the last block. For example in FIGURE 2.1, $P[5]$ 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 $\text{RANK}(B_1, 5) = 3$, and we can see that $A_2[3] = p_{(5,2)}$ is the last block from $B_2[3] = \texttt{0}$.

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 $\mathcal{O}(L)$ time in the worst case. In practice, byte-oriented DACs with $b = 8$ are well-used because very fast extraction can be supported.

2.2.4 Grammar-Based Compression

Grammar-Based Compression [28] finds a small context-free grammar that generates the text to compress. Finding the smallest grammar for a given text is NP-hard.

Re-Pair [83] is a practical technique of grammar-based compression [28]. It finds the most frequent pair xy in a text and replaces all its occurrences with a new symbol z. A new rule $z \to xy$ is added to dictionary R. This process iterates until all remaining pairs are unique in the text. As a result, the original text is encoded into a compressed sequence C with dictionary R. Each symbol of C is decoded by recursively expanding the rules in R. The time taken depends on its recursion depth.

2.2.5 Succinct Trees

A tree is an acyclic connected graph and one of the most pervasive data structures. Here, we consider an *ordinal tree* that is a rooted tree where the order of the children of each node is specified. General pointer-based data structures represent a tree of n nodes in $\mathcal{O}(n \log n)$ bits, but this space is much larger than the information-theoretical lower bound as follows. The number of ordinal trees is equal to the *Catalan number* $C_n = \frac{1}{n+1}\binom{2n}{n}$ [55]. It is easy to derive the approximation $C_n \approx 4^n/n^{3/2}$ from the Stirling approximation. Actually, this implies that $\log C_n = 2n - \Theta(\log n)$, that is, representation of an ordinal tree needs at least $\lceil \log C_n \rceil = 2n$ bits.

Succinct representations of ordinal trees use just $2n + o(n)$ bits and can carry out many operations on the tree in constant time. Basically, we can classify succinct tree representations into four types [7]:

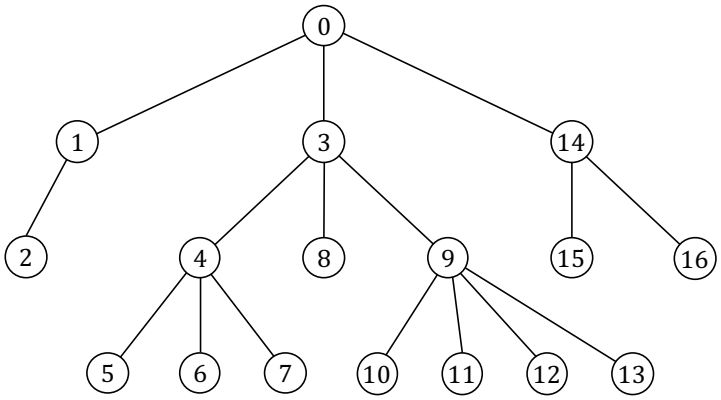
- the level-ordered unary degree sequence (LOUDS) representation [38, 67],
- the balanced parentheses (BP) representation [67, 103],
- the depth-first unary degree sequence (DFUDS) representation [17], and
- the fully-functional succinct tree (FF) representation [31, 108].

Here, we describe LOUDS and DFUDS because they are applied for some space-efficient string dictionary implementations.

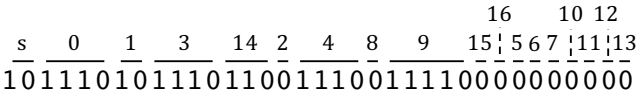
LOUDS It is the simplest succinct representation of ordinal trees. It encodes a node with d children into d `1`s and one `0`, while traversing nodes in breadth-first order. For example, a node with three children is encoded into `1110`. Note that leaves are also considered as nodes with non-children, and the super root `10` is added to the head. The resulting bit array consists of n `0`s and $n+1$ `1`s, as shown in FIGURE 2.2b.

Let us define two basic tree operations: $\textsc{Child}(v, i)$ returns the i-th child of node v, and $\textsc{Parent}(v)$ returns the parent of node v. LOUDS can support these by constructing the FID to the bit array as follows.

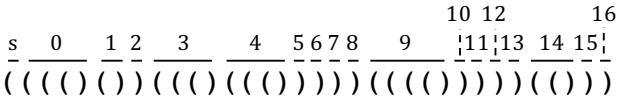
- $\textsc{Child}(v, i) = \textsc{Select}_{\mathtt{0}}(\textsc{Rank}_{\mathtt{1}}(v)) + 1 + i$



(A) Ordinal tree



(B) LOUDS representation



(C) DFUDS representation

FIGURE 2.2: Succinct tree representations.

- $\text{PARENT}(v) = \text{SELECT}_1(\text{RANK}_0(v))$

The operations can be performed in constant time. As the resulting bit array is of length $2n + 1$, the LOUDS representation takes only $2n + o(n)$ bits.

DFUDS It is a succinct representation supporting powerful tree operations. It encodes a tree into a sequence of parentheses. A node with d children is encoded into d `(`s and one `)`, while traversing nodes in depth-first order. For example, a node with three children is encoded into `(((`. Note that leaves are also considered as nodes with non-children, and the super root `(` is added to the head. The resulting parentheses consist of n `(`s and n `)`s, where n denotes the number of nodes. FIGURE 2.2c shows an example of DFUDS for the tree in FIGURE 2.2a.

We first need to define operations on *balanced parentheses (BP)* in order to describe the tree operations of DUFDS. A sequence of BP is inductively defined as follow: an empty sequence is BP; if α and β are sequences of BP, then also $(\alpha)\beta$ is a sequence of BP, where `(` and `)` are called *mates*. Let BP is a sequence of BP, we define the following operations on the sequence:

- FindClose(i), for a value i such that $BP[i] =$ `(`, returns the position $j > i$ such that $BP[j] =$ `)` is its mate.
- FindOpen(i), for a value i such that $BP[i] =$ `)`, returns the position $j < i$ such that $BP[j] =$ `(` is its mate.

In general, those parentheses are represented as bits, where `1` and `0` represent `(` and `)`, respectively. Let the length of BP be $2m$ (i.e., m `(`s and m `)`s), BP can be encoded in $2m + o(m)$ bits, while supporting the above operations in constant time [67, 103, 108].

Reconsidering the DFUDS representation, the sequence of parentheses is balanced; therefore, it can be encoded in $2n + o(n)$ optimal bits, while supporting the operations for parentheses in constant time. DFUDS supports the basic tree operations using the balanced parentheses as follows:

- $\text{Child}(v, i) = \text{FindClose}(\text{Select}_{)}(\text{Rank}_{)}(v) + 1) - i) + 1$
- $\text{Parent}(v) = \text{Select}_{)}(\text{Rank}_{)}(\text{FindOpen}(v - 1))) + 1$

2.3 Applications

Traditional applications of string dictionaries are in natural language processing and information retrieval. String dictionaries have been an important component to store natural language words as in full-text inverted indexes [14] and morphological analyzers [80]. As described in Chapter 1, Heaps' Law [61] states that the vocabulary of a text of n words is of size $\mathcal{O}(n^{\beta})$, where β depends on the particular text. Some experiments [5, 13] on the TREC-2 collection have shown that the most common values for β are between 0.4 and 0.6. This means that the space consumption of the dictionary of a terabyte-size collection would be a few megabytes. When using modern computers, we will be able to easily handle such dictionaries in main memory even if naïve data structures are used. On the other hand, as mentioned in [26, 95], there are many recent applications handling very large string dictionaries without according to Heaps' Law, as follows.

In natural language applications, Web collections and N-grams are well-known large datasets. Web collections are much less *clean* than text collections whose content quality is carefully controlled. As such collections include typos, unique identifier and regular words, dictionaries as in Web search engines easily exceed gigabytes. N-grams are widely adopted for many tasks such as auto-completion in search engines, similarity search, automatic speech recognition, and machine translation [33, 90]. The datasets are at the heart of statistical natural language processing, and the amount is deeply related to the performances of systems. Actually, efficient data structures for N-gram language models have been developed apart from the topics in this thesis [60, 111, 118, 120]. As they are

suitable for storing N-grams, both the time and space performances of them are outperform those of compressed string dictionaries if implementing N-gram language models. However, the data structures for N-grams support only a simple lookup operation for a query string. A part of applications such as input method editors [81] need string dictionary implementation supporting rich operations such as prefix-based and reverse lookup operations.

Web graphs are coherent subsets of the World Wide Web and are used for many tasks such as Web mining, Web spam detection, and finding communities of interest [39, 78, 128]. The Web graph is a directed graph whose nodes are Web pages and edges are hyperlinks among them. As the size of graph representation was the significant problem, some compressed representations were proposed in the last 15 years [24, 29]. Due to the works, we can represent the nodes and edges in very compact space, whereas the size of URL names becomes relatively problematic; therefore, space-efficient string dictionaries for storing URLs are required. Related to the area, Web crawlers [22, 132] keep track of massive URLs it has visited.

Other important applications arise in Semantic Web, or so-called Web of Data [18]. It interconnects RDF datasets from diverse fields of knowledge into a cloud of data-to-data hyperlinks. RDF [91] is a framework to describe attributes and relation of resources, which consist of the terms: URIs, blank nodes, and literal values. As the Web of Data grows in popularity, larger datasets emerge and a number of efficient management systems of RDF datasets, or RDF stores, have been developed [109, 136]. In those systems, the terms of RDF datasets are compactly stored by encoding them into integers using string dictionaries. A recent paper [94] reported the significant of the performance of RDF dictionaries.

The other applications arise as in NoSQL, bioinformatics, internet routing and geographic information systems. The descriptions are provided by [95]. Actually, most of applications described above are satisfied using static string dictionaries because the data are not frequently changed. Nevertheless, a part of applications need to dynamic ones, such as vocabulary accumulation [62], Web crawlers [22, 132] and dipLODocus$_{[\mathrm{RDF}]}$ [136].

2.4 Definitions

A *string dictionary* is a data structure that stores a set of strings $\mathcal{X} \subset \Sigma^*$. This section defines static and dynamic string dictionaries used throughout this thesis.

2.4.1 Static Dictionaries

A *static string dictionary* is a data structure that provides a bijection between strings in $\mathcal{X}$ and unique integer IDs between 0 and $|\mathcal{X}| - 1$. It supports the following operations:

- Lookup(x) returns the ID of string x if $x \in \mathcal{X}$; otherwise returns -1.
- Access(i) returns the string corresponding to the ID $i \in [0, |\mathcal{X}|)$.
- PrefixLookup(x) returns the set of strings in $\mathcal{X}$ included as prefixes of string x.
- PredictiveLookup(x) returns the set of strings in $\mathcal{X}$ starting with string x.
- SubstringLookup(x) returns the set of strings in $\mathcal{X}$ containing string x.

Lookup is the most basic and minimum required operation. Depending on design of string dictionaries, several other definitions can be considered such as reporting only the membership and obtaining the associated values; however, we define Lookup as the pair operation of Access to support *dictionary encoding* that replaces strings into the integer IDs. The dictionary encoding is essential for database systems, RDF stores, Web graphs and so on. Nevertheless, we can easily implement the design storing key-value pairs by introducing a satellite array of size $|\mathcal{X}|$. Needless to say, Access is the inverse operation of Lookup.

The other operations are additional but required from particular applications. The prefix-based operations, PrefixLookup and PredictiveLookup, are useful for natural language applications such as input method editors [81], morphological analyzers [80], stemmed searches [14] and auto-completions [16]. The substring-based operation SubstringLookup is required, for example, in SPARQL regex queries [125], mainly used for full-text purposes in Semantic Web applications [50].

2.4.2 Dynamic Dictionaries

A *dynamic string dictionary* is a data structure that associates arbitrary values to each string $x \in \mathcal{X}$. It supports the following operations:

- Search(x) returns the associated value of string x if $x \in \mathcal{X}$; otherwise, returns nil.
- Insert(x, v) registers the string x to the dictionary, or $\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$, and associates value v to the string.
- Delete(x) removes string x from the dictionary, or $\mathcal{X} \leftarrow \mathcal{X} \setminus \{x\}$.
- PrefixSearch(x) returns the set of strings in $\mathcal{X}$ included as prefixes of string x.
- PredictiveSearch(x) returns the set of strings in $\mathcal{X}$ starting with string x.

We are not aware of any dynamic dictionary implementation to provide the bijection and substring-based operations. Therefore, we define the dynamic string dictionary storing key-value pairs. Search and Insert are basic and required at least. Delete is also

important but is not required in applications only constructing dictionaries online, such as the above-described ones. PrefixSearch and PredictiveSearch are used in some search engines, database systems, and so on, in the same manner as the static case.

2.5 Data Structures

This section summarizes data structures for implementing string dictionaries, while categorizing them based on conceptualistic techniques. We discuss leading data structures we think and do not discuss traditional ones such as linked lists and binary pointer trees. Detailed descriptions of string dictionaries involving traditional data structures are in [9], although state-of-the-art dictionaries are not included because [9] is a little bit old thesis.

2.5.1 Hash Tables

Hashing [32] is a folklore method to implement dictionary structures with any types. Hashing string dictionaries store strings in a *hash table* by generating an element index from a string using a *hash function* and by placing the string into the element. If multiple strings are hashed to the same location, some form of collision resolution is needed. Since the origin of hashing was proposed by H.P. Luhn in 1953 [79], many methods have been proposed for resolving collisions. Basically, there are two classes: *closed and open hashing*. Here, we first describe how to implement static string dictionaries using closed hashing, and then describe how to implement dynamic ones using open hashing.

We denote a closed hash table of length m as H. Closed hashing (also called *open addressing*) [119] stores every object (or string in our case) directly at an address of H and resolves collisions by searching for another empty element through some probing scheme such as *linear probing* and *double hashing*. Linear probing sequentially searches another empty element from the collision position until finding it. Double hashing computes another hash function and searches another empty element by skipping elements using the result. The space usage and time performance depends on its load factor $\alpha = n/m$, where n is the number of registered objects. Using good hash functions, unsuccessful and successful searches respectively require on average $\frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$ and $\frac{1}{2}(1 + \frac{1}{1-\alpha})$ probes with linear probing. With double hashing, unsuccessful and successful searches require on average $(1 - \alpha)^{-1}$ and $-\alpha^{-1} \ln(1 - \alpha)$ probes, respectively. Although value α takes a trade-off between space and time, the search can be performed in constant expected time if setting a proper α, for example, $\alpha < 0.9$ in the double hashing [79].

Static string dictionaries are implemented by computing hash functions with key strings and by storing the strings in H. However, as H includes some empty elements, the addresses of H cannot simply provide the mapping between strings in $\mathcal{X}$ to IDs in the range

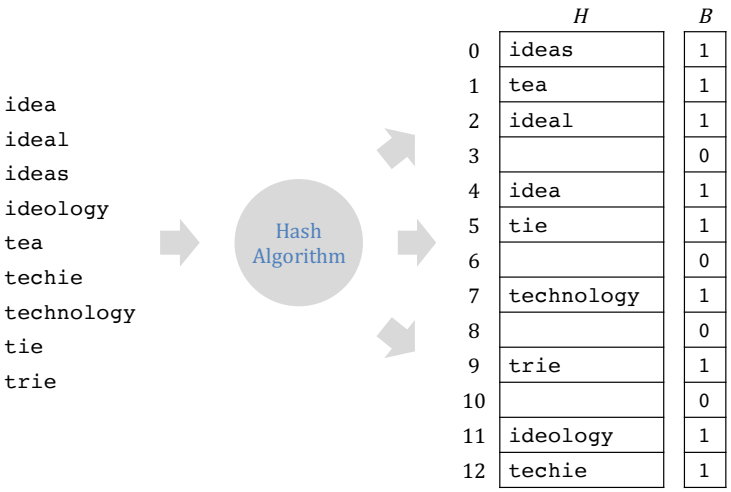


FIGURE 2.3: Closed hashing dictionary.

$[0, |\mathcal{X}|)$. To solve this problem, we introduce a bit array B such that $B[i] = \texttt{1}$ iff $H[i]$ is nonempty, and constructs the FID to B. LOOKUP(x) searches address i such that $H[i] = x$ through the hash algorithm and returns $\text{RANK}_1(B, i)$. ACCESS(i) is simply $H[\text{SELECT}_1(B, i)]$.

Example 2.1. FIGURE 2.3 *shows an example of a closed hashing dictionaries for strings* `idea`, `ideal`, `ideas`, `ideology`, `tea`, `techie`, `technology`, `tie` *and* `trie`.[1] *The strings are distributed in the hash table at random via a hash algorithm. The string IDs are also defined according to their addresses. The load factor is* $\alpha = 9/13 = 0.69$. LOOKUP(`idea`) *obtains the target address 4 via the hash algorithm and returns the ID as* $\text{RANK}_1(B, 4) = 3$. ACCESS(3) *is simply* $H[\text{SELECT}_1(B, 3)] = H[4] =$ `idea`.

Closed hashing can dynamically insert objects to the hash table, but the performance rapidly declines as the load factor approaches 1, and we cannot easily resize the hash table. On the other hand, open hashing (also called *separate chaining*) can insert objects to the hash table without such restriction; therefore, it is often used to implement standard hash maps at present, such as `std::unordered_map` in C++. Here, we describe how to implement dynamic string dictionaries using open hashing.

In common with the closed hash table, we denote an open hash table as H of length m. Each element stores a pointer to the beginning node of a linked list, where each node in the list has a registered string, the associated value, and a pointer to the next node. Specifically, a pointer to the string is stored as the length is variable. SEARCH(x) obtains the address of H by computing a hash function with string x, and search the target node in the linked list. INSERT and DELETE are also implemented in the same manner.

[1] In this section, we basically show examples of string dictionaries using the set of strings.

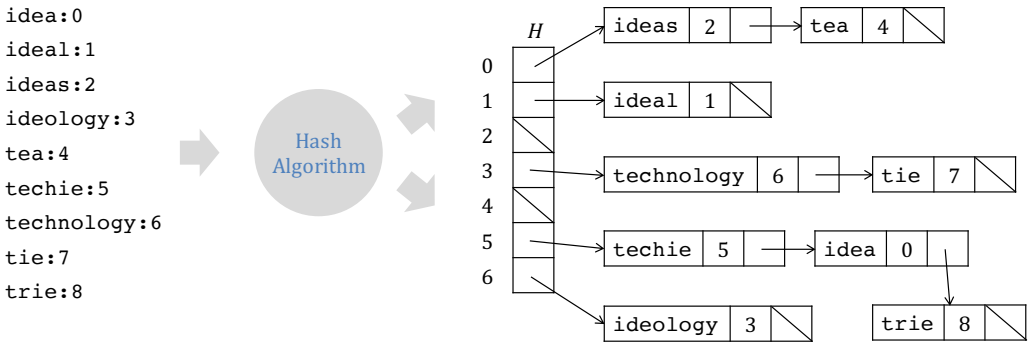


FIGURE 2.4: Open hashing dictionary.

Example 2.2. FIGURE 2.4 *shows an example of an open hashing dictionary. In common with* FIGURE 2.3*, the strings are distributed at random via a hash algorithm and stored to the linked lists.* SEARCH(`trie`) *obtains the target address 5 via the hash algorithm and the head of the linked list. By traversing the nodes with comparing the strings, the value 8 stored in the target node is returned.*

HashDAC Martínez-Prieto et al. [95] proposed static compressed dictionaries based on closed hashing, namely *HashDAC*. It distributes strings to the hash table H in the same manner as described above, but does not store them in H. Instead, a new table H' of length n is used such that $H'[\text{RANK}_1(B, i)] = H[i]$, where n denotes the number of strings. H' is a string array whose indices correspond to string IDs, and is represented in compact space by compressing the strings through Re-Pair and by eliminating the string pointers through DACs.

Array Hash The bottleneck of standard open hashing dictionaries, as shown in FIGURE 2.4, is that a node of the linked lists has two pointers of a string and next node. The pointers incur not just space overheads but also cache misses. Askitis and Zobel [12] designed cache-conscious dynamic dictionaries by eliminating the pointers, namely *array hash*. It concatenates the strings in a linked list and sequentially searches them. Although the time complexities of the operations are increased, the practical performances are improved due to the cache efficiency. Also, the space usage can be improved because the pointers are eliminated.

2.5.2 Front Coding

Front Coding [135] is a folklore technique to compress lexicographically sorted strings. It encodes each string as a pair (ℓ, α), where ℓ is the length of the longest common prefix with its predecessor and α is the remaining suffix. This technique exploits the fact that

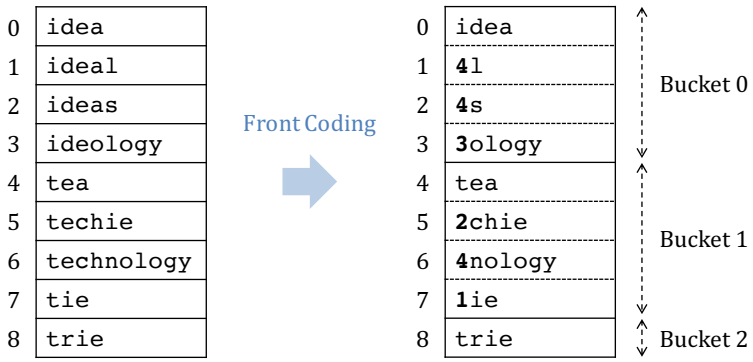


FIGURE 2.5: Front Coding dictionary ($b = 4$).

real-world strings have similar prefixes such as URLs and natural language words. For example, it was used to compress the set of URLs in the Web Graph framework [21].

This technique is used only to implement static dictionaries because the strings must be sorted. To allow for direct access, the strings are divided into buckets encoding b strings each. Each first string (referred to as the *header*) is explicitly stored and the remaining $b - 1$ strings (referred to as *internal strings*) are differentially encoded, each with respect to its predecessor.

In the directly accessible Front Coding dictionary, LOOKUP(x) can be performed in two steps. The first step consists of a binary search for string x in the set of headers to obtain the target bucket. The second step sequentially decodes the internal strings of the bucket while comparing each with x. ACCESS(i) is performed in two steps as well. The first step calculates the appropriate bucket ID with a simple division $\lfloor i/b \rfloor$. The second step sequentially decodes the internal strings of the bucket until it obtains the (i mod b)-th internal string. LOOKUP(x) and ACCESS(i) $= x$ take $\mathcal{O}(|x|(\log \frac{n}{b} + b))$ and $\mathcal{O}(|x|b)$ times, respectively. Also, PREDICTIVELOOKUP can be supported by searching the target range of sorted strings.

Example 2.3. FIGURE 2.5 *shows an example of the Front Coding dictionary of bucket size* $\ell = 4$. *As shown in this figure, the string* `ideal` *is differentially encoded into pair* $(4, \texttt{l})$ *from the previous string* `idea`. *Also, the string* `ideas` *is differentially encoded into pair* $(4, \texttt{s})$ *from the previous string* `ideal`, *and so on. However,* `idea`, `tea` *and* `trie` *are not encoded because they are headers for each bucket.*

In the example, ACCESS(6) *is performed as follows. We get the target bucket ID as* $\lfloor 6/4 \rfloor = 1$. *We also get the offset of the target string in bucket 1 as* $6 \bmod 4 = 2$. *Hence, we decode strings in bucket 1 until the second internal string. The first internal string is decoded by concatenating the prefix of length 2 of the header* `tea` *and the remaining suffix* `chie`. *The result is* `techie`. *Next, the second internal string is decoded by concatenating*

the prefix of length 4 of the previous string `techie` *and the remaining suffix* `nology`*. The result is* `technology`*. Consequently,* Access(6) = `technology`*.*

Lookup(`technology`) *is performed as follows. We first obtain the bucket that may store the query string, through binary search for each header. In this case, the bucket ID is 1 because* `tea` *is less than* `technology` *and* `technology` *is less than* `trie` *lexicographically. Next, we search the string by decoding the internal strings in the same manner as* Access*. As a result,* Lookup(`technology`) = 6*.*

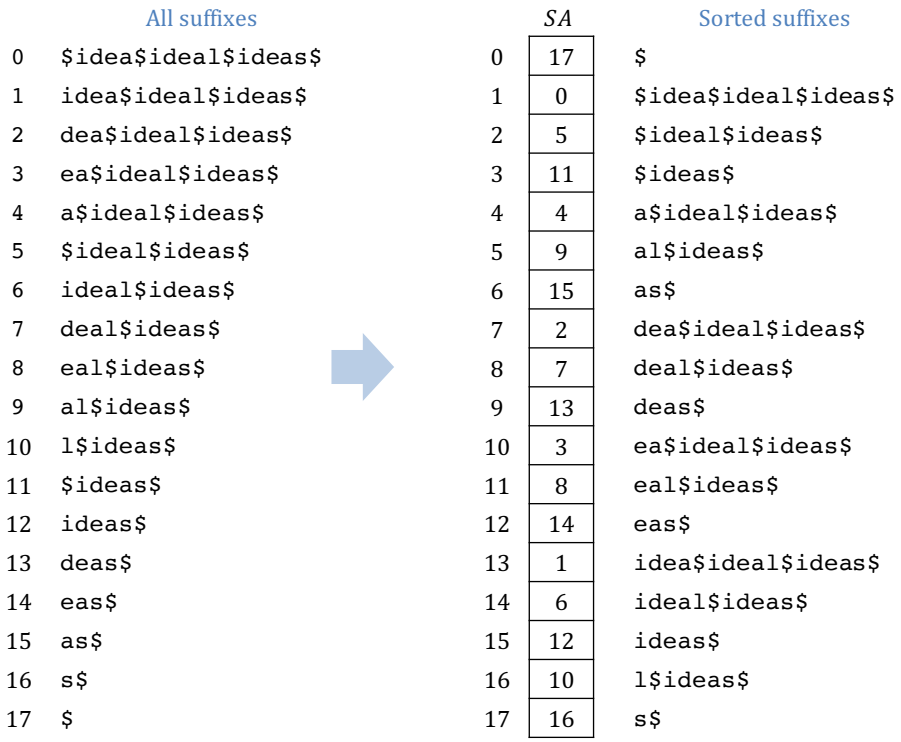
Plain Front Coding (PFC) It is a very simple byte-oriented Front Coding implementation [95]. It encodes the Front Coding dictionary into a byte sequence by concatenating the shared lengths and remaining strings. Note that the length is encoded using Vbyte and a special terminator is appended to each string. The beginning positions of each header are kept in a pointer array of size $\lceil n/b \rceil$. As the byte data other than the headers are always accessed sequentially, we can compress them using an arbitrary text compression technique such as Re-Pair or Huffman coding [66].

Hu-Tucker Front Coding (HTFC) It is a more compact implementation at the price of slightly slower operations [95]. It compresses the bucket headers, not compressed in PFC, using the Hu-Tucker coding [65, 79]. The Hu-Tucker coding assigns optimal prefix codes like Huffman coding, while retaining the lexicographic order of the symbols; therefore, two Hu-Tucker encoded sequences can be lexicographically compared. This feature enables the binary search for the headers in compressed form.

2.5.3 Full-Text Index

Given a text T, a *full-text index* is a data structure that fast reports all the positions where pattern p occurs in T. A *suffix array* [89] is one of efficient data structures for the full-text index. We consider all the N suffixes of a text T of length N. The suffix array $SA[0, N)$ of T contains all the starting positions of the suffixes of T listed in lexicographic order. All substrings in T appear as prefixes of successive suffixes in SA; therefore, their appearance positions correspond to ranges $SA[sp, ep]$. The limits sp and ep can be found with two binary searches in $\mathcal{O}(|p| \log N)$ time [89].

Given a set of strings $\mathcal{X} = \{x_0, x_1, \ldots, x_{n-1}\}$, we can implement static dictionaries by constructing the suffix array over a text $T = \$x_0\$x_1\$ \ldots \$x_{n-1}\$$ of length N, where \$ is a special character that is lexicographically smaller than any characters. The suffix array SA is then as follows: $SA[0] = N - 1$ always holds since $T[N-1] = \$$, and $SA[i+1]$ indicates the suffix $\$x_i\$x_{i+1}\$ \ldots$ for all $0 \leq i < n$. By using the suffix array, the basic operations are performed as follows. Lookup(x) can be carried out by finding the range $SA[sp, ep]$



	All suffixes
0	$idea$ideal$ideas$
1	idea$ideal$ideas$
2	dea$ideal$ideas$
3	ea$ideal$ideas$
4	a$ideal$ideas$
5	$ideal$ideas$
6	ideal$ideas$
7	deal$ideas$
8	eal$ideas$
9	al$ideas$
10	l$ideas$
11	$ideas$
12	ideas$
13	deas$
14	eas$
15	as$
16	s$
17	$

	SA
0	17
1	0
2	5
3	11
4	4
5	9
6	15
7	2
8	7
9	13
10	3
11	8
12	14
13	1
14	6
15	12
16	10
17	16

	Sorted suffixes
0	$
1	$idea$ideal$ideas$
2	$ideal$ideas$
3	$ideas$
4	a$ideal$ideas$
5	al$ideas$
6	as$
7	dea$ideal$ideas$
8	deal$ideas$
9	deas$
10	ea$ideal$ideas$
11	eal$ideas$
12	eas$
13	idea$ideal$ideas$
14	ideal$ideas$
15	ideas$
16	l$ideas$
17	s$

FIGURE 2.6: Full-text dictionary using a suffix array.

over the pattern $\$x\$$ for the range $SA[1,n]$. If x is registered, $sp = ep$ essentially and its ID is $sp - 1$. ACCESS$(i) = T[SA[i+1]+1, SA[i+2])$ simply.

The main advantage of full-text dictionaries is to support the substring-based operations. By using the suffix array, SUBSTRINGLOOKUP(x) is performed as follows. We first find the range $SA[sp, ep]$ for pattern x. The results will indicate all the occurrences of x within any string of $\mathcal{X}$; therefore, we extract the corresponding strings for each occurrence. Let $k = SA[i]$ for $sp \leq i \leq ep$, $T[k]$ is the head character of x inside some string in $\mathcal{X}$. We then extract the area $T[j,l]$ such that $j \leq k \leq l$, $T[j-1] = \$$, $T[l+1] = \$$, and $\$ \notin T[j,l]$, by scanning characters from the position k until encountering $\$$. Essentially, PREDICTIVELOOKUP(x) is also supported by search for $\$x$ in the same manner as SUBSTRINGLOOKUP.

The suffix array can implement string dictionaries supporting powerful substring operations, but the space usage is very large because of maintaining the original text T in $N\lceil \log \sigma \rceil$ bits and suffix array SA in $N\lceil \log N \rceil$ bits.

Example 2.4. FIGURE 2.6 *shows an example of the suffix array for strings* `idea`, `ideal` *and* `ideas`, *that is, a text* $T[0,N) =$ `$idea$ideal$ideas$`, $|T| = 18$. $SA[1] = 0$ *points to the suffix* $T[0..] =$ `$idea$...`, $SA[2] = 5$ *points to the suffix* $T[5..] =$ `$ideal$...`, *and* $SA[3] = 11$ *points to the suffix* $T[11..] =$ `$ideas$...`. *The dictionary operations are performed as follows.* ACCESS$(1) = T[SA[2]+1..SA[3]) = T[6,11) =$ `ideal`. LOOKUP(`ideal`)

searches the string for a range from $SA[1]$ *to* $SA[3]$ *with binary search, while restoring each registered string with* Access.

FM-Index The *FM-index* [44, 45] is a practical technique to represent the suffix array in compressed space. The FM-index exploits the behavior of the LF-mapping obtained from the Burrows-Wheeler transform [27]. It can compute *sp* and *ep* for string x in $\mathcal{O}(|x| \log \sigma)$ time, and extract $x \in \mathcal{X}$ in $\mathcal{O}(|x| \log \sigma)$ time also. Martínez-Prieto et al. [95] presented two variants of the FM-Index: faster and smaller versions. The faster version is so-called *succinct suffix array* [45], which achieves zero-order compression of a text. The smaller version uses the *implicit compression boosting* idea [88], which achieves high-order compression.

2.5.4 Tries

A *trie* [37, 48, 79] is an edge-labeled tree structure for storing a set of strings. It is built by merging prefixes and giving a character to each edge. Strings are registered on the root-to-leaf paths. In other words, a query string is searched by traversing root-to-leaf nodes using the characters. Note that when a string $x \in \mathcal{X}$ is the prefix of another one $y \in (\mathcal{X} \setminus \{x\})$, i.e., $x = y[0, |x|)$, the terminal of string x corresponds to an internal node (not a leaf). Therefore, we need some way to identify the nodes corresponding to string terminals. In the thesis, such nodes are called *terminal nodes*. Basically, there are two ways to identify terminal nodes: *using terminal flags* and *using terminal characters*.

The former way introduces a bit array B where $B[s] = 1$ iff node s is terminal. Needless to say, we can identify terminal nodes by checking the bit flag in B. Moreover, it can efficiently support the Lookup/Access operations, because a bijection between terminal nodes and string IDs can be implemented by constructing the FID to B; therefore, this way is often used to implement the static dictionaries. The basic operations are carried out as follows. Lookup(x) traverses nodes from the root using each character of x and returns the string ID obtained from $\textsc{Rank}_1(B, s)$ for the reached terminal node s. In contrast, Access(i) traverses nodes from the terminal node obtained from $\textsc{Select}_1(B, i)$ and returns concatenation of the edge labels on the path. Assuming that Rank and Select are performed in constant time, both Lookup/Access are performed in $\mathcal{O}(k)$ optimal time, where k is the length of the target string.

Example 2.5. Figure 2.7 *shows an example of a trie using terminal flags (although we do not depict the bit array because the node IDs are not assigned). The number of nodes is 30. The nodes depicted as a square are terminal ones and denote the string IDs assigned in the preorder.* Lookup(`techie`) *traverses the colored nodes using each character and returns the string ID 5. In contrast,* Access(5) *traverses the colored nodes in reverse and returns* `techie` *by concatenating the edge characters on the path.*

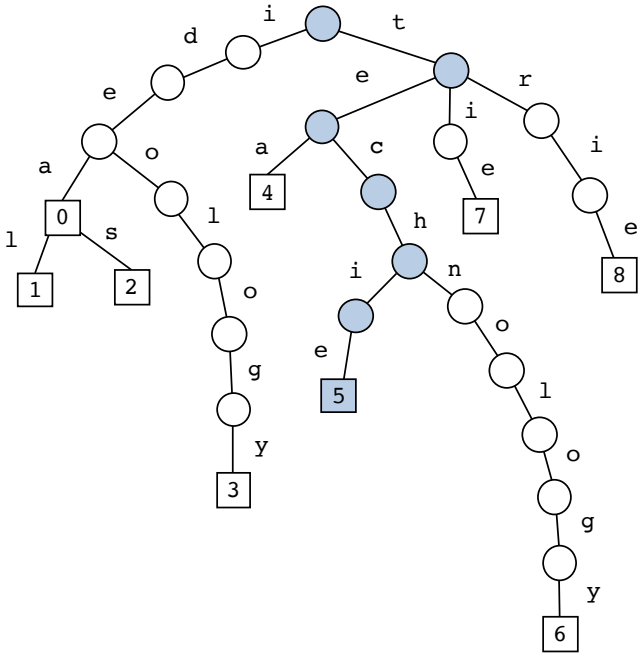


FIGURE 2.7: Trie using terminal flags.

The latter way appends a special terminator `$` to each string in order that each leaf corresponds to each terminal of strings. In other words, we can check the membership if we can arrive at a leaf using a string with the special terminator. Compared to using terminal flags, the number of nodes is larger by the number of strings because of the additional leaf nodes, instead the bit array is not needed. Dynamic string dictionaries are often implemented by this way because such a trie can store arbitrary values to the leaves instead of pointers to its children. When values are stored in the leaves, the operations are carried out as follows. Given a query string x with the special terminator, or $x[|x|-1] = \$$, SEARCH(x) traverses nodes until the leaf in the same manner as LOOKUP and returns the value stored in the leaf. The operation can be supported in $\mathcal{O}(|x|)$ time. INSERT(x, v) initially searches the string x using SEARCH(x). If the search fails, it adds nodes with the remaining suffix from the failed node and stores the value v to the new leaf. DELETE(x) also initially searches the string x using SEARCH(x). If the search succeeds, it removes the nodes corresponding only to the string from the leaf. The operations can also be supported in $\mathcal{O}(|x|)$ time if node addition and deletion is performed in constant time.

Example 2.6. FIGURE 2.8 *shows an example of the trie using terminal characters. The number of nodes is 39 and nine more than the number of nodes in* FIGURE 2.7*, because of the nine registered strings. The nodes depicted as a square are leaves and denote the values stored; thus, the leaf corresponding to string* `techie$` *stores value 5.* SEARCH(`techie$`) *traverses the colored nodes using each character and returns the value 5.* INSERT(`teche$`, 9) *first traverses the nodes until the red one with the prefix* `tech` *and then appends new nodes*

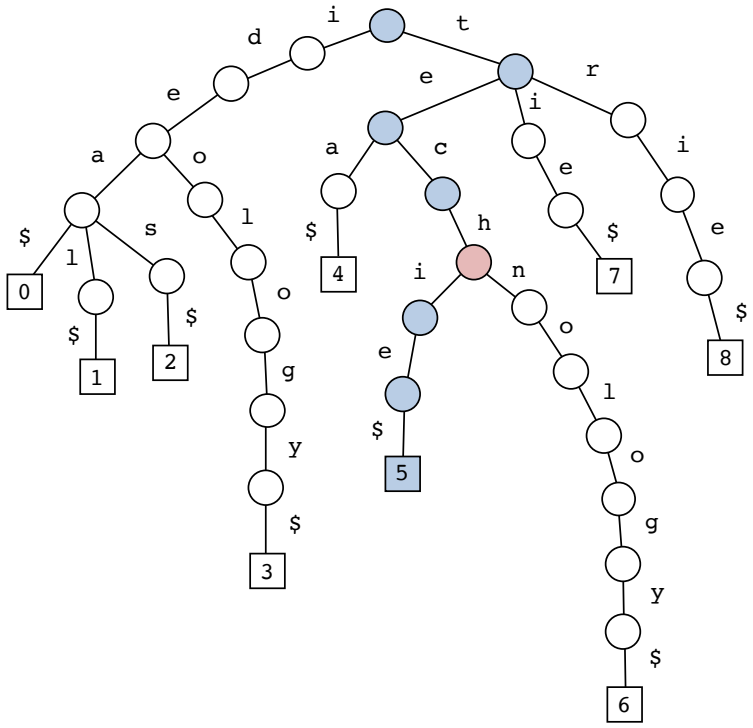


FIGURE 2.8: Trie using terminal characters.

from the red node with the remaining suffix `e$`*. Finally, the value 9 is stored to the new leaf.* DELETE(`techie$`) *first traverses the colored nodes in the same manner as* SEARCH*, and then removes the blue nodes from the reached leaf to the red node in reverse traversal.*

The feature of the trie is to support prefix-based operations such as PREFIXLOOKUP and PREDICTIVELOOKUP because the prefixes of strings are merged as nodes. The operations in static dictionaries are carried out as follows. PREFIXLOOKUP(x) detects all the terminal nodes we met while traversing nodes to search the string x, and then returns the prefixes corresponding to the terminal nodes. PREDICTIVELOOKUP(x) initially searches the string x. If the string is not registered, it returns an empty set; otherwise, it enumerates all the terminal nodes in the subtrie of the terminal node of the string x, and then returns the corresponding strings. In dynamic string dictionaries, PREFIXSEARCH and PREDICTIVESEARCH are also performed in the same manner as these.

Compact Trie Variants For trie compression, there is the important fact that nodes near leaves are likely to be sparse, as shown in FIGURE 2.7, because the suffixes are not shared. One of solutions to reduce the redundancy merges the suffixes backward also. The resulting structure is not a tree but a graph, generally called a *directed acyclic word graph (DAWG)* or an *minimal acyclic deterministic finite automaton (MADFA)* [34]. The DAWG can store strings in compact space, but the one-to-one mapping between the strings

and terminal nodes in tries is lost; therefore, the dictionary implementations as described above can not be adopted. Although we can implement a bijection between strings and IDs by using a *numbered automaton* [86], additional integer data must be attached to each edges. Here, we describe three trie variants with maintaining the one-to-one mapping as follows.

- A *minimal-prefix trie (MP-trie)* [3, 41] maintains only minimal prefixes to identify each string as nodes and the remaining suffixes as separated strings. The leaves have the links to own separated strings instead pointers to children. The MP-trie can simply reduce the number of node pointers.
- A *double trie* [4] is based on the MP-trie, which constructs another trie from the separated strings backward. By using the two tries, it can merge both prefixes and suffixes while maintaining the mapping.
- A *patricia trie* [52, 100] is a form constructed by eliminating one-way nodes and attaching string labels to edges. Especially, it can implement compact dictionaries for long strings because many one-way nodes are included. As the edge labels are variable, its efficient implementation is more difficult than the other variants.

Array and List Tries We introduce two traditional trie representations called the *array and list tries* before sophisticated data structures are described below. The array trie represents a node using an array of $\sigma + 1$ pointers to its σ children and one parent. Let A_s is the array of node s, $A_s[c]$ stores the pointer to the child indicated with character c; therefore, the child can be directly found in $\mathcal{O}(1)$ time. However, the arrays for a trie are very sparse and its space usage is $\mathcal{O}(m\sigma \log m)$ bits.

The list trie (a.k.a. binary search trie) represents a node using two pointers to its first child and next sibling, and a character between the node and its parent. The list trie can represent a trie without useless space unlike the array trie. Its space usage is $\mathcal{O}(m(\log m + \log \sigma))$ bits, but finding a child takes $\mathcal{O}(\sigma)$ time.

Double Arrays A *double array* [2] is a trie representation using two pointer arrays, as the name suggests. It has both the advantages of the array and list tries by sophisticatedly arranging node addresses such that a few empty elements arise in the arrays, while supporting constant node-to-node traversal. The retrieval speed is the fastest and the space usage is practical occupying $\Omega(m \log m)$ bits.

Since the original double array was proposed to implement semi-dynamic trie dictionaries, its update speed is slow and unstable; however, a number of works [99, 115, 143, 147] attempted to solve this problem. Owing to these, recent dynamic implementations such

as Cedar [148] enables update times that are close to those of a hashing dictionary, such as the C++ standard library's `std::unordered_map`. Therefore, there have been some applications using dynamic double-array tries such as a full-text search engine and online text-stream processing [149].

As for static applications, Yata et al. [142] proposed a compact form of double arrays, namely the *compact double array (CDA)*. It is composed of one pointer array and one character array, that is, space bottlenecks of the original pointer arrays can be reduced. On the other hand, the disadvantage of CDA is that ACCESS is impossible due to the pointer elimination; therefore, its applications are limited. Fuketa et al. [49] proposed the *single array with multi code (SAMC)* that eliminates the remaining pointer array in CDA, instead of using additional code tables. However, its applications are more limited for fixed length keywords such as zip codes.

LOUDS Tries The simplest approach to implement tries in optimal space is use of the succinct trees. By using an additional character array storing edge labels, all the succinct trees can represent a trie in $2m + o(m) + m\lceil \log \sigma \rceil$ bits. Although these have the same space efficiency, LOUDS is well used for trie dictionary implementation because of the simpleness and adequateness [7]. For example, *TX-trie* [112] is a simple LOUDS trie library. This library has been well used as a basic succinct trie implementation in many academic settings. *UX-trie* [113] is an enhanced version of TX-trie, which is a more compact implementation applying the double trie. UX-trie achieves high space efficiency by representing the two tries using LOUDS. Kudo et al. [81] used LOUDS for dictionary representation in Google input method editors. MARISA-trie described below also applies LOUDS for trie representation.

MARISA Yata [139] proposed a space-efficient trie dictionary called *MARISA (Matching Algorithm with Recursively Implemented StorAge)*, which enhances the idea of the double trie by adapting it to the patricia trie rather than the MP-trie. In other words, MARISA stores string edge labels of a patricia trie into another patricia trie in the same manner as the double trie; however, MARISA repeats the trie compression recursively. Depending on the depth of the recursion, the space efficiency can be increased. MARISA-trie [140] applies LOUDS for trie representation.

XBW *XBW* [47] is an extension of the FM-index to represent a trie instead of a text. It can support substring-based operations in spite of a trie structure. The space usage achieves the optimal bits in the same as succinct tree representations. Recently, Manzini [92] proposed an efficient construction algorithm of XBW.

Path-Decomposed Tries *Path decomposition* [46] is a technique that transforms a trie. It converts a path in the original trie into a single node and recurse in each subtree hanging down from the path. The number of random accesses during retrieval is reduced compared to the original trie as the height of the resulting tree is suppressed. By applying the *centroid path decomposition* [46], the height is bounded by $\mathcal{O}(\log |\mathcal{X}|)$ and the average lookup time over a string is bounded by $\mathcal{O}(\frac{\log |\mathcal{X}|}{\log \sigma})$. Grossi and Ottaviano [57] proposed practical succinct representation of path-decomposed tries using DFUDS, as described in Chapter 4.

LZ-Compressed String Dictionaries Arz and Fischer [8] proposed a new data structure based on the LZ78 compression algorithm [150], which implements compressed static string dictionaries. The data structure adopts the Lookup and Access operations to the resulting LZ-trie for a text created by concatenating strigns, while introducing another new trie named the *phrase trie*. The LZ-trie can be implemented using any data structures as long as Access and longest prefix search are supported. The phrase trie with a large alphabet is represented using an especial data structure presented in [8].

HAT-Tries The *HAT-trie* [10, 11] is one of efficient dynamic trie implementations, which is an enhanced version of *burst tries* [62]. The burst trie is a dynamic data structure that the number of trie nodes of an array trie is reduced at little cost, by collapsing trie-chains into dynamic containers that share a common prefix. The containers are represented as any associative arrays with string key. Askitis and Sinha engineered cache- and space-efficient HAT-tries by applying the array hash to the container representation.

Judy *Judy* is a trie-based dynamic data structure developed by Doug Baskins at Hewlett-Packard Research labs [15, 130]. It is carefully designed to leverage a CPU cache in modern machines as much as possible. Judy dynamically changes the structural representation of its nodes according to the number of data items they store.

Adaptive Radix Tree The *adaptive radix tree (ART)* [84] is similar to Judy in that the structural representation adaptively changes. It is based on the array trie but uses small arrays if the number of children is small. There are some types of internal node structures, which are adopted depending on the number of children.

Chapter 3

Static Compressed Double-Array Tries

In our work, we propose compressed static string dictionaries based on double-array tries, described in Section 2.5.4. The double array is a data structure well-used to represent a trie. As the name suggests, it uses two pointer arrays and provides the fastest node-to-node traversal, but its scalability is a significant problem because of the pointer arrays. Although several compression techniques of static double-array tries were proposed to solve this problem [49, 142], these techniques remain problems for scalability.

In this chapter, we present two static compressed double-array tries: the *double array through linear functions (DALF)* and *XOR-compressed double array (XCDA)*, after the data structures of the double array and its existing compact form called the *compact double array (CDA)* are introduced. CDA is a technique that replaces one pointer array into a character array. In general, the character array is represented in smaller space, where each element takes 8 bits in practice. However, there remains the other pointer array. DALF compresses the pointer array such that each element is practically represented in 8 bits.

Although DALF can implement each double-array element in 16 bits, it cannot support the Access operation, because DALF is based on CDA that does not support the operation. Therefore, we cannot use CDA and DALF as string dictionaries. XCDA compresses the original double array with a different approach in order to support both Lookup and Access. We do not give any theoretical evaluation of XCDA, but we show that XCDA can implement static string dictionaries similar in size to DALF, through experiments using real-world datasets.

3.1 Double Arrays

This section shows data structures of double-array tries.

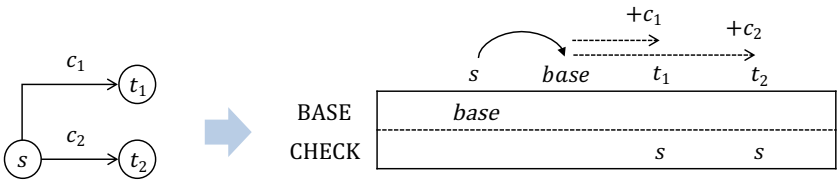


FIGURE 3.1: Node arrangement of a double array.

3.1.1 Data Structure

A double array [2] represents a trie using two integer arrays called BASE and CHECK, where BASE$[s]$ and CHECK$[s]$ correspond to node s. A pair of BASE and CHECK elements corresponding to node s are denoted as *element* s. There exists an edge from internal node s to node t with label c, the double array satisfies the following equations:

$$\text{BASE}[s] + c = t \text{ and } \text{CHECK}[t] = s. \tag{3.1}$$

In other words, node IDs of the double array are arranged to satisfy Equation 3.1 as FIGURE 3.1. The double array can find child t indicated with label c from node s by the following two steps: the child ID is obtained by $t \leftarrow \text{BASE}[s] + c$; and we can identify if the child exists by comparing CHECK$[t]$ to s. Parent s of node t is simply gotten from CHECK$[t]$. The edge label between node s and its parent is obtained by $s - \text{BASE}[\text{CHECK}[s]]$. The most significant advantage of double arrays is the fastest node-to-node traversal.

For space usage, BASE and CHECK can include *empty elements* like hash tables because nodes are located while satisfying Equation 3.1. In this thesis, we call node IDs corresponding to empty elements *unused node IDs*. In other words, the length of BASE (or CHECK) is the sum of trie nodes and empty elements. Let N be the array length, the total space usage of BASE and CHECK is $2N\lceil \log N \rceil$ bits. The space usage depends on not only the number of nodes but also that of empty elements; however, in practice, empty elements are negligibly few compared to trie nodes.

In general, the MP-trie described in Section 2.5.4 is applied to implement compact dictionaries using the double arrays. The separated strings are stored in a character array called TAIL, while appending a special terminator `$` for each string. Links to TAIL are kept in the BASE values of leaves. To identify the leaves, we define a bit array LEAF such that LEAF$[s]$ = `1` iff node s is a leaf. In static construction, we can shorten the length of TAIL by unifying common suffixes of the separated strings [142, 145].

Example 3.1. FIGURE 3.2 *shows an example of a double-array dictionary using an MP-trie for strings* `aaa`, `aabc`, `acb`, `acbab`, *and* `bbab`.[1] *The node IDs are arranged as satisfying*

[1] These strings are used throughout the examples in this chapter. Note that the character values are `a` = 0, `b` = 1, and `c` = 2.

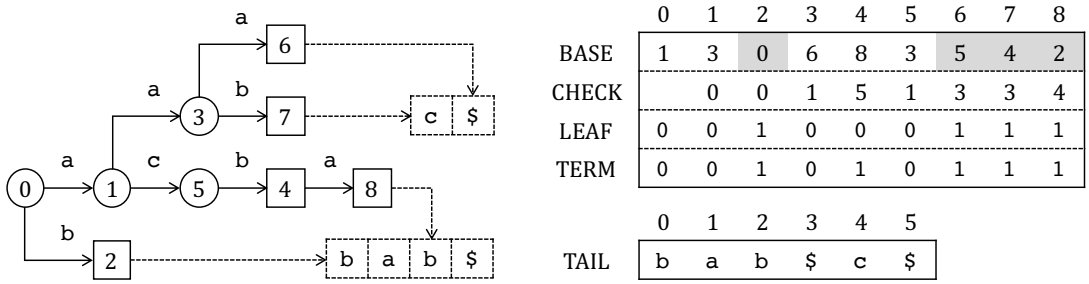


FIGURE 3.2: Double-array representation of an MP-trie. The shaded elements indicate to store TAIL links.

Equation 3.1. The string IDs over `aaa`, `aabc`, `acb`, `acbab`, *and* `bbab` *are assigned by* $\text{RANK}_1(\text{TERM}, 6) = 2$, $\text{RANK}_1(\text{TERM}, 7) = 3$, $\text{RANK}_1(\text{TERM}, 4) = 1$, $\text{RANK}_1(\text{TERM}, 8) = 4$, *and* $\text{RANK}_1(\text{TERM}, 2) = 0$, *respectively. The common suffixes are shared in* TAIL *such as* `bab` *and* `b`.

LOOKUP(`aabc`) *is performed as follows. We first start to traverse nodes from root* 0 *and obtain the child ID* 1 *indicated with the first character* `a` *by* $\text{BASE}[0] + \texttt{a} = 1 + 0 = 1$ *and* $\text{CHECK}[1] = 0$. *Subsequently, we obtain the child ID* 3 *indicated with the second character* `a` *by* $\text{BASE}[1] + \texttt{a} = 3 + 0 = 3$ *and* $\text{CHECK}[3] = 1$, *and the child ID* 7 *indicated with the third character* `b` *by* $\text{BASE}[3] + \texttt{b} = 6 + 1 = 7$ *and* $\text{CHECK}[7] = 3$. *Since node 7 is a leaf from* $\text{LEAF}[7] = 1$, $\text{BASE}[7] = 4$ *indicates the link to* TAIL. *We can see that the query string is registered because* $\text{TAIL}[4]$ *is the same as the last character* `c` *and* $\text{TAIL}[5]$ *has the terminator* `$`. *As a result, the string ID for* `aabc` *is* $\text{RANK}_1(\text{TERM}, 7) = 3$.

In contrast, $\text{ACCESS}(3) =$ `aabc` *is performed as follows. We first obtain the corresponding leaf ID as* $\text{SELECT}_1(\text{TERM}, 3) = 7$. *We then traverse nodes to the root in reverse by* $\text{CHECK}[7] = 3$, $\text{CHECK}[3] = 1$, *and* $\text{CHECK}[1] = 0$. *At the same time, we extract the edge labels by* $7 - \text{BASE}[\text{CHECK}[7]] =$ `b`, $3 - \text{BASE}[\text{CHECK}[3]] =$ `a`, *and* $1 - \text{BASE}[\text{CHECK}[1]] =$ `a`. *The target prefix* `aab` *is obtained by concatenating the labels in reverse. Next, we can see that the target suffix exists in* TAIL *because of* $\text{LEAF}[7] = 1$; *thus, we append the suffix* $\text{TAIL}[\text{BASE}[7]] =$ `c` *to the prefix. As a result, the target string for ID 3 is* `aabc`.

3.1.2 Construction Algorithm

The double array is built by arranging node IDs to satisfy Equation 3.1. Let $\text{EDGE}(s)$ be a set of edge characters from node s. The child IDs from node s are arranged as $\text{BASE}[s] \leftarrow \text{XCHECK}(\text{EDGE}(s))$, where $\text{XCHECK}(E)$ returns an arbitrary integer *base* such that node IDs $base \oplus c$ are unused for each character $c \in E$. Along with defining $\text{BASE}[s]$, the child IDs t are also defined as $t \leftarrow \text{BASE}[s] \oplus c$ and $\text{CHECK}[t] \leftarrow s$ for each character $c \in E$. In other words, arranging node IDs implies searching BASE values satisfying Equation 3.1.

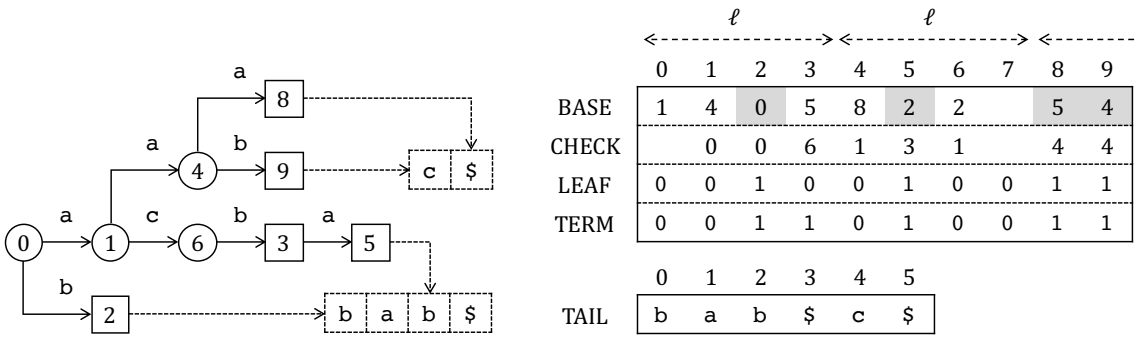


FIGURE 3.3: Double-array representation of an MP-trie, based on Equation 3.2.

3.1.3 Artful Implementation

Aoe defined the left part of Equation 3.1 using a PLUS operation (+) in the original literature [2], whereas recent double-array implementations, such as Darts-clone [138] and Cedar [148], commonly use an XOR operation ($\oplus$) instead. In other words, Equation 3.1 is replaced into

$$\text{BASE}[s] \oplus c = t \text{ and } \text{CHECK}[t] = s. \tag{3.2}$$

Probably, this technique was formally presented by Yata et al. [144] for the first time. In this literature, the technique is proposed to efficiently check the target cache line during node arrangement. The advantage of using the XOR operation is to enable block management of double-array elements. When locating child t indicated with character c from node s, we define $\text{BASE}[s]$ such that node IDs $t = \text{BASE}[s] \oplus c$ are unused, and then store $\text{CHECK}[t] = s$. Let $\ell = 2^{\lceil \log \sigma \rceil}$, $\lfloor \text{BASE}[s]/\ell \rfloor = \lfloor t/\ell \rfloor$ is satisfied due to the XOR operation. That is, the children from the same parent are located within the same block. The feature will be utilized to element compression in Section 3.3 and fast rearrangement in Section 6.2.

Example 3.2. FIGURE *3.3 shows an example of a double-array dictionary based on Equation 3.2. Compared to the double array in* FIGURE *3.2, the node arrangement differs because of the XOR operation. As* $\ell = 4$ *from* $\sigma = 3$*, the elements are partitioned into blocks of size 4. The children from the same parent are located within the same block, such as nodes 3 and 5 from node 1. We can find the child indicated with character* c *from node 1 as* $\text{BASE}[1] \oplus \texttt{c} = 4 \oplus 2 = 6$ *and* $\text{CHECK}[6] = 1$.

3.1.4 Compact Double Array

The main bottleneck of double arrays is that BASE and CHECK are pointer-based arrays whose each element uses $\lceil \log N \rceil$ bits. To solve it, Yata et al. [142] proposed the *compact double array (CDA)* that reduces the space usage of CHECK. CDA uses a new array LCHECK

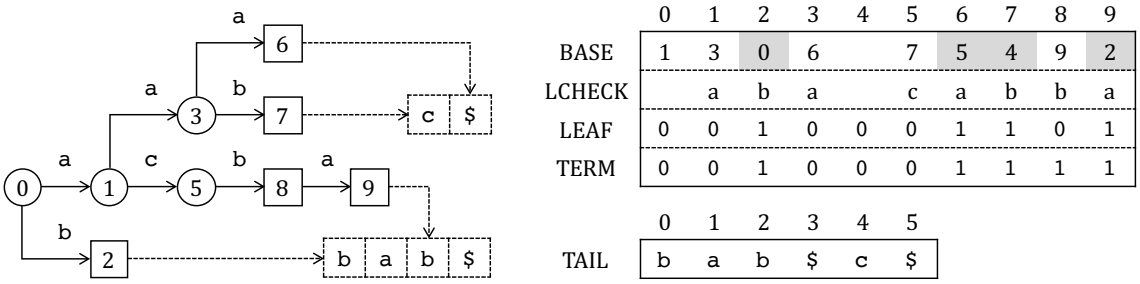


FIGURE 3.4: Compact double-array representation of an MP-trie.

such that LCHECK[s] stores the edge label between node s and its parent, instead of CHECK. In other words, CDA changes Equation 3.1 into the following equations:

$$\text{BASE}[s] + c = t \text{ and } \text{LCHECK}[t] = c. \tag{3.3}$$

Note that $\text{BASE}[s] \neq \text{BASE}[s']$ has to be satisfied for any distinct internal nodes s and s'.

Each LCHECK element can be represented in $\lceil \log \sigma \rceil$ bits. Since practical applications often use byte characters, $\log \sigma \leq 8$ is much smaller than $\log N$, basically 32 or 64. However, CDA cannot support ACCESS because LCHECK does not indicate parents. Moreover, dynamic update cannot be supported because nodes are frequently relocated by using CHECK values, although this chapter focuses on static dictionaries. Therefore, its applications are limited.

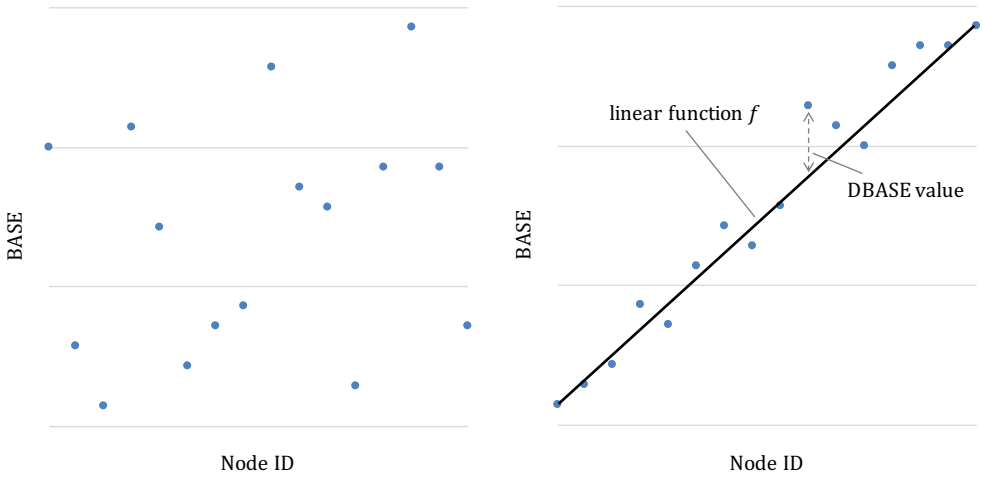
Example 3.3. FIGURE 3.4 *shows an example of a compact double-array representation of an MP-trie.* LCHECK *keeping characters is used instead of* CHECK. *The node IDs are arranged depending on Equation 3.3. We can find the child from node 1 with character* c *as* $\text{BASE}[1] + \mathsf{c} = 3 + 2 = 5$ *and* $\text{LCHECK}[5] = \mathsf{c}$.

3.2 Double Array through Linear Functions

CDA achieves the compression of CHECK, but BASE is uncompressed; thus, the fundamental problem in space is not improved. In this section, we propose a data structure that compresses the BASE array of CDA, namely the *double array through linear functions (DALF)*. DALF represents each BASE element in arbitrary Δ bits in the three steps:

1. Define a linear function $f(s) = \mathsf{a}s + \mathsf{b}$ for node IDs s.
2. Determine $\text{BASE}[s]$ approximating $f(s)$.
3. Use a new array DBASE such that $\text{DBASE}[s] = \text{BASE}[s] - f(s)$, instead of BASE.[2]

[2]Strictly speaking, we round down the result of $f(s)$ as $\lfloor f(s) \rfloor$, but the floor function is omitted for legibility throughout this thesis.



(A) CDA (B) DALF

FIGURE 3.5: Illustration of scatter diagrams for CDA and DALF.

FIGURE 3.5 shows simple examples of scatter diagrams with node IDs on the x-axis and BASE values on y-axis. DALF is built by determining BASE values approximating linear function f and using the differences instead of the original BASE values. When BASE values are determined such that $\text{BASE}[s] - f(s)$ can be represented in Δ bits for all nodes s, each DBASE element can be represented in Δ bits.

However, the approach has a problem that it is difficult to arrange node IDs while determining such BASE values, because Equation 3.3 must be satisfied. If we cannot determine such BASE values during construction, it is necessary to rearrange the node IDs by resetting linear function f from scratch; however, the construction speed becomes slow. It is difficult to define appropriate linear function f in the first place. To solve this problem, we partition the arrays into blocks of size ℓ and defines linear functions $f_b(s) = \mathsf{a}_b s + \mathsf{b}_b$ for each block $0 \leq b \leq \lceil N/\ell \rceil$, where $b = \lfloor s/\ell \rfloor$. In case of failure during construction, we only need to rearrange node IDs for the target block. Furthermore, we can easily define appropriate linear functions f_b.

In this section, we first describe rules on construction and then definitions of linear functions.

3.2.1 Rules on Construction

For discussion, we define the range of child IDs indicated from nodes in block b as $\mathcal{C}_b$, such that $\mathcal{C}_b = \emptyset$ if block b does not include any internal nodes. In addition, we define

$$\mathcal{C}_{0\ldots b} := \bigcup_{k=0}^{b} \mathcal{C}_k.$$

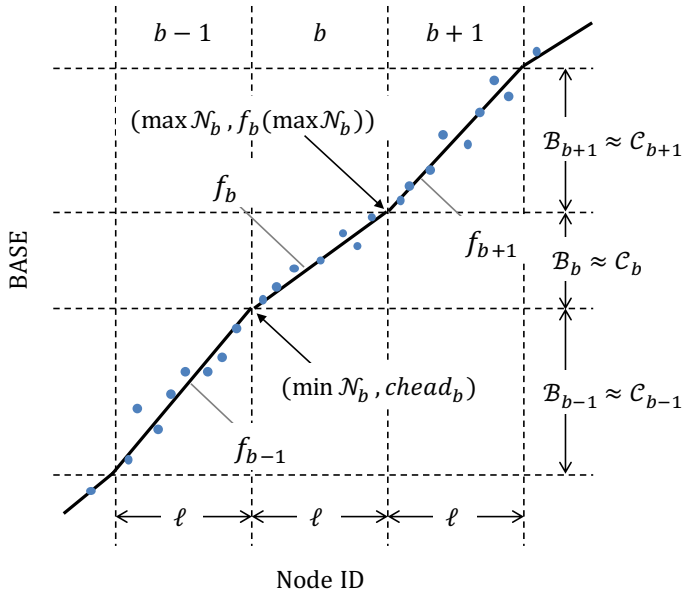


FIGURE 3.6: Illustration of a scatter diagram when Rules 3.1 and 3.2 are kept.

Note that $\mathcal{C}_{0\ldots b} \neq \emptyset$ holds for any b because the first block $b = 0$ always includes the root node. To represent the range of node IDs in block b, we define $\mathcal{N}_b = [b \cdot \ell, (b+1) \cdot \ell)$.

In our discussion, constructing the double array of block b indicates defining BASE values in $\mathcal{N}_b$. The construction of DALF is proceeded block by block in block ID order, while keeping the following rules:

Rule 3.1. $\max \mathcal{C}_{0\ldots b-1} < \min \mathcal{C}_b$ *for* $0 < b$.

Rule 3.2. *For any node with a non-zero block IDs, its children have strictly larger block IDs.*

In this regard, we do not consider the case that block b does not include internal nodes. By keeping Rule 3.1, the following equation is satisfied;

$$\mathcal{C}_b \cap \mathcal{C}_{b+1} = \emptyset. \tag{3.4}$$

Due to Rule 3.1, we can easily rearrange node IDs. This is because we only need to initialize BASE values in $\mathcal{N}_b$ and LCHECK values in $\mathcal{C}_b$ for node rearrangement in block b. Moreover, we can simply define linear function f_b without depending on nodes in other blocks. Rule 3.2 is kept by satisfying $\min \mathcal{N}_{b+1} \leq \min \mathcal{C}_b$ $(b > 0)$ to determine BASE values block by block. FIGURE 3.6 shows the scatter diagram by keeping the rules such as FIGURE 3.5. The details of FIGURE 3.6 are described below.

3.2.2 Definition of Linear Functions

First of all, we define $chead_b$ to discuss definitions of linear functions as the following;

$$chead_b = \begin{cases} 0 & (b = 0) \\ \max\{\max \mathcal{C}_{0\ldots b-1} + 1, \min \mathcal{N}_{b+1}\} & (b > 0). \end{cases} \tag{3.5}$$

The rules can be kept by defining node IDs s traversed from block b such that $chead_b \leq s$.

We divide double-array elements into blocks of ℓ elements and construct the double array for each block in ascending order of block ID, like FIGURE 3.6. During the construction, we define linear functions f_b as a line through point $(\min \mathcal{N}_b, chead_b)$. Here, we show the definitions of a_b and b_b. Note that we suppose block b always includes internal nodes.

To define linear function f_b, it is important to be able to determine BASE values in block b approximating it. Let $\mathcal{B}_b$ be a range of BASE values in block b. If BASE values in block b are determined approximating f_b, $\mathcal{B}_b \simeq [chead_b, \max \mathcal{N}_b]$ will be satisfied. Then, $\mathcal{B}_b \simeq \mathcal{C}_b$ since we can consider $\text{BASE}[s] + c = t$ as $\text{BASE}[s] \simeq t$ in $N \gg \sigma$. In other words,

$$\mathcal{C}_b \simeq [chead_b, f_b(\max \mathcal{N}_b)]. \tag{3.6}$$

If slope a_b is too large, $f_b(\max \mathcal{N}_b)$, or $|\mathcal{C}_b|$, become too large. As a result, $\mathcal{C}_b$ includes many unused node IDs. If slope a_b is too small, $|\mathcal{C}_b|$ becomes too small and we can not determine BASE values in block b approximating f_b. When $\mathcal{C}_b$ does not include unused node IDs and we can determine BASE values in block b near linear function f_b, the slope a_b is the best parameter. Suppose that $\mathcal{C}_b$ does not include unused node IDs ideally, then

$$|\mathcal{C}_b| = sumdeg_b, \tag{3.7}$$

where $sumdeg_b$ denotes the sum of degrees of nodes in block b. Because Equation 3.4 holds by keeping Rule 3.1, Equation 3.7 holds. From Equation 3.6, f_b becomes appropriate in $|[chead_b, f_b(\max \mathcal{N}_b)]| = sumdeg_b$. Consequently, slope a_b is defined as following equation;

$$\mathsf{a}_b = \frac{sumdeg_b}{\ell} \quad (b > 0). \tag{3.8}$$

While $sumdeg_b$ is not changed after f_b is defined from Rule 3.2, $sumdeg_0$ can not be calculated because the rule omits the first block $b = 0$. Thus, we define a_0 by using the average degree in a trie. Let $avedeg$ be the average degree of internal nodes, we consider $sumdeg_0$ as $avedeg \cdot \ell$. Slope a_0 is defined as

$$\mathsf{a}_0 = \frac{avedeg \cdot \ell}{\ell} = avedeg. \tag{3.9}$$

Y-intercept b_b is defined by using slope a_b and the point $(b \cdot \ell, chead_b)$ as

$$\mathsf{b}_b = chead_b - \mathsf{a}_b \cdot b \cdot \ell. \tag{3.10}$$

Reconstruction If BASE$[s]$ cannot be determined such that BASE$[s] - f_b(s)$ is represented in Δ bits, it is necessary to redetermine BASE values on $\mathcal{N}_b$ after slope a_b increases to expand $\mathcal{C}_b$. Slope a_b is updated as the equation;

$$\mathsf{a}_b = \hat{\mathsf{a}}_b + gain \cdot r_b, \tag{3.11}$$

where *gain* is the addition value to slope a_b, r_b is the number of reconstructions in block b, and $\hat{\mathsf{a}}_b$ is the initial parameter defined by Equations 3.8 or 3.9. If parameter *gain* is too large, $\mathcal{C}_b$ expands too much and the number of empty elements in block b increases. If parameter *gain* is too small, $\mathcal{C}_b$ expands little and r_b increases. As a result, the construction speed becomes slow. Therefore, parameter *gain* needs to be defined as an appropriate value.

Furthermore, we give the proof that the reconstruction using Equation 3.11 can always terminate for any tries. Here, the upper bound of r_b is shown by discussing the upper bound of slope a_b. To discuss the upper bound of slope a_b, suppose the following cases;

Case 3.1. *In terms of the number of trie nodes and the size of* DBASE *element, the worst cases for block b are shown the follows;*

- *Block b does not include an unused node ID, and the nodes are all internal nodes.*
- *The degree of each node is σ.*
- $\Delta = 1$, *that is,* BASE$[s] = f_b(s)$.

The slope a_b for this case is the upper bound. In the case, the number of internal nodes in the block is ℓ and each internal node has σ children. Node IDs can be arranged freely without depending on nodes in other blocks from Equation 3.4. Since the children traversed from block b are consecutive, $\mathcal{C}_b$ does not include unused node IDs and $|\mathcal{C}_b| = \ell \cdot \sigma$ is satisfied. Therefore, the slope a_b satisfying the equation in $\Delta = 1$ is the upper bound. If BASE values are arranged in ascending order, such values for the worst case satisfy

$$\text{BASE}[\min \mathcal{N}_b + i] = chead_b + i \cdot \sigma \ (0 \leq i < \ell). \tag{3.12}$$

The slope a_b and y-intercept b_b to satisfy Equation 3.12 in $\Delta = 1$ (or BASE$[s] = f_b(s)$) are as follows;

$$\mathsf{a}_b = \sigma \text{ and } \mathsf{b}_b = chead_b - \min \mathcal{N}_b \cdot \sigma. \tag{3.13}$$

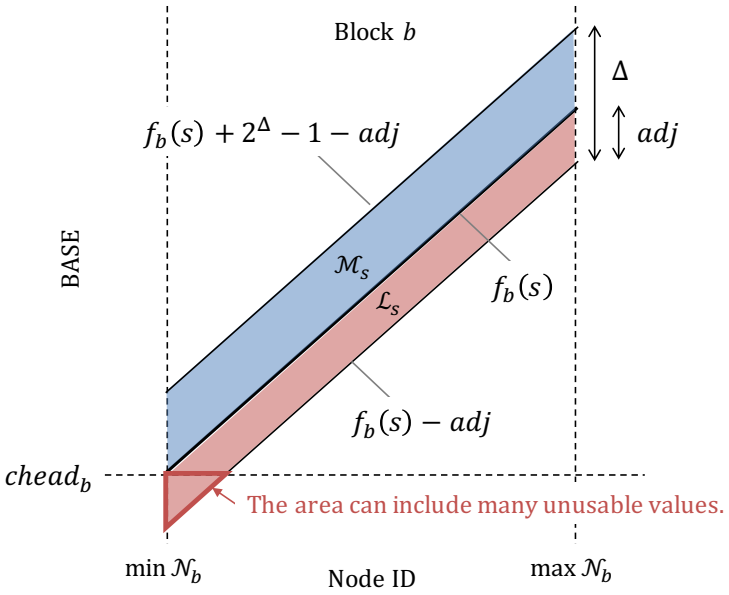


FIGURE 3.7: Illustration for ranges of BASE values in block b.

From $\mathsf{a}_b = \sigma$ in Equation 3.13, the number of reconstructions in a block is always as

$$r_b \leq \left\lceil \frac{\sigma - \hat{\mathsf{a}}_b}{gain} \right\rceil. \tag{3.14}$$

The discussion shows that reconstruction processing can always terminate in any case. From 3.14, the number of reconstructions does not depend on that of trie nodes; therefore, reconstruction using Equation 3.11 is practical for a large trie.

3.2.3 Ranges of BASE values

Here, we show ranges of BASE values to represent DBASE values in Δ bits. The range of BASE$[s]$ is defined as

$$\text{BASE}[s] \in (\mathcal{M}_s \cup \mathcal{L}_s), \tag{3.15}$$

where $\mathcal{M}_s$ and $\mathcal{L}_s$ are ranges of BASE values. FIGURE 3.7 shows an example for ranges of BASE values in block b. Higher and lower BASE values than $f_b(s)$ are $\mathcal{M}_s$ and $\mathcal{L}_s$, respectively. Note that $f_b(s) \in \mathcal{L}_s$.

$\mathcal{M}_s$ and $\mathcal{L}_s$ are adjusted by using parameter adj ($0 \leq adj < 2^\Delta - 1$). $\mathcal{M}_s$ is represented as

$$\mathcal{M}_s = [f_b(s) + 1, f_b(s) + 2^\Delta - 1 - adj). \tag{3.16}$$

To represent empty elements using DBASE, $\mathcal{M}_s$ does not include $f_b(s) + 2^\Delta - 1 - adj$. In other words, DBASE$[s] = 2^\Delta - 1$ indicates that element s is an empty element. $\mathcal{L}_s$ is

represented as

$$\mathcal{L}_s = [f_b(s) - adj, f_b(s)]. \tag{3.17}$$

In terms of values in $\mathcal{L}_s$ smaller than $chead_b$, many unusable BASE values can be included because of the following reasons;

- To keep the rules, DALF determines node ID t traversed from block b such that $chead_b \leq t$. Because $\text{BASE}[s] + c = t$ from Equation 3.3, $chead_b \leq \text{BASE}[s] + c$. From the equation, BASE values smaller than $chead_b - c$ can not be used in block b.
- When BASE values smaller than $chead_b$ are used in block $b' < b$, the BASE values can not be used in block b.

Therefore, $\mathcal{L}_s$ can include many unusable values when adj is big and $|\mathcal{L}_s|$ is large. On the other hand, when $|\mathcal{L}_s|$ decreases with decreasing adj, BASE values become big and $|\mathcal{C}_b|$ becomes large, that is to say, the number of empty elements increases. Therefore, parameter adj requires to be defined as appropriate value.

3.2.4 Data Structure

In DALF, Equation 3.3 is changed as follows:

$$\text{DBASE}[s] + f_b(s) + c = t \text{ and } \text{LCHECK}[t] = c. \tag{3.18}$$

Note that $\text{DBASE}[s] + f(s) \neq \text{DBASE}[s'] + f(s')$ has to be satisfied for any distinct internal nodes s and s' because DALF is based on CDA. The node-to-node traversal can be performed in constant time.

For space usage, DBASE and LCHECK use Δ and $\lceil \log \sigma \rceil$ bits for each element. Representation of the linear functions (i.e., a_b and b_b) uses $2w$ bits for each block, where w denotes the word length. The overall space usage is $N(\Delta + \lceil \log \sigma \rceil) + 2w\lceil N/\ell \rceil$ bits. The compression efficiency of DALF depends on parameters Δ and ℓ, but we cannot theoretically compare the space usage with other double arrays, because the number of empty elements is greatly affected from the parameters Δ, ℓ, adj and $gain$, empirically. Therefore, we evaluate the space efficiency through experiments.

As for dictionary implementation, we apply the MP-trie to DALF in the same manner as the conventional double-array dictionaries. Although TAIL links cannot be represented in Δ bits, we can easily solve the problem. We first construct an FID to LEAF to support RANK, and then introduce a new integer array LINK. By using these arrays, a link value a of leaf s is stored as $\text{DBASE}[s] = a \bmod 2^{\Delta}$ and $\text{LINK}[\text{RANK}_1(\text{LEAF}, s)] = \lfloor a/2^{\Delta} \rfloor$. Let M be the length of TAIL, LINK uses $\lceil \log M \rceil - \Delta$ bits for each element. The length of LINK is the same as the number of leaves.

3.3 XOR-Compressed Double Array

DALF can implement a double array algorithm in compact space (although depending on the parameters), but there is a fundamental problem that LCHECK cannot obtain parent IDs. In other words, DALF cannot support ACCESS in common with CDA. To implement string dictionaries supporting ACCESS, it is necessary to directly compress the original double array.

In this section, we attempt to compress the BASE and CHECK arrays using DACs; however, simply representing an array of many large integers using DACs is inefficient in space and time. In our work, we solve this problem by transforming BASE and CHECK into arrays of many small integers. Moreover, we propose a fast version of DACs not to sacrifice the traversal speed.

3.3.1 XOR Transformation

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

Theorem 3.1. *Integer $P'[i]$ can be represented in $\log \ell$ bits for $P[i]$ such that $\lfloor P[i]/\ell \rfloor = \lfloor i/\ell \rfloor$.*

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

Example 3.4. *Let $P[23] = 21$ in $\ell = 4$. Function $\lfloor 23/4 \rfloor = 5$ denotes to right shift $(23)_2 =$ `10111` by $\log 4 = 2$ bits as $(5)_2 =$ `101`. Similarly, $\lfloor 21/4 \rfloor = 5$ denotes to right shift $(21)_2 =$ `10101` by 2 bits. Binaries `10111` and `10101` consist of the same bits except for the lowest 2 bits because $\lfloor 23/4 \rfloor = \lfloor 21/4 \rfloor$. Therefore, $P'[23] = 21 \oplus 23 = 2$ can be represented in 2 bits as `10111` $\oplus$ `10101` $=$ `00010`.*

3.3.2 Construction Algorithm

As described in Section 2.2.3, 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 3.1 in $\ell = 2^b$, most P' values are in $\log \ell = b$ bits. For BASE and CHECK, the values can be freely determined as long as

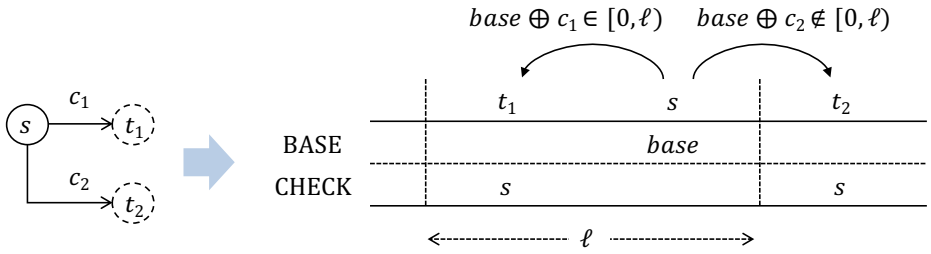


FIGURE 3.8: Relation between node s and its children t_1 and t_2. Suppose that $\text{BASE}[s] = base$ satisfies $\lfloor base/\ell \rfloor = \lfloor s/\ell \rfloor$, $\lfloor \text{CHECK}[t_1]/\ell \rfloor = \lfloor s/\ell \rfloor = \lfloor t_1/\ell \rfloor$ is also satisfied in $c_1 \in [0, \ell)$.

Equation 3.1 is satisfied. Therefore, we can obtain BASE and CHECK values satisfying the condition in $\ell = 2^b$.

We present a function YCHECK_ℓ that attempts to determine BASE values satisfying the condition. Let E be a set of edge characters from node s, we define BASE values as $\text{BASE}[s] \leftarrow \text{YCHECK}_\ell(E, s)$.

Algorithm 3.1 $\text{YCHECK}_\ell(E, s)$

```
1: for all base ← [⌊s/ℓ⌋ · ℓ, (⌊s/ℓ⌋ + 1) · ℓ) do
2:     if Node IDs base ⊕ c are unused for each c ∈ E then
3:         return base                                  ▷ ⌊base/ℓ⌋ = ⌊s/ℓ⌋
4:     end if
5: end for
6: return XCHECK(E)                                     ▷ ⌊XCHECK(E)/ℓ⌋ ≠ ⌊s/ℓ⌋
```

Function $\text{YCHECK}_\ell(E, s)$ targets to determine $\text{BASE}[s]$ such that $\lfloor \text{BASE}[s]/\ell \rfloor = \lfloor s/\ell \rfloor$. This loop searches such $\text{BASE}[s]$ satisfying Equation 3.1 on the block $\lfloor s/\ell \rfloor$. If the loop cannot find it, $\text{BASE}[s]$ is determined in the same manner as the conventional algorithm.

Function $\text{YCHECK}_\ell(E, s)$ is effective for characters $c \in [0, \ell)$ as the following reason. Let t be the child of node s with such character c, the following equation is satisfied;

$$\lfloor \text{BASE}[s]/\ell \rfloor = \lfloor (\text{BASE}[s] \oplus c)/\ell \rfloor = \lfloor t/\ell \rfloor. \tag{3.19}$$

When $\lfloor \text{BASE}[s]/\ell \rfloor = \lfloor s/\ell \rfloor$ is satisfied, Equation 3.19 and the right part of Equation 3.1 give $\lfloor s/\ell \rfloor = \lfloor t/\ell \rfloor = \lfloor \text{CHECK}[t]/\ell \rfloor$. That is to say, we only have to search $\text{BASE}[s]$ such that $\lfloor \text{BASE}[s]/\ell \rfloor = \lfloor s/\ell \rfloor$ in order to obtain $\text{BASE}[s]$ and $\text{CHECK}[t]$ satisfying the condition of Theorem 3.1 (see FIGURE 3.8).

In practice, $\sigma \leq 256$ always holds because byte characters are used to edge labels. Therefore, YCHECK_ℓ can obtain BASE and CHECK values satisfying the condition in $\ell = 2^8 = 256$. In other words, the function is compatible with the byte-oriented DACs with $b = 8$. The effectiveness will be shown in Section 3.4.

3.3.3 Data Structure

We call our data structure the *XOR-compressed double-array (XCDA)*. Let XBASE and XCHECK be arrays such that XBASE$[i]$ = BASE$[i] \oplus i$ and XCHECK$[i]$ = CHECK$[i] \oplus i$, respectively. XCDA is built by representing XBASE and XCHECK using the byte-oriented DACs. Since YCHECK$_\ell$ can provide XBASE and XCHECK including many 8-bit integers, 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 values will be large by using the XOR transformation. Therefore, XCDA represents the values as follows.

- As CHECK$[t] = s$ means that the parent of node t is node s, inequation $s \neq t$ always holds. We can consider CHECK$[i] = i$ as empty elements. The XCHECK values always become zero because of XCHECK$[i]$ = CHECK$[i] \oplus i = i \oplus i = 0$. If BASE$[s]$ is empty, CHECK$[s]$ is also empty. Therefore, we do not have to identify if BASE elements are empty. XCDA sets BASE$[i] = i$ for empty elements.

- XCDA represents TAIL links by using the first array A_1 and an additional array LINK, in the same manner as DALF. Suppose BASE$[s] = a$ in LEAF$[s] = 1$, $A_1[s] = a \bmod 2^b$ and LINK[RANK$_1$(LEAF, s)] $= \lfloor a/2^b \rfloor$. XCDA supports fast extraction of TAIL links because only A_1 and LINK are used.

Example 3.5. FIGURE 3.9 *shows an example of XCDA for the double array of* FIGURE 3.2. *The shaded elements denote* TAIL *links. Except for the links,* XBASE *and* XCHECK *are built by using the XOR transformation. For example,* XCHECK[3] *is transformed by* CHECK$[3] \oplus 3 = 6 \oplus 3 = 5$. *Empty* XBASE[7] *and* XCHECK[7] *become zero by setting* BASE$[7] =$ 7 *and* CHECK$[7] = 7$. *For the* TAIL *link* BASE$[9] = 4$, *the lowest* b *bits of* (BASE$[9]$)$_2 = (4)_2 =$ *100 and the rest bits are stored in* XBASE[9] *and* LINK[RANK$_1$(LEAF, 9)] = LINK[3], *respectively. Let* $b = 2$, XBASE[9] = *00 and* LINK[3] = *1. XCDA is built by representing the* XBASE *and* XCHECK *using DACs.*

It is very easy to extract original BASE *and* CHECK *values from the XCDA. Value* CHECK$[3] = 6$ *is extracted by* XCHECK$[3] \oplus 3 = 5 \oplus 3 = 6$. *From* BASE$[7] = 0$, *we can identify that this element is empty. From* LEAF$[9] = 1$, *the link* (BASE$[9]$)$_2 = (4)_2 =$ *100 is extracted by concatenating the bits* LINK[RANK$_1$(LEAF, 9)] = LINK$[3] =$ *1 and* XBASE[9] = *00.*

3.3.4 Pointer-based Fast DACs

We introduced the technique to transform BASE and CHECK into XBASE and XCHECK including many small integers, respectively. XCDA represents XBASE and XCHECK by

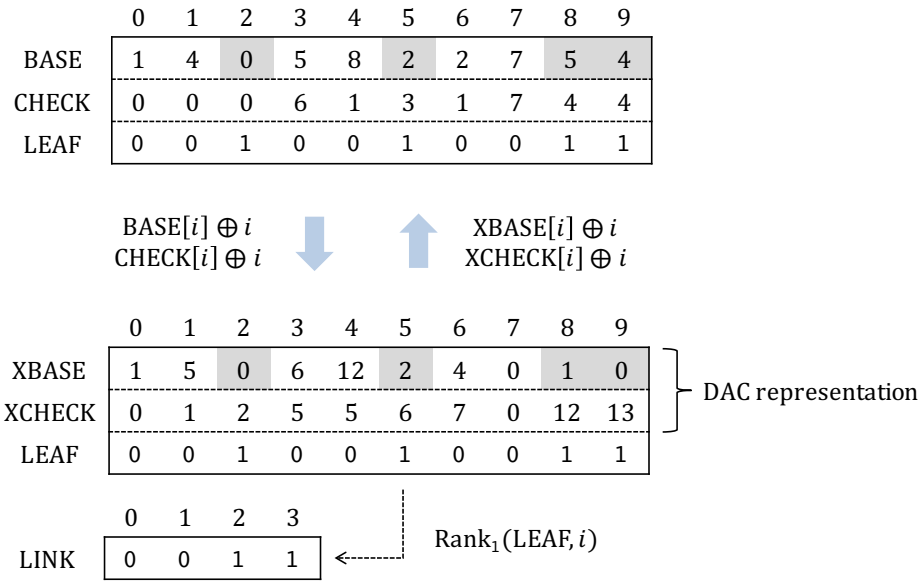


FIGURE 3.9: Transformed arrays in $b = 2$ from the double array of FIGURE 3.3.

using DACs. On the other hand, all XBASE and XCHECK values will be not represented in b bits because of Equation 3.1. Although 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 will be not competitive to that of double arrays using plain pointers. This section presents new pointer-based DACs called *fast DACs (FDACs)*, supporting directly extraction without RANK.

For simplicity, we introduce FDACs corresponding to DACs in Section 2.2.3. More precisely, $P[i]$ is extracted through the same path, $i_1, i_2, \ldots, i_{k_i}$. FIGURE 3.10 shows an example of a FDAC representation. In this figure, as FIGURE 2.1, $P[5]$ is extracted by following the 5 and 3 positions on the first and second arrays, respectively. Such FDACs consist of the following arrays:

- 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 Section 2.2.3.
- 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 $\text{RANK}_1(B'_j, i_j)$, 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

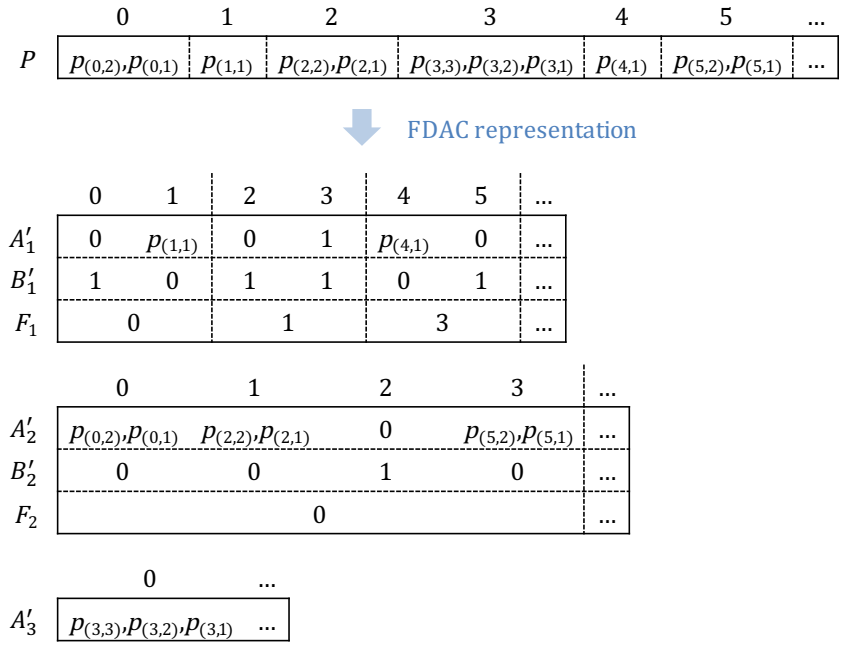


FIGURE 3.10: FDAC representation corresponding to the DACs of FIGURE 2.1. We assume the DACs with $b = 1$ and the FDACs with $b_1 = 1, b_2 = 2$ and $b_3 = 3$, that is, $r_1 = 2$ and $r_2 = 4$.

head of blocks on A'_j, as $F_j[x] = \text{RANK}_1(B'_j, r_j x)$. Arrays A'_j store the differences as $A'_j[i_j] = \text{RANK}_1(B'_j, i_j) - F_j[\lfloor i_j/r_j \rfloor]$. 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 Equation 2.1 into

$$i_{j+1} = A'_j[i_j] + F_j[\lfloor i_j/r_j \rfloor] \quad (B'_j[i_j] = 1). \tag{3.20}$$

We explain how to carry out the extraction using the example of FIGURE 3.10. 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 $\text{RANK}_1(B_1, 5) = 3$, 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 $\text{RANK}_1(B'_1, 2r_1) = \text{RANK}_1(B'_1, 4) = 3$, $A'_1[5]$ 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 $\mathcal{O}(L)$, 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 XBASE and XCHECK values represented in A'_1 because of YCHECK. Therefore, FDACs is 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.

On the other hand, YCHECK works for characters $c \in [0, 128)$ 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, but byte characters generated by splitting multi-byte ones such as UTF-8 in Japanese and Chinese often satisfy $128 < \sigma$. We can simply solve this problem by assigning code values to characters in descending order of frequency of its appearance in the dataset. In reality, the distribution of characters is often heavily skewed. Therefore, most of characters are represented in range $[0, 128)$. For example, in all page titles of the Japanese Wikipedia of Jan. 2015 [133], 99.7% characters are represented in range $[0, 128)$ although $\sigma = 189$.

3.4 Experimental Evaluation

This section analyzes practical performances of DALF and XCDA through real-world datasets. We compare them with other string dictionaries and give evaluations in practice.

3.4.1 Settings

We carried out the experiments on Quad-Core Intel Xeon 2 x 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 six real-world datasets:

- GEONAMES: Geographic names on the *asciiname* column from GeoNames dump [51].
- NWC: Japanese word N-grams in the Nihongo Web Corpus 2010 [141].
- JAWIKI: All page titles from the Japanese Wikipedia of Jan. 2015 [133].
- ENWIKI: All page titles from the English Wikipedia of Feb. 2015 [133].
- UK: URLs of a 2005 crawl by the UbiCrawler [22] on the `.uk` domain [82].
- DNA: All substrings of 12 characters found in the Gene DNA data set from Pizza&Chili corpus [121].

TABLE 3.1 summarizes the statistics about each dataset: the raw size in MiB, number of different strings, average number of characters per string when including a terminator,

TABLE 3.1: Information about datasets.

	Size	Strings	Ave. length	σ	# of nodes	|TAIL|
GEONAMES	106.1	6,784,722	15.6	96	11,378,833	8,733,434
NWC	460.8	20,722,756	22.2	180	52,047,795	548,133
JAWIKI	33.9	1,518,205	22.3	189	3,516,248	5,234,145
ENWIKI	238.2	11,519,354	20.7	199	25,749,451	23,108,877
UK	2,855.5	39,459,925	72.4	103	117,568,967	289,826,785
DNA	198.5	15,265,943	13.0	16	20,688,222	39,244

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 our dictionaries to previous double-array tries and state-of-the-art compressed string dictionaries. For DALF parameters, we chose $\Delta = 8, \ell = 512, adj = 128$ and $gain = 1.0$ from preliminary experiments [74]. For XCDA-tries, we tested the four patterns:

- *XCDA-x* using the byte-oriented DACs and XCHECK.
- *XCDA-y* using the byte-oriented DACs and YCHECK$_{256}$.
- *FXCDA-x* using the byte-oriented FDACs and XCHECK.
- *FXCDA-y* using the byte-oriented FDACs and YCHECK$_{128}$.

For previous double arrays, we tested the original double array (DA) [2] and CDA [142], representing the MP-trie. Note that DALF and CDA can not support ACCESS. We implemented XCHECK using fast algorithms proposed in [99]. All the double-array implementations are available at https://github.com/kampersanda/cda-tries. We can get more technical details from the library.

As for the state-of-the-art, *Cent* is the centroid path-decomposed trie and *Cent-rp* is the compressed version with Re-Pair [83], from [57]. We also tested *PFC*, *HTFC-rp* and *HashDAC-rp* from [95]. PFC is a plain Front-Coding dictionary. HTFC-rp is a Hu-Tucker [65] Front-Coding dictionary compressed with Re-Pair. HashDAC-rp is a hashing dictionary compressed with 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 [8, 57]. 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. [95] evaluated 5 load factors. Since their performances do not change significantly, we chose load factor 0.5 (i.e., half slots are empty) supporting the fastest LOOKUP. Although LZ-compressed string dictionaries [8] are effective for

TABLE 3.2: Experimental results about percentages of values on each level in DACs and FDACs.

(A) GEONAMES

	1st	2nd	3rd	4th
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

	1st	2nd	3rd	4th
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

	1st	2nd	3rd	4th
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

	1st	2nd	3rd	4th
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

	1st	2nd	3rd	4th
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) DNA

	1st	2nd	3rd	4th
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	–

synthetic datasets containing many often repeated substrings, Cent-rp outperforms the LZ-dictionaries for real-world datasets from previous experiments; therefore, our experiments did not include them. Cent and Cent-rp were implemented by using the software *path_-decomposed_tries* [116]. PFC, HTFC-rp and HashDAC-rp were implemented by using the software *libCSD* [93].

3.4.2 Results

We first observe DACs and FDACs using XCHECK and YCHECK. Next, we evaluate the practical performance of our data structures.

Construction Algorithms TABLE 3.2 shows the percentages of values for each level of DACs and FDACs using XCHECK and YCHECK. In DACs, A_j can represent $8j$-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 Equation 3.1. Function YCHECK provides

better results than XCheck, especially in FDACs whose allocation of the 1st level is smaller. Therefore, YCheck can contribute to improvement of XCDAs.

String Dictionaries Tables 3.3–3.8 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. 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 because the construction complexity exceeded the memory resources of our computational configuration. Moreover, it did not complete the construction on dna in 6 hours; hence we had to kill the process.

When comparing the construction algorithms in XCDA, using YCheck slightly outperforms using XCheck because more values are represented in the 1st level. Function YCheck provides better compression ratios in all cases because they obediently depend on the percentages in Table 3.2. The Lookup and Access times also depend largely on the percentages; therefore YCheck provides faster operations in most cases. There are no significant problems in construction. Therefore, using YCheck is a better choice.

When comparing the XCDA tries using YCheck, 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, uk and dna while XCDA-y is superior in geonames, jawiki and enwiki. 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 XCDA tries.

We compare FXCDA-y with DALF. DALF provides competitive compression ratios, and faster Lookup in geonames, jawiki, and dna; however, the Lookup becomes slow for large datasets such as uk because DALF is built by arranging nodes block by block. The construction algorithm can locate parent and child nodes far away on the array; therefore, cache misses can occur frequently in node-to-node 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. In what follows, we compare it to other data structures.

Compared to the previous double-array tries, FXCDA-y is 1.7–2.6x smaller than DA and solves the problem that we cannot apply the previous double-array 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

Table 3.3: Experimental results about string dictionaries for GEONAMES.

	Constr. (sec)	Cmpr. (%)	Lookup (μs/str)	Access (μs/ID)
New double-array tries				
DALF	9.5	52.8	**0.80**	–
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 double-array tries				
DA	**4.9**	95.8	0.61	**0.95**
CDA	5.0	**63.7**	**0.49**	–
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

Table 3.4: Experimental results about string dictionaries for NWC.

	Constr. (sec)	Cmpr. (%)	Lookup (μs/str)	Access (μs/ID)
New double-array tries				
DALF	35.4	**34.2**	1.88	–
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 double-array tries				
DA	**13.7**	92.4	1.00	**1.58**
CDA	14.7	**58.5**	**0.83**	–
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

Table 3.5: Experimental results about string dictionaries for JAWIKI.

	Constr. (sec)	Cmpr. (%)	Lookup (μs/str)	Access (μs/ID)
New double-array tries				
DALF	2.6	56.1	**0.61**	–
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 double-array tries				
DA	**1.3**	100.3	0.52	**0.90**
CDA	**1.3**	**69.1**	**0.40**	–
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

Table 3.6: Experimental results about string dictionaries for ENWIKI.

	Constr. (sec)	Cmpr. (%)	Lookup (μs/str)	Access (μs/ID)
New double-array tries				
DALF	23.3	51.5	1.39	–
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 double-array tries				
DA	**11.0**	98.1	0.82	**1.31**
CDA	11.3	**65.7**	**0.67**	–
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

TABLE 3.7: Experimental results about string dictionaries for UK.

	Constr. (sec)	Cmpr. (%)	LOOKUP (μs/str)	ACCESS (μs/ID)
New double-array tries				
DALF	110.1	**24.1**	6.00	–
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 double-array tries				
DA	**65.9**	43.8	1.95	**2.93**
CDA	68.0	**31.5**	**1.63**	–
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)	12598.4	18.3	7.96	4.41
HashDAC-rp	–	–	–	–

TABLE 3.8: Experimental results about string dictionaries for DNA.

	Constr. (sec)	Cmpr. (%)	LOOKUP (μs/str)	ACCESS (μs/ID)
New double-array tries				
DALF	7.8	**33.0**	**0.54**	–
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 double-array tries				
DA	**4.5**	87.4	0.58	**0.88**
CDA	6.0	**55.3**	**0.46**	–
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	–	–	–	–

for large datasets such as UK because of technological factors as follows. DALF is built by arranging nodes in breadth-first order while general double-array 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 DNA. Moreover, it provides the fastest LOOKUP. The running time of FXCDA-y is up to 3x and 2x 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 2.6x, 1.8x and 1.5x 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 3.1x and 2.0x faster than those of Cent-rp and HashDAC-rp, respectively. Compared to HTFC-rp, FXCDA-y is up to 5.5x faster in bucket size 8. 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.

Chapter 4

Static Dictionary Compression Using Dictionary Encoding

Chapter 3 presented novel compressed string dictionaries based on improved double-array tries. They can provide very fast search operations based on the double-array algorithms, but their space usages are larger than those of existing dictionaries compressed using powerful text compression techniques. In particular, Re-Pair offers very compact dictionary structures as presented in [57, 95]; however, the construction costs are very large. Therefore, more lightweight but powerful compression approaches are required.

In this chapter, we propose an alternative compression strategy that applies dictionary encoding (a.k.a. domain encoding) to dictionary compression rather than using Re-Pair. Dictionary encoding replaces strings into integers using a string dictionary. This is a widely used technique for compression and query acceleration as in column-store database systems [1, 85, 102, 151]. Actually, such a strategy has already attempted as in [4, 113, 139]. The existing data structures compress trie structures using the same structures, recursively. As our work is inspired by the existing studies, we present how to apply our strategy to the state-of-the-art string dictionaries compressed by Re-Pair at present in Section 4.1. Moreover, we develop novel dictionary structures suited to dictionary encoding in Section 4.2.

4.1 String Dictionaries and Compression Strategies

Encoding strings $x_1, x_2, ..., x_n$ into integers $i_1, i_2, ..., i_n$ using a string dictionary is referred to as *dictionary encoding*. Basically, the space required to store integers is less than that needed to store strings. In our work, we attempt to improve existing string dictionaries by applying dictionary encoding to strings appearing in them. This section describes data structures of existing string dictionaries based on the path-decomposed trie (PDT) [57] and Front Coding [95], while presenting how to apply dictionary encoding to these.

In this section, we show examples for each string dictionary using a set of strings `ideal`, `ideas`, `ideology`, `tea`, `techie`, `technology`, `tie`, and `trie`.

4.1.1 Path-Decomposed Trie Dictionaries

A PDT is a tree structure constructed by recursively decomposing a trie into node-to-leaf paths. Each node of the PDT corresponds to each path in the trie. We introduce a succinct PDT representation proposed in [57].

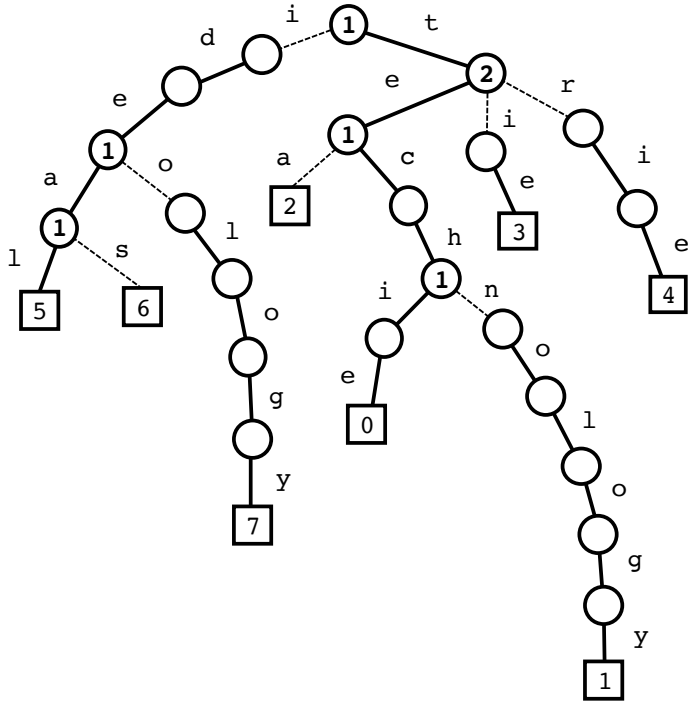
The tree in FIGURE 4.1b is a PDT constructed by decomposing the trie in FIGURE 4.1a into node-to-leaf paths connected by a solid line. The nodes and edges have string labels and branching characters, respectively. Such a PDT is constructed as follows. First, we choose a root-to-leaf path π in the trie. Second, we create a string by concatenating edge characters along path π, interleaved with special characters $\mathbf{1}, \mathbf{2}, \ldots$ that indicate how many subtries branch off that point along path π. Third, we associate the string with root node u_π of the PDT. The children of root node u_π are recursively defined as the root nodes of PDTs corresponding to each subtrie hanging off the path π.

Although a strategy of choosing a path is arbitrary, the example chooses the heavy path [46]. The heavy path always follows a heavy child, which is the one whose subtrie has the most leaves. This strategy is called *centroid path decomposition*. The height of the resulting tree is bounded by $\mathcal{O}(\log |\mathcal{X}|)$ and the average retrieval time is bounded by $\mathcal{O}(\frac{\log |\mathcal{X}|}{\log \sigma})$. In other words, centroid path decomposition can reduce the number of node-to-node random accesses. The centroid path decomposition is a default setting in our work.

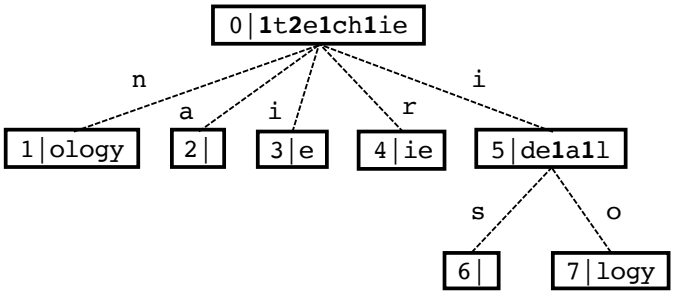
For PDT representation, each node s is represented by three sequences: L_s stores the node label, E_s stores the branching characters from node s in reverse, and B_s is the DFUDS representation of node s. The node IDs are assigned in depth-first order. The PDT is represented by sequences L, E, and B obtained by concatenating L_s, E_s, and B_s in order of node ID. The boundaries of the node labels are maintained using the Elias-Fano representation [42, 43]. We do not describe how to perform LOOKUP and ACCESS because of be complex. The interested reader can refer to the original literature [57].

Compression Strategies Grossi and Ottaviano [57] compressed the sequence of node labels L using a variant of Re-Pair based on the approximate Re-Pair [29]. This version can support scanning labels in constant time per character. In other words, the decoding and construction costs are reduced, while some space efficiency is sacrificed.

On the other hand, our strategy replaces node labels with integer IDs using dictionary encoding. As shown at the bottom of FIGURE 4.1b, the sequence of IDs L' is generated from L. Note that the node labels consist of characters drawn from $\Sigma' = \Sigma \cup \{\mathbf{1}, \mathbf{2}, \ldots, \sigma - \mathbf{1}\}$. In practice, $\Sigma' = [0, 511)$ because $\Sigma = [0, 256)$. We encode the characters into byte characters



(A) Trie



	0	1	2	3	4	5	6	7
L	1t2e1ch1ie|ology||ie|e|de1a1l||logy							
	01234567							
L'	GFADCBAE							

Dictionary encoding

A	
B	de1a1l
C	e
D	ie
E	logy
F	ology
G	1t2e1ch1ie

	0123456789012345678
E	irianos
B	((((((((((((((()))))

(B) PDT

FIGURE 4.1: String dictionaries based on a trie and PDT.

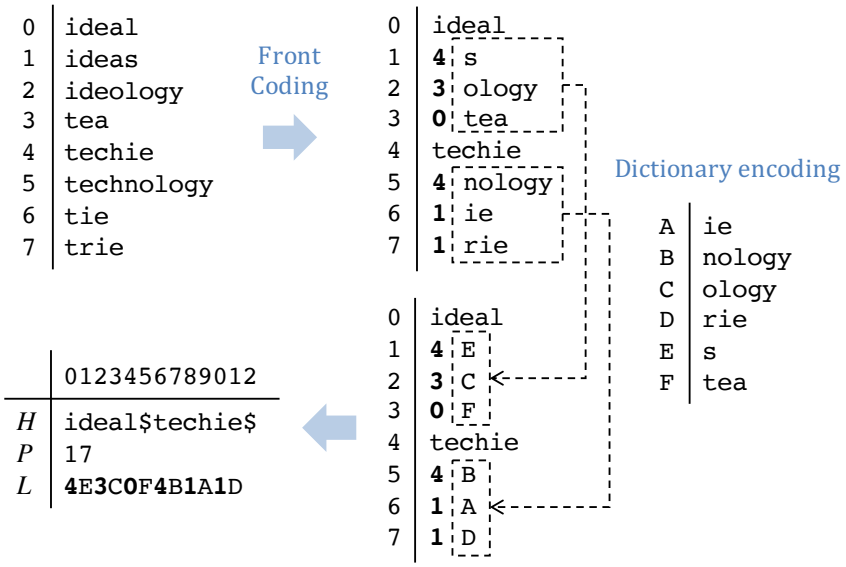


FIGURE 4.2: String dictionary based on Front Coding with $b = 4$.

using VByte coding [134] to use dictionary encoding. To shorten the length of the byte sequence, we assign character code values from zero in order of frequency of appearance. The Elias-Fano representation is not needed because L' is a fixed-length array.

4.1.2 Front Coding Dictionaries

Front Coding [134] is a technique to compress lexicographically sorted strings. As explained in Section 2.5.2, we can apply the technique for dictionary implementation by dividing the strings into buckets. The top of FIGURE 4.2 shows an example of front coding with $b = 4$.

Compression Strategies Martínez-Prieto et al. [95] compressed PFC by applying Re-Pair to the internal strings.[1] The authors used a public implementation of Re-Pair based on the original one [107]. Therefore, the compression rate was very high, but so was construction cost.

On the other hand, our strategy replaces internal strings with integer IDs using dictionary encoding. The bottom of FIGURE 4.2 shows an example. Our front coding dictionary consists of three arrays: H stores the header strings with a special terminator \$, P stores the header initial addresses, and L interleaves the shared lengths and the IDs.

[1] Although they also proposed a header-compressed version namely HTFC, we intend to compare Re-Pair compression with dictionary encoding; therefore, we do not evaluate header compression.

4.2 Auxiliary String Dictinoaries

To avoid confusion, we refer to a string dictionary used for dictionary compression as an *auxiliary string dictionary*. As described in Section 4.1, it encodes strings appearing in the PDT and front coding dictionaries into integer IDs. Note that we do not restrict the integer range of the IDs. The auxiliary string dictionary supports the following operations:

- Extract(i) returns the string with ID i.
- Compare(i, w) returns the comparison result between strings Extract(i) and w.

Although Extract is the same as Access, we redefine it to avoid notational confusion. Compare is called during Lookup. It is always supported when Extract is supported; however, Compare can stop decoding when a mismatch occurs. The auxiliary string dictionary does not need string search operations such as Lookup because its role is to decode the stored strings.

In this section, we propose several auxiliary string dictionaries, considering the following:

- For each Lookup or Access, Extract and Compare are called multiple times; therefore, decoding speed is especially important.
- As tries and front coding merge prefixes, the likelihood that the target strings for compression have many similar suffixes is high; therefore, we implement auxiliary string dictionaries by merging the suffixes.

We show examples for each auxiliary string dictionary using strings `ch`, `faggy`, `ide`, `ie`, `nology`, `ology`, and `rie` mapped to IDs `A`, `B`,..., and `G`, respectively.

4.2.1 Plain Concatenation and Sharing

The simplest data structure to implement an auxiliary string dictionary concatenates the strings with a terminator. Each starting address is obtained as an ID. When a string is included in the suffix of another, the suffix can be shared. Such an array is generally called *TAIL* [145, 70]. Figure 4.3 shows an example of TAIL. Strings `ie` and `ology` are shared by `rie` and `nology`, respectively. Compared to other auxiliary string dictionaries described below, its compression rate is low but its decoding speed is the fastest.

4.2.2 Reverse Trie

A *reverse trie* is constructed by merging suffixes rather than prefixes. The root corresponds to string terminations. The strings are decoded by traversing nodes to the root. That is

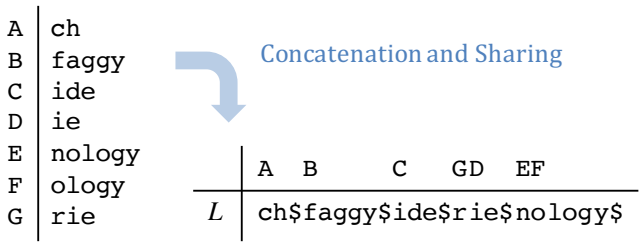
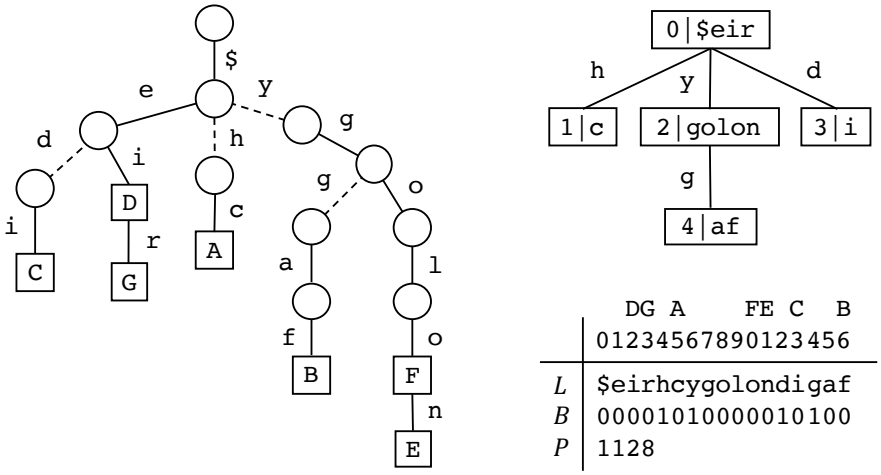


FIGURE 4.3: Auxiliary string dictionary using TAIL.



(A) Reverse trie (B) RPDT

FIGURE 4.4: Auxiliary string dictionaries using a reverse trie and RPDT.

to say, we can perform EXTRACT and COMPARE by maintaining the starting node IDs. FIGURE 4.4a shows an example of a reverse trie. For the purpose of illustration, the reverse trie has a super root with a special terminator $.

Several dictionary structures using reverse tries have been proposed [4, 139] and implemented as open-source software, such as the UX [113] and MARISA [140] libraries. These structures implement reverse tries using the double array and LOUDS. The double array is a pointer-based data structure that can provide the fastest trie representation; however, its space efficiency is low. LOUDS is a succinct tree that can construct very small dictionaries; however, its node-to-node traversal is slow.

To solve the trade-off problem, the use of path decomposition is a workable alternative, but the existing representation [57] cannot immediately detect mismatches in COMPARE because the node label must be scanned from the head. Therefore, we propose a novel reverse trie representation with path decomposition, namely the *reverse path-decomposed trie (RPDT)*.

Reverse Path-decomposed Trie An implementation of RPDT is simpler than that in [57] because the reverse trie for auxiliary string dictionaries does not require finding

children. FIGURE 4.4b shows an example of the RPDT constructed from the reverse trie of FIGURE 4.4a. The example RPDT is constructed by applying centroid path decomposition to the reverse trie and assigning node IDs in breadth-first order. The node labels do not contain the special characters $\mathbf{1}, \mathbf{2}, \ldots$ because it is not necessary to find children.

To represent the RPDT, we use three sequences: L stores strings obtained by concatenating pairs of branching characters and node labels in order of node ID, B is a bit sequence such that $B[i] = 1$ if $L[i]$ stores a branching character, and P stores addresses of L where each edge branches off in order of node ID. As P becomes a non-decreasing sequence, we can use the Elias-Fano representation. We perform EXTRACT on the sequences as in Algorithm 4.1.

Algorithm 4.1 EXTRACT(i) in RPDT

```
1: Initialize x to an empty string
2: while L[i] ≠ $ do
3:     x ← x + L[i]
4:     if B[i] = 1 then
5:         i ← P[RANK_1(B, i)]
6:     else
7:         i ← i − 1
8:     end if
9: end while
10: return x
```

Theoretical Analysis We assume that the number of nodes in a reverse trie is n. L and B use $n\lceil \log \sigma \rceil$ bits and $n + o(n)$ bits, respectively. P uses $2m + m\lceil \log \frac{n}{m} \rceil + o(m)$ bits, where $|P| = m$. A succinct tree uses $2n + n\lceil \log \sigma \rceil + o(n)$ bits to represent the reverse trie. In roughly $2m + \lceil \log \frac{n}{m} \rceil < n$, the RPDT representation is smaller than the succinct tree representations. Note that m is the number of leaves in the reverse trie minus one. As shown in FIGURE 4.4, m becomes considerably smaller than n. Therefore, its space efficiency can outperform that of the succinct tree representation. Moreover, its cache efficiency is high because of centroid path decomposition.

4.2.3 Back Coding

We implement an auxiliary string dictionary by applying front coding to suffixes. In this paper, we refer to the technique as *Back Coding*. The left part of FIGURE 4.5 shows an example of Back Coding. In the same manner as the Front Coding dictionaries, we divide strings into buckets of size b and encode them from each header. Since auxiliary string dictionaries need fast operations, we implement the Back Coding dictionary using PFC.

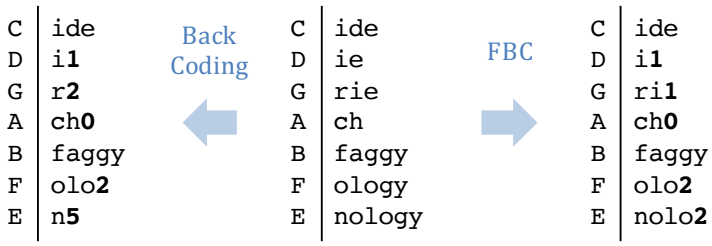


FIGURE 4.5: Auxiliary string dictionaries using Back Coding with $b = 4$.

Faster Decodable Implementation To support faster decoding, we introduce an alternative implementation using the differences among headers instead of predecessors [102]. We refer to the technique as *Fast Back Coding (FBC)*. The right part of FIGURE 4.5 shows an example of FBC. The technique does not need to decode internal strings other than the target string. In other words, the maximum number of memory copies for each EXTRACT or COMPARE is 2. However, some space efficiency is sacrificed because the number of shared characters decreases.

4.3 Experimental Evaluation

This section analyzes the practical performance of string dictionaries compressed by the proposed dictionary encoding on real-world datasets.

4.3.1 Settings

We carried out the experiments on Intel Xeon E5540 @2.53 GHz, with 32 GiB of RAM (L2 cache: 1 MiB; L3 cache: 8 MiB), running Ubuntu Server 16.04 LTS. The data structures were implemented in C++ and compiled using g++ (version 5.4.0) with optimization -O9. The runtimes were measured using `std::chrono::duration_cast`.

Data Structures We applied the auxiliary string dictionaries described in Section 4.2 (i.e., TAIL, RPDT, back coding, and FBC) to the string dictionaries described in Section 4.1 (i.e., PDT and Front Coding). For Back Coding and FBC, we tested two bucket sizes of 4 and 8. The dictionaries are referred to as *BC4*, *BC8*, *FBC4*, and *FBC8*.

We also evaluated Re-Pair compression. Although some Re-Pair implementations are available, we used those used by each proponent described in Section 4.1. Note that we can choose other lightweight compression techniques, such as Huffman coding [66] and online grammar compression [96, 97]; however, Re-Pair is the most popular compression tool at present because its compression rate is very high and its decoding speed is fast.

To implement PDT and front-coding dictionaries, we used the *path_decomposed_tries* [116] and *libCSD* [93], respectively. We set the bucket size of front coding to 8 based on

TABLE 4.1: Information about datasets.

	Size	Strings	Ave. length	Chars
GEONAMES	101.2	6,784,722	15.6	96
NWC	439.4	20,722,756	22.2	180
ENWIKI	227.2	11,519,354	20.7	199
INDOCHINA	612.9	7,414,866	86.7	98
UK	2,723.3	39,459,925	72.4	103
DBPEDIA	3,326.5	64,626,232	54.0	95

past experiments [57, 8, 70]. To implement the basic tools described in Section 2.2, we used the *Succinct* library [56].

Datasets We used the following real-world datasets:

- GEONAMES: Geographic names on the *asciiname* column from GeoNames dump [51].
- NWC: Japanese word N-grams in the Nihongo Web Corpus 2010 [141].
- ENWIKI: All page titles from the English Wikipedia of Feb. 2015 [133].
- INDOCHINA: URLs of a 2004 crawl by the UbiCrawler [22] on the country domains of Indochina [82].
- UK: URLs of a 2005 crawl by the UbiCrawler [22] on the `.uk` domain [82].
- DBPEDIA: URIs extracted from the dataset generated by the DBpedia SPARQL Benchmark [101], obtained from [98, DS2].

TABLE 4.1 summarizes relevant statistics for each dataset, where *Size* is the total length of strings (i.e., the raw size) in MiB, *Strings* is the number of different strings, *Ave. length* is the average number of characters per string, and *Chars* is the number of different characters in the dataset.

TABLE 4.2 summarizes statistics of target strings of the auxiliary string dictionaries, where *Size* is the total length of strings in MiB, *Strings (before)* is the number of strings, *Strings (after)* is the number of strings obtained by removing duplication (i.e., the string set size), *Reduction* is its reduction rate in percentage, *Ave. length* is the average number of characters per string in the string set, and *Ave. calls* is the average number of COMPARE calls for each LOOKUP (or EXTRACT calls for each ACCESS). From the table, we can considerably reduce the number of strings by removing duplication. In front coding, the average number of calls is essentially 3.5 because the number of internal strings for each bucket is 7.

TABLE 4.2: Information about strings.

	Size (MiB)	Strings (before)	Strings (after)	Reduction (%)	Ave. length	Ave. calls
GEONAMES						
PDT	36.4	6,784,722	1,670,795	24.6	11.5	6.16
Front coding	34.6	5,936,631	1,354,818	22.8	10.9	3.50
NWC						
PDT	77.8	20,722,756	1,873,988	9.0	18.5	6.59
Front coding	69.6	18,132,411	202,058	1.1	8.5	3.50
ENWIKI						
PDT	90.3	11,519,354	3,762,531	32.7	14.4	6.26
Front coding	83.0	10,079,434	2,921,168	29.0	13.5	3.50
INDOCHINA						
PDT	134.6	7,414,866	1,299,775	17.5	44.7	6.61
Front coding	121.6	6,488,007	1,204,600	18.6	42.5	3.50
UK						
PDT	655.4	39,459,925	11,957,102	30.3	35.2	6.96
Front coding	591.7	34,527,434	10,756,301	31.2	34.2	3.50
DBPEDIA						
PDT	1423.3	64,626,232	12,199,056	18.9	30.1	7.41
Front coding	1274.8	56,547,953	9,969,214	17.6	30.3	3.50

4.3.2 Results

Tables 4.3 and 4.4 show the experimental results for the construction time in seconds (*Constr*), percentage of compression ratio between the data structure and the raw data size (*Cmpr*), and average running times of LOOKUP and ACCESS in microseconds (*Lookup* and *Access*). The top two results are highlighted. To measure the running times of LOOKUP, we chose one million random strings from each dataset. The running times of ACCESS were measured for one million IDs corresponding to the random strings. Each test was averaged over 10 runs.

Results for PDT (TABLE 4.3) All auxiliary string dictionaries yield short construction times. Compared to Re-Pair, our strategy provides up to 8.2x faster construction. The compression rate of Re-Pair is the lowest, except for INDOCHINA and UK. In INDOCHINA and UK, BC8 and RPDT construct slightly smaller dictionaries. For the runtimes of LOOKUP and ACCESS, TAIL is the fastest in all cases. Compared to Re-Pair, TAIL respectively provides up to 1.7x and 1.5x speed up on LOOKUP and ACCESS; however, its compression rate is the worst.

TABLE 4.3: Results of PDT.

	GEONAMES				NWC			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	26.6	**31.5**	2.17	2.11	58.0	**16.9**	2.73	2.72
TAIL	**4.0**	44.4	**1.51**	**1.54**	**11.0**	26.7	**2.10**	**2.04**
RPDT	5.8	**34.9**	**1.92**	**1.90**	16.0	23.0	2.68	**2.53**
BC4	5.4	38.5	3.51	3.66	14.2	22.9	5.14	5.11
BC8	**5.2**	36.6	3.91	4.05	14.8	**22.2**	5.58	5.80
FBC4	5.3	39.1	1.93	2.12	14.2	23.2	**2.48**	2.62
FBC8	5.3	38.0	2.11	2.11	**14.1**	22.8	2.72	2.90
	ENWIKI				INDOCHINA			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	62.1	**31.6**	2.59	2.50	55.0	11.8	3.70	3.61
TAIL	**8.3**	41.7	**1.72**	**1.76**	**7.8**	13.5	**2.19**	**2.40**
RPDT	11.8	**32.4**	2.40	**2.26**	9.3	**10.7**	3.32	3.38
BC4	10.6	36.0	4.56	4.66	**8.6**	10.9	6.41	6.91
BC8	**10.5**	33.8	5.13	5.33	**8.6**	**10.3**	7.50	8.05
FBC4	10.6	37.0	**2.29**	2.38	**8.6**	11.3	**2.74**	**3.03**
FBC8	10.6	35.9	2.49	2.62	**8.6**	11.2	3.20	3.44
	UK				DBPEDIA			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	437.2	17.5	4.00	3.98	798.9	**14.7**	2.30	**2.32**
TAIL	55.7	20.7	**2.59**	**2.94**	**102.8**	18.5	**1.59**	**1.82**
RPDT	58.6	**16.4**	4.16	3.91	158.3	15.8	2.40	2.52
BC4	63.4	16.9	7.02	7.60	124.8	15.9	6.26	6.68
BC8	**53.3**	**15.9**	7.93	8.45	**122.1**	**15.2**	6.69	7.16
FBC4	**53.5**	17.3	**3.24**	**3.55**	133.2	16.2	**2.17**	2.41
FBC8	**53.5**	16.9	3.66	3.91	132.8	15.9	2.56	2.78

Overall, RPDT and FBC appear to be good data structures taking into account all aspects. RPDT altogether outperforms Re-Pair on INDOCHINA. FBC altogether outperforms Re-Pair on INDOCHINA and UK. In the other datasets, RPDT and FBC provide much shorter construction times than Re-Pair with the competitive compression rates and operation times. Back coding is compact but requires long operation times. If fast LOOKUP and ACCESS operations are needed, TAIL becomes a viable alternative.

Results for Front Coding (TABLE 4.4) Since this Re-Pair implementation is based on the original, its compression rates are better than those of the PDT results, but its construction and decoding costs are higher. On UK and DBPEDIA, Re-Pair respectively takes approximately 3.5 and 3 hours because the size of the target strings is very large. These times are impractical. On the other hand, all auxiliary string dictionaries provide practically short construction times as well as satisfactory PDT results. When comparing

Table 4.4: Results of Front Coding.

	GEONAMES				NWC			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	83.6	**38.4**	2.09	1.25	289.2	**25.4**	2.16	1.09
TAIL	**2.7**	48.3	**1.24**	**0.53**	**7.7**	28.3	**1.47**	**0.50**
RPDT	4.7	**41.6**	1.65	0.85	14.2	27.7	**1.64**	**0.58**
BC4	**4.3**	44.5	1.80	0.98	13.8	27.4	1.84	0.87
BC8	4.4	43.0	2.07	1.15	13.8	**27.3**	1.95	0.90
FBC4	4.5	45.0	**1.57**	**0.82**	**13.7**	27.4	1.68	0.70
FBC8	**4.3**	44.2	1.68	0.93	**13.7**	27.4	1.82	0.85
	ENWIKI				INDOCHINA			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	667.2	**36.5**	2.63	1.59	2042.8	**18.2**	3.88	2.25
TAIL	**6.1**	45.3	**1.51**	**0.66**	**4.3**	25.0	**2.03**	**0.71**
RPDT	9.6	**38.7**	2.23	1.21	6.7	22.3	2.60	1.17
BC4	**8.6**	41.9	2.16	1.19	**5.9**	22.5	2.71	1.32
BC8	10.0	40.2	2.29	1.33	**5.9**	**22.0**	3.05	1.65
FBC4	**8.6**	42.7	**2.00**	**1.09**	**5.9**	22.9	**2.45**	**1.14**
FBC8	8.8	42.0	2.04	1.13	6.1	22.7	2.73	1.30
	UK				DBPEDIA			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	11835.2	**22.3**	4.28	2.41	10774.6	**22.6**	4.27	2.85
TAIL	**28.0**	31.4	**2.32**	**0.77**	**55.3**	28.6	**1.53**	**0.74**
RPDT	50.0	27.1	3.64	1.73	123.1	27.0	**2.02**	**1.28**
BC4	**42.6**	27.7	3.18	1.59	94.0	27.0	2.17	1.40
BC8	43.4	**26.8**	3.60	1.97	92.2	**26.4**	2.41	1.61
FBC4	43.3	28.1	**3.02**	**1.47**	**91.6**	27.3	2.05	1.34
FBC8	43.4	27.7	3.04	1.51	96.6	27.0	2.16	1.45

TAIL and Re-Pair, the differences are from 30.8x to 422.5x. In terms of compression rate, Re-Pair is the smallest in all cases. Compared to the second smallest dictionaries, Re-Pair is up to 4% smaller. However, its Lookup and Access times are slower.

Overall, our dictionaries are competitive with the Re-Pair dictionaries except in terms of construction time. Further, our construction times are very short and practical. Back coding is not very slow compared to the PDT results because the number of calls is small. TAIL can provide very fast Lookup and Access operations as well as PDT results.

Chapter 5

Dynamic Path-Decomposed Tries

This chapter discusses dynamic string dictionaries, whereas Chapters 3 and 4 presented static ones. As described in the previous chapters, a number of compressed static data structures for implementing string dictionaries in much compact space were developed inspired by the motivation presented in [26]. On the other hand, dynamic dictionaries do not still achieve high space efficiency compared to the static ones, although existing data structures, such as Judy [130] and HAT-trie [10], attempt to improve upon the space efficiency by reducing pointer overheads.

Related to this topic, we mention also about dynamic compact trie representations. Several works [6, 68, 104, 127] provided asymptotic worst-case results for dynamic tries. For example, Arroyuelo et al. [6] introduced succinct representations for dynamic tries on n nodes, requiring almost optimal $2n + n \log \sigma + o(n \log \sigma)$ bits, while supporting trie operations in $\mathcal{O}(1)$ time if $\sigma = \text{polylog}(n)$. However, they are not in practice. Darragh et al. [36] proposed the *Bonsai* tree, which is a practical trie representation based on compact hash tables. Recently, Poyias et al. [124, 123] proposed new practical dynamic tries by improving the Bonsai tree, namely *m-Bonsai*. It can represent a dynamic trie in $\mathcal{O}(n \log \sigma)$ bits, while supporting trie operations in $\mathcal{O}(1)$ expected time. Although both the representations are compact and practical, there has been no discussion or evaluation about string dictionary implementation.

Other interests are the *packed compact trie* [131] and the *wavelet trie* [58]. The former is an efficient data structure that dynamically stores a set of strings for online string precessing; however, its empirical results are provided without memory usage measurements. The latter is a data structure for a sequence of strings. It can be dynamically implemented in theory, but we do not aware of any implementation of a dynamic wavelet trie. Consequently, we must address the engineering of more space-efficient dynamic dictionary structures in practice.

In this chapter, we propose a novel space-efficient dynamic string dictionary, namely the *dynamic path-decomposed trie (DynPDT)*. Our data structure is based on a trie formed

by *path decomposition* [46], which is a trie transformation technique. The path decomposition was proposed for constructing cache-friendly trie structures and was utilized in static applications [57, 64]; however, we use it for dynamic dictionary construction with a different approach. We implement space-efficient dictionaries by applying the m-Bonsai representation to this approach. From experiments using read-world datasets considering various applications, we show that our implementation is much more compact than existing dynamic ones.

5.1 m-Bonsai

The succinct trees can represent a trie in optimal space, but they are limited to static implementation in practice. For practical compact dynamic tries, [36] proposed the *Bonsai* data structure that represents a trie using the compact hash table [30]. Bonsai locates trie nodes on the table using open address hashing. Recently, Poyias et al. [124, 123] proposed the improved version, namely *m-Bonsai*. We describe the m-Bonsai tree as follows.

Let n denote the number of trie nodes. We refer to $\alpha = n/m$ $(0 \leq \alpha \leq 1)$ as a *load factor*. We assume that n and m are pre-given. Since m-Bonsai locates each node at some slot, node IDs are assigned using slot addresses. That is, a node with ID s (or node s) is located on $Q[s]$. Note that the node locations (or IDs) do not change with time (unless the hash table is not rebuilt).

Defining a new child from node s with symbol c is implemented in three steps. The first step creates a hash key $\langle s, c\rangle$ of the child. The second step obtains an *initial address* of the child, $i = h(\langle s, c\rangle)$, where h is a hush function such that $h : \{0, \ldots, m\sigma - 1\} \rightarrow \{0, \ldots, m - 1\}$. The third step locates the child on $Q[t]$, where t is the first empty slot address from the initial address i using linear probing. In other words, the new child ID is defined as t.

For representation of the hash table, if we simply store the hash key $\langle s, c\rangle$ in $Q[t]$ to check for membership, then each slot uses $\log m + \log \sigma$ bits. However, this space is asymptotically the same as that of pointer-based representations. m-Bonsai reduces this space by using the *quotienting* technique [79, Exercise 6.13] as follows. The technique designs the hash function h as $h(k) = (ak \bmod p) \bmod m$, where p is the first prime such that $p > m\sigma$, and a is a multiplier such that $1 \leq a < p$. Then, Q only keeps the *quotient* value $q(k) = \lfloor (ak \bmod p)/m \rfloor$ corresponding to k. Given $h(k)$ and $q(k)$, we can reconstruct k to check for membership. In other words, m-Bonsai stores $q(\langle s, c\rangle)$ to $Q[t]$, probed from the initial address $h(\langle s, c\rangle)$. Since $q(\langle s, c\rangle) \leq 2\sigma$ obviously, each slot takes only $\log \sigma + \mathcal{O}(1)$ bits.

In addition, m-Bonsai introduces a *displacement* array D of length m such that $D[t]$ stores the distance of t and $h(\langle s, c\rangle)$ (i.e., the number of collisions), because we need to

know the corresponding initial address $h(\langle s, c\rangle)$ for any $Q[t] = q(\langle s, c\rangle)$. From the pair $Q[t]$ and $D[t]$, we can obtain both the initial address and its quotient to reconstruct the original hash key. Although the maximum value of D is m, the array can be represented in compact space since the average value is very small from [123, Proposition 5]. Poyias et al. [123] showed how D can be represented with a data structure using $\mathcal{O}(m)$ bits and constant amortized-time operations, by using CDRW arrays [122]. Consequently, m-Bonsai can represent a dynamic trie in $m(\log \sigma + \mathcal{O}(1)) = \mathcal{O}(n \log \sigma)$ bits, while supporting tree operations in $\mathcal{O}(1)$ expected time. As α is some constant parameter (where $0 < \alpha < 1$), this space usage closes to the information-theoretic space lower bound of a trie, $n \log \sigma + \mathcal{O}(n)$ bits [17].

5.1.1 Practical Implementation

Actually, the above representation of D is very complex and the tree operations are slow in practice. Poyias et al. [123] therefore proposed a practical alternate representation of D. The practical version represents D using three-layered data structures: an array of fixed-size small entries, a compact hash table [30], and a plain associative array. It tries to store displacement values to the data structures from the top level layer (i.e., from the array of fixed-size entries). However, it is difficult to estimate the predefined length of the compact hash table when adopted to our data structure. We simply modifies the practical version for our data structure.

In our representation, we first try to store values of D in an array D_0 with each entry using Δ_0 bits. If $D[i] < 2^{\Delta_0} - 1$, we set $D_0[i] = D[i]$. Otherwise, we set $D_0[i] = 2^{\Delta_0} - 1$ and store $D[i]$ to an auxiliary associative array implemented using a standard data structure, such as `std::map`. In other words, our version omits the compact hash table.

Parameter Test We tested the simple version to search the parameter Δ_0 that implements the smallest m-Bosnai structure, as a preliminary experiment. We built the m-Bosnai tries for a million strings sampled from GEONAMES in Section 5.3.1, and measured the required memory using the `/usr/bin/time` command. We tested parameters $4 \leq \Delta_0 \leq 10$ on $\alpha = 0.8$. FIGURE 5.1 shows the result. From the result, we can see that the best parameter is $\Delta_0 = 6$. We will set this parameter in Section 5.3.1. The source code of the preliminary experiment is available at https://github.com/kampersanda/bonsais.

5.2 Dynamic Path-Decomposed Trie

This section presents a new dynamic string dictionary through path decomposition, namely, the *dynamic path-decomposed trie (DynPDT)*.

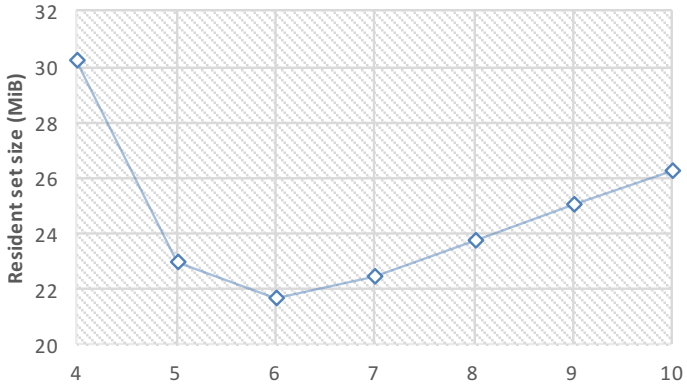


FIGURE 5.1: Result of the preliminary experiment for Δ_0.

5.2.1 Basic Idea: Incremental Path Decomposition

We present a basic idea called *incremental path decomposition*. This idea constructs a path-decomposed trie by incrementally defining nodes corresponding to each registered string in insertion order. Note that we append a special terminal character `$` to every string in order to give a bijection between nodes and strings. That is, the number of nodes of the resulting tree is the same as the number of strings. We show the data structure of the path-decomposed trie while describing the insertion procedure for a query string x as follows:

- If the dictionary is empty, a root labeled with string x is defined. In this paper, we denote such a label on node s by L_s.
- If the dictionary is not empty, a search starts from the root with two steps, by setting s to the root ID. The first step compares x with L_s. If $x = L_s$, then the procedure terminates because the query is already registered; otherwise, the second step finds a child with symbol $\langle i, x[i]\rangle$ such that $x[i] \neq L_s[i]$ and $x[0, i) = L_s[0, i)$. If not found, the query is inserted by adding a new edge labeled with the symbol $\langle i, x[i]\rangle$ and a new child labeled with the remaining suffix $x[i + 1, |x|)$. If found, the procedure returns to the first step after updating s to the child ID and x to the remaining suffix.

The path-decomposed trie has node labels representing some suffixes of strings. A search is also performed by that procedure. The feature of the incremental path decomposition is to locate nodes corresponding to early inserted strings near the root. In other words, the search cost for such strings is low. However, Section 5.3 evaluates the performance of DynPDT for random-ordered strings without considering the feature.

Example 5.1. FIGURE 5.2 *shows insertion process for strings* $x_1 =$ `technology$` *and* $x_2 =$ `technics$`. *For the first query* x_1, *we simply define the root* s_1 *and attach the string*

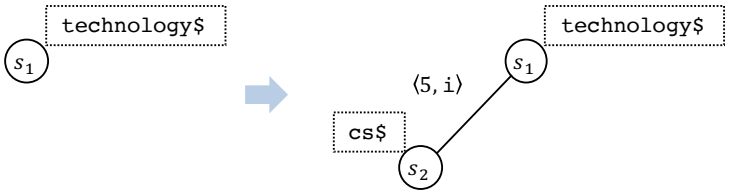


FIGURE 5.2: Incremental path decomposition.

*x_1 to it as L_{s_1} = **technology$** because the dictionary is empty. For the second query x_2, we first compare x_2 with L_{s_1}. As $x_2[0,5) = L_{s_1}[0,5) =$ **techn**, we create a symbol $\langle 5, x[5]\rangle = \langle 5, \mathtt{i}\rangle$ and add a child s_2 indicated with the symbol. We terminate this process by attaching the remaining suffix $x_2[6, |x_2|) =$ **cs$** to the child.*

5.2.2 Implementation with m-Bonsai

To obtain high space efficiency, DynPDT represents the path-decomposed trie using m-Bonsai; however, this representation has the following problem. As the edge label is a pair $\langle i, c\rangle$ composed of node label position i and character c, the edge labels are drawn from an alphabet of size $\sigma' = \sigma \cdot \Lambda$, where Λ denotes the maximum length of the node labels. The m-Bonsai representation requires the fixing of the σ' parameter to predefine the allocation size of each Q slot and the hash function, but Λ, or σ', is an unfixed parameter in dynamic applications registering unknown strings.

To solve this problem, we forcibly fix the alphabet size as $\sigma' = \sigma \cdot \lambda$ by introducing a new parameter λ. If position i on L_s is greater than or equal to λ, we create virtual nodes called *step nodes* with a special symbol ϕ by repeating to add child t from node s with symbol ϕ, to set s to t, and to decrement i by λ, until $i < \lambda$. This solution creates additional step nodes depending on λ. When λ is too small, many step nodes are created. When λ is too large, the space usage of Q becomes large because each slot uses $\lceil \log \sigma' \rceil = \lceil \log(\sigma \cdot \lambda) \rceil$ bits. Therefore, it is necessary to define a proper λ. Section 5.3.1 shows such parameters obtained from experiments using read-world datasets.

Example 5.2. FIGURE 5.3 *shows an example of DynPDT constructed by inserting strings **technology$**, **technics$**, **technique$**, **technically$**, and **technological$** in this order, setting $\lambda = 8$. The nodes are defined in order of $s_1, s_2, \ldots, s_6$. We show how to search **technically$** using the example. First, we set x to the query string and compare x with L_{x_1}. As $x[0,5) = L_{s_1}[0,5) =$ **techn**, we move to s_2 using symbol $\langle 5, x[5]\rangle = \langle 5, \mathtt{i}\rangle$ and update x to the remaining suffix **cally$**. Next, we compare x with L_{s_2}. As $x[0,1) = L_{s_2}[0,1) =$ **c**, we move to s_4 using symbol $\langle 1, x[1]\rangle = \langle 1, \mathtt{a}\rangle$ and update x to the remaining suffix **lly$**. Finally, we can see that the query string is registered from $x = L_{s_4}$.*

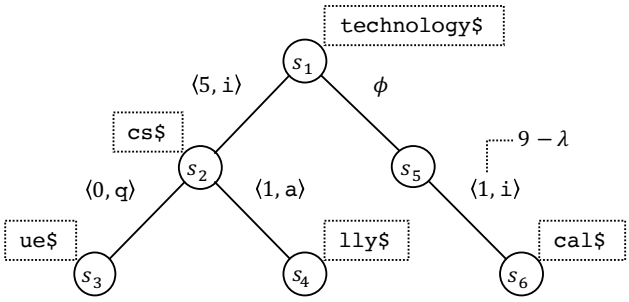


FIGURE 5.3: DynPDT when $\lambda = 8$.

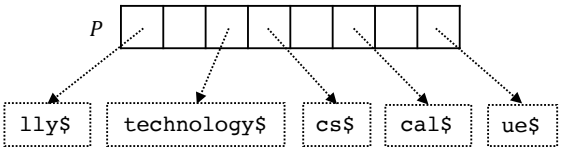
We also show how to search ***technological$****. In the same manner as above, we set x to the query string and compare x with L_{s_1}. The result is $x[0, 9) = L_{s_1}[0, 9) =$* ***technolog****, but we cannot create symbol $\langle 9, \mathtt{i} \rangle$ because this symbol exceeds the alphabet size from $\lambda \leq 9$. Therefore, we move to step node s_5 using symbol ϕ. From $9 - \lambda < \lambda$, i.e., $1 < \lambda$, we can create symbol $\langle 1, \mathtt{i} \rangle$ and move to node s_6 using the symbol. Finally, we can see that the query string is registered because the remaining suffix* ***cal$*** *is the same as L_{s_6}.*

Remarks Arbitrary values associated with each string can be maintained using the space of each node label. Deletion can be simply implemented by introducing flags for each node (i.e., for each string) in a manner similar to closed hash tables. In other words, we add a bit array of length m such that s-th bit is `1` iff the string corresponding to node s is deleted. However, this implementation includes two problems. One uses additional m bits. The other is that the node label corresponding to the deleted string remains after deletion, because the label may be a part of other strings. Therefore, DynPDT has the disadvantage for deletion.

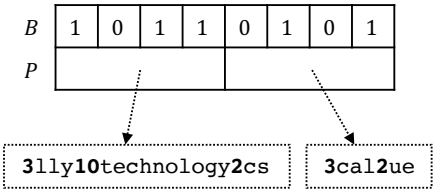
To use the m-Bonsai representation, it is necessary to predefine the number of Q slots depending on the number of nodes and the load factor. In other words, it is necessary to estimate the number of nodes expected for a dataset. Fortunately, we can roughly estimate the number of nodes of DynPDT easier than a plain trie because this is the same as the number of strings and some step nodes depending on a proper λ.

5.2.3 Node Label Management

The node labels are stored separately from the m-Bonsai structure (i.e., the hash table and the displacement array) because these labels are variable-length strings. The plainest implementation uses pointer array P of length m such that $P[i]$ stores a pointer to L_i. This implementation can perform to access and append a node label in constant time, but it uses large space with m pointers, or mw bits, where w denotes the word length. We call this implementation *plain management*. FIGURE 5.4a shows an example of plain management.



(A) Plain management



(B) Bitmap management in $\ell = 4$

FIGURE 5.4: Node label management for DynPDT in FIGURE 5.3.

We present an alternative compact implementation that reduces the pointer overhead in a manner similar to *sparsetable* of Google Sparse Hash [54]. This implementation divides node labels into *groups* of ℓ labels over the IDs. That is, the first group consists of $L_0 \dots L_{\ell-1}$, the second group consists of $L_\ell \dots L_{2\ell-1}$, and so on. Moreover, we introduce bitmap B such that $B[i] = \texttt{1}$ iff L_i exists. The implementation concatenates node labels L_i such that $B[i] = \texttt{1}$ in each group, while keeping the ID order. The length of P becomes $\lceil m/\ell \rceil$ by maintaining pointers to the concatenated label strings for each group. We call this implementation *bitmap management*.

Using array P and bitmap B, accessing L_i is performed as follows. If $B[i] = \texttt{0}$, L_i does not exist; otherwise, we obtain the target concatenated label string from $P[g]$, where $g = \lfloor i/\ell \rfloor$. We also obtain bit chunk $B_g = B[g \cdot \ell, (g+1) \cdot \ell)$ over the target group. Let j be the number of occurrences of `1`s in $B_g[0, i \bmod \ell + 1)$. L_i is the j-th node label of the concatenated label string. As ℓ is constant, counting the bit occurrences in chunk B_g is supported in constant time using the *popcount* operation [53]. Therefore, the access time is the same as the time of scanning the concatenated label string until the j-th node label.

By simply concatenating node labels (e.g., the second group in FIGURE 5.4a is `cal$ue$` in $\ell = 4$), the scan is performed by sequentially counting terminators in $\mathcal{O}(\ell \cdot \Lambda)$ time, where Λ again denotes the maximum length of the node labels. We shorten the scan time using the *skipping* technique used in *array hashing* [12]. This technique puts its length in front of each node label using the VByte encoding [134]. Note that we can omit the terminators of each node label. The skipping technique allows us to jump ahead to the start of the next node label; therefore, the scan is supported in $\mathcal{O}(\ell)$ time. FIGURE 5.4b shows an example of the bitmap management with the skipping technique.

For space usage, the P uses $w\lceil m/\ell \rceil$ bits and the B uses m bits. Let $\ell = \Theta(w)$, the total space usage becomes $\mathcal{O}(m)$ bits, which are smaller than mw bits of the plain management; however, a node label can be accessed in $\mathcal{O}(w)$ time. The other interest is the total length

of node labels using VByte encoding. The skipping technique can omit the terminators of each node label, but writes byte codes indicating the lengths instead. Here, we assume that node labels are practically represented as byte strings. Let node labels be practically represented as byte strings, there is no overhead from the VByte encoding if all node labels are shorter than 128, because such a code length is one byte. Fortunately, almost 100% of the node labels were shorter than 128 in all datasets in Section 5.3; therefore, the VByte encoding does not become a significant overhead.

5.3 Experimental Evaluation

This section analyzes the practical performance of DynPDT. The source code of our implementation is available at `https://github.com/kampersanda/dynpdt`.

5.3.1 Settings

We performed experiments on an Intel Xeon E5540 @2.53 GHz with 32 GB of RAM (L2 cache: 1 MB, L3 cache: 8 MB), running Ubuntu Server 16.04 LTS. The data structures were implemented in C++ and compiled by g++ (version 5.4.0) with optimization -O9. We used `/proc/<PID>/statm` to measure the maximum resident set size. We used `std::chrono::duration_cast` to measure the runtimes of operations.

Datasets We selected six real-world datasets:

- GEONAMES: Geographic names on the *asciiname* column from GeoNames dump [51].
- ENWIKI: All page titles from the English Wikipedia of Feb. 2015 [133].
- UK: URLs of a 2005 crawl by the UbiCrawler [22] on the `.uk` domain [82].
- WEBBASE: URLs of a 2001 crawl performed by the WebBase crawler [63, 82].
- LUBM: URIs extracted from the dataset generated by the Lehigh University Benchmark [59] for 1,600 universities, obtained from [98, DS5].
- DNA: All substrings of 12 characters found in the Gene DNA data set from Pizza&Chili corpus [121].

TABLE 5.1 summarizes relevant statistics for each dataset, where *Size* is the total length of strings in MiB, *Strings* is the number of distinct strings (or $|\mathcal{X}|$), *Nodes* is the number of nodes in a plain trie, *NPK* is the average number of plain trie nodes per key string, *BPK* is the average number of bytes per key string, and *BPNL* is the average number of bytes per node label in DynPDT.

TABLE 5.1: Information about datasets.

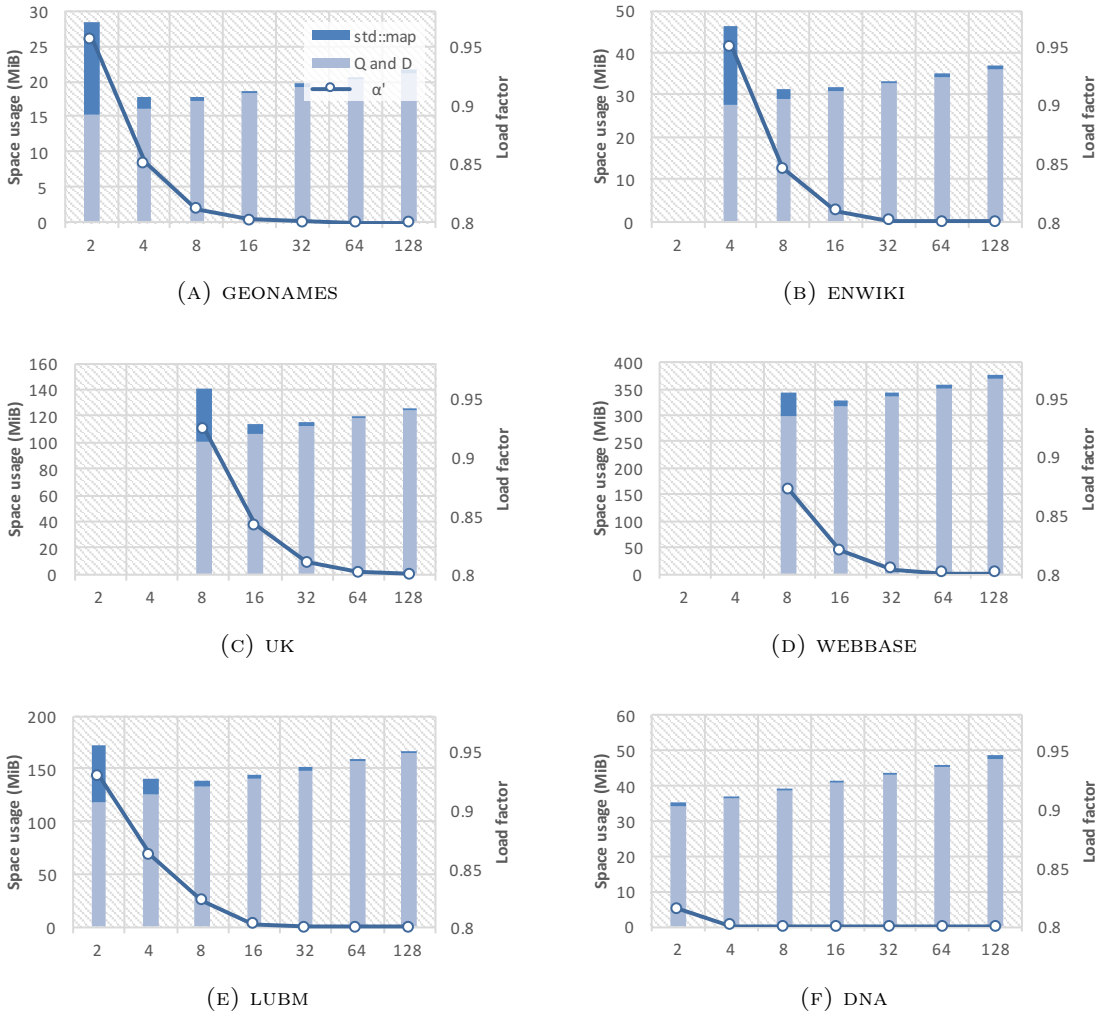
	Size	Strings	Nodes	NPK	BPK	BPNL
GEONAMES	101.2	6,784,722	48,240,884	7.1	15.6	10.1
ENWIKI	227.2	11,519,354	110,962,030	9.6	20.7	12.6
UK	2,723.3	39,459,925	748,571,709	19.0	72.4	22.0
WEBBASE	6,782.1	118,142,155	1,426,314,849	12.1	60.2	15.1
LUBM	3,194.1	52,616,588	247,740,552	4.7	63.7	7.7
DNA	189.3	15,265,943	36,223,473	2.4	13.0	5.4

Data Structures We compared the performance of DynPDT with that of m-Bonsai. For DynPDT, we tested plain and bitmap management denoted by *Plain* and *Bitmap-ℓ*, respectively. For Bitmap-ℓ, we considered that ℓ is 8, 16, 32, and 64. We set the `int` data type to associated values in DynPDT. We tested m-Bonsai based on a plain trie without maintaining associated values. For both DynPDT and m-Bonsai, we implemented the auxiliary associative array using `std::map`.

We also compared some existing dynamic dictionaries. We selected five space-efficient ones as follows: *Sparsehash* is Google Sparse Hash that is an associative array with keys and values of arbitrary data types [54], *Judy* is a trie implementation developed at Hewlett-Packard Research Labs [130], *HAT-trie* is a string dictionary implementation with the combination of a trie and a cache-conscious hash table [10, 69], *ART* is a trie implementation designed for efficient main-memory database systems [84, 35], and *Cedar* is a state-of-the-art double-array prefix trie implementation [149]. In common with DynPDT, we set the `int` data type to associated values. As Cedar uses 32-bit integers to represent trie nodes, we could not run the test on WEBBASE.

Parameters Both DynPDT and m-Bonsai have two parameters α and Δ_0. We set $\alpha = 0.8$ in common with previous settings [36, 123]. We set the number of Q slots to that of strings divided by 0.8 in DynPDT. Note that the resulting load factor α' in DynPDT is increased from α, depending on the number of step nodes. In m-Bonsai, we set the number of Q slots to that of plain trie nodes divided by 0.8. Note that the difference of the number of Q slots between DynPDT and m-Bonsai closes with decreasing NPK. We set $\Delta_0 = 6$ from the preliminary experiments in Section 5.1.

DynPDT also has parameter λ, which involves α' and the space usage of hash table Q and the auxiliary associative array. When λ is large, the allocation size of Q becomes large. When λ is small, the number of step nodes, or α', is increased. The latter poses slow operations and a large auxiliary associative array because the average value of D is increased. To search a proper λ, we pretested $\lambda = 2^a$ in $2 \leq a \leq 7$ for each dataset. FIGURE 5.5 shows the results. The figures show the sum space usage of Q, D, and the



(A) GEONAMES (B) ENWIKI

(C) UK (D) WEBBASE

(E) LUBM (F) DNA

FIGURE 5.5: Results of parameter test for each $\lambda \in [2, 128]$.

auxiliary `std::map`. The parameter α' is also shown. We could not construct the dictionary for $\lambda = 2$ on ENWIKI because α' became too large. The results of UK and WEBBASE are also the same. From the results, α' closes 0.8, and the space usage is moderately increased from some λ. In the experiments, we chose the smallest λ such that $\alpha' \leq 0.81$ for each dataset. We set λ to 16, 16, 64, 32, 16, and 4 on GEONAMES, ENWIKI, UK, WEBBASE, LUBM, and DNA, respectively.

5.3.2 Results

We constructed the dictionaries by inserting strings in random order. We measured the resident set size required for the construction. We measured the insertion and search runtimes without I/O overheads. The insertion time was averaged on 3 runs. To measure

TABLE 5.2: Results of space usage in bytes per key string.

	GEONAMES	ENWIKI	UK	WEBBASE	LUBM	DNA
Plain	46.0	46.6	54.4	47.5	45.0	44.8
Bitmap-8	18.7	21.2	31.3	24.0	15.5	13.0
Bitmap-16	16.8	18.8	28.2	21.0	13.8	11.0
Bitmap-32	15.0	17.4	27.1	19.8	12.1	9.8
Bitmap-64	14.5	16.9	26.4	19.2	11.5	9.0
m-Bonsai	17.7	23.6	46.1	29.3	11.4	5.9
Sparsehash	62.3	71.1	131.0	119.0	122.0	43.4
Judy	47.6	50.5	60.3	53.5	33.9	24.3
HAT-trie	35.4	40.2	82.3	68.9	64.7	28.9
ART	87.1	93.1	140.9	126.9	118.9	71.1
Cedar	30.5	41.1	58.4	–	29.7	22.1

the search time, we chose 1 million random strings from each dataset. The search time was averaged on 10 runs.

Space Usage TABLE 5.2 shows the results. It is obvious that bitmap management can reduce the pointer overhead of plain management. Bitmap-64 is up to 5x smaller than Plain on DNA. When BPNL is small, the compression rate is high based on the comparison analysis in Section 5.2.3. Compared with m-Bonsai, Bitmap-64 is 1.2–1.7x smaller except for LUBM and DNA. Bitmap-64 is 1.5x larger on DNA because the difference of the Q lengths is small from NPK. Note that m-Bonsai did not maintain associated values of the `int` type. If m-Bonsai maintained those values ideally without any overhead, 4-bytes space (i.e., `sizeof(int)`) is added per string. That is, Bitmap-64 becomes smaller than m-Bonsai on all the datasets. In the existing dictionaries, Cedar is basically small although 32-bit integers are used to represent node pointers. In ENWIKI and WEBBASE, HAT-trie and Judy are the smallest. Compared with the smallest existing dictionaries, Bitmap-64 is 2.1–2.8x smaller. On UK and WEBBASE whose BPNL is large, Plain is also smaller than the existing dictionaries because the pointer overhead is relatively small over the overall space usage.

Insertion Time TABLE 5.3 shows the results. In DynPDT, Plain is the fastest and Bitmap-64 is the slowest essentially, but Bitmap-8 is not much slower than Plain. We compare Bitmap-8 for the following. Compared with m-Bonsai, Bitmap-8 is faster except for DNA. In particular, the difference is very large on datasets whose BPK is large owing to the path decomposition. Bitmap-8 is 3.6x, 2.9x and 2.5x faster than m-Bonsai on UK, WEBBASE, and LUBM, respectively. Bitmap-8 is 1.5x slower than m-Bonsai on DNA. Compared with the existing dictionaries, Bitmap-8 is not the fastest but is very competitive

Table 5.3: Results of insertion time in microseconds per key string.

	GEONAMES	ENWIKI	UK	WEBBASE	LUBM	DNA
Plain	1.00	1.14	1.65	2.37	1.65	1.35
Bitmap-8	1.25	1.38	1.99	2.64	1.91	1.58
Bitmap-16	1.37	1.57	2.29	2.93	1.99	1.66
Bitmap-32	1.69	1.93	2.91	3.47	2.29	1.91
Bitmap-64	2.13	2.65	4.12	4.60	2.87	2.30
m-Bonsai	1.62	2.22	7.13	7.69	4.80	1.03
Sparsehash	4.31	5.15	9.13	11.32	8.72	1.99
Judy	0.93	1.06	2.15	2.94	1.53	0.90
HAT-trie	0.96	1.13	1.63	1.75	2.58	0.84
ART	1.07	1.19	2.20	2.98	1.44	0.87
Cedar	1.05	1.07	2.56	–	2.50	0.90

Table 5.4: Results of search time in microseconds per key string.

	GEONAMES	ENWIKI	UK	WEBBASE	LUBM	DNA
Plain	1.01	1.13	1.53	2.20	1.12	1.08
Bitmap-8	1.22	1.38	2.15	2.40	1.26	1.26
Bitmap-16	1.38	1.61	2.47	2.74	1.43	1.38
Bitmap-32	1.71	2.06	3.25	3.72	1.61	1.83
Bitmap-64	2.31	3.01	4.88	5.29	2.16	2.18
m-Bonsai	1.47	2.06	6.69	8.30	3.08	0.86
Sparsehash	0.34	0.44	0.67	0.80	0.67	0.29
Judy	0.70	0.88	2.02	2.42	0.79	0.44
HAT-trie	0.31	0.35	0.61	0.80	0.51	0.22
ART	0.81	1.03	1.84	2.68	0.67	0.63
Cedar	0.42	0.69	2.51	–	0.69	0.22

on UK, WEBBASE, and LUBM; however, Bitmap-8 is the slowest on the other datasets except for Sparsehash.

Search Time Table 5.4 shows the results. Like the insertion time results, Plain is the fastest in DynPDT, and Bitmap-8 is faster than m-Bonsai except for DNA. On the other hand, DynPDT is basically slower compared with the existing dictionaries. Bitmap-8 is up to 5.7x slower than the fastest HAT-trie. On UK and WEBBASE, Bitmap-8 is close to Judy, ART, and Cedar.

Chapter 6

Practical Rearrangement of Dynamic Double-Array Tries

In this chapter, we consider rearrangement approaches of dynamic double-array dictionaries. The original double-array trie was proposed to be used in semi-static digital search [2]; therefore, its update time was slow and unstable. However, a number of studies have investigated fast update algorithms and techniques [99, 105, 129, 147]. A recent double-array trie implementation [148] enables update times that are close to those of a hashing dictionaries such as the C++ standard library's `std::unordered_map`. Consequently, there have been some applications using dynamic double-array tries such as a full-text search engine [23] and text-stream processing [149].

On the other hand, the original double array has another problem about space efficiency. As described in Section 3.1, the double-array structure can include empty elements like hash tables. The space efficiency depends on the *load factor* (i.e., the proportion of nonempty elements to the total elements). The load factor genearlly does not become problematic in static construction; however, dynamic construction leads to reduction of it, especially in key deletion. Some works [40, 99, 115, 143] attempted to solve the problem, and enabled to maintain a high load factor.

Here, we describe how the methods related to the second problem still 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 double array structure with arbitrary timing, simply applying these existing methods requires a significant amount of time. Therefore, we propose several rearrangement methods and evaluate their practical performance through experiments using real-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.

6.1 Dynamic Double-Array Dictionaries

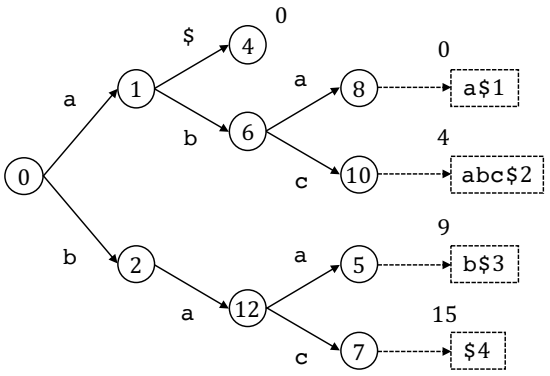
We implement dynamic double-array dictionaries using the MP-trie, based on Equation 3.2. The MP-trie is the version of using *terminal characters*. The dictionary is composed of BASE, CHECK, LEAF and TAIL arrays, as shown in Section 3.1.3. Note that TERM is not needed because the leaves always correspond to the terminal nodes. In TAIL, common suffixes of the separated strings are not shared because the structure is frequently changed during updates.

We give some definitions related to empty elements in the double-array structure. We define a bit array EMPTY such that $\text{EMPTY}[s] = 1$ iff element s is empty. As empty elements arise in the internal BASE and CHECK elements, $\text{EMPTY}[0] = \text{EMPTY}[N-1] = 0$ 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 $\alpha = n/N$ $(0 \leq \alpha \leq 1)$. Also in TAIL, useless spaces can arise 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 $0 \leq \beta \leq 1$.

Example 6.1. FIGURE 6.1 *shows an example of an MP-trie dictionary and the DA representation for key-value pairs* $(\texttt{a\$}, 0)$, $(\texttt{abaa\$}, 1)$, $(\texttt{abcabc\$}, 2)$, $(\texttt{baab\$}, 3)$, *and* $(\texttt{bac\$}, 4)$ *from the alphabet* $\{\texttt{\$} = 0, \texttt{a} = 1, \texttt{b} = 2, \texttt{c} = 3\}$. *Note that* CHILD *and* SIB *are introduced in Section 6.1.1. In* FIGURE 6.1, *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 6.1.1.*

The example dictionary embeds associated values in a fixed-length space on TAIL after the terminator, e.g., $\text{TAIL}[11] = 3$. If a string is registered without TAIL, the associated value is embedded in the corresponding leaf BASE element, e.g., $\text{BASE}[4] = 0$. Embedding is the most practical dictionary implementation using dynamic double-array tries. This is because node addresses change frequently during updates. Briefly, the three basic dictionary operations are implemented as follows. SEARCH is implemented by traversing root-to-leaf nodes and by checking TAIL. INSERT 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. DELETE is implemented by removing the nodes that correspond to the key.

Example 6.2. $\text{SEARCH}(\texttt{abcabc\$}) = 2$ *is performed in* FIGURE 6.1, *as follows. First, child 1 of root 0 is found as* $\text{BASE}[0] \oplus \texttt{a} = 0 \oplus 1 = 1$ *and* $\text{CHECK}[1] = 0$. *Similarly, nodes 6 and 10 are found as follows:* $\text{BASE}[1] \oplus \texttt{b} = 4 \oplus 2 = 6$, $\text{CHECK}[6] = 1$, $\text{BASE}[6] \oplus \texttt{c} = 9 \oplus 3 = 10$, *and* $\text{CHECK}[10] = 6$. *Equation* $\text{LEAF}[10] = 1$ *means that the other characters can be found*



	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 6.1: MP-trie dictionary and the corresponding double-array representation.

in TAIL *starting from* BASE[10] = 4. *From* TAIL[4..7] = **abc$**, *the key is registered and the value* TAIL[8] = 2 *is returned.*

INSERT(**abccb$**, 5) *is performed in* FIGURE 6.1, *as follows. A part of the resulting dictionary is shown in* FIGURE 6.2. *First, new branches labeled* **a** *and* **c** *from node 10 are added. Then, a new leaf node labeled* **a** *is defined and linked to* TAIL[5]. *Consequently,* TAIL[4] *becomes empty. At the same time, another new leaf node labeled* **c** *is also defined. The suffix* **b$** *and value 5 are appended to the end of* TAIL, *and the node forms a link to* TAIL[17].

DELETE(**abcabc$**) *is performed in* FIGURE 6.1, *as follows. A part of the resulting dictionary is shown in* FIGURE 6.3. *First, node 10 corresponding to the key is removed by emptying the element. At the same time, node 6 becomes a new leaf for key* **abaa#** *as the structure of the minimal-prefix keys is changed. The new suffix* **aa$** *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.*

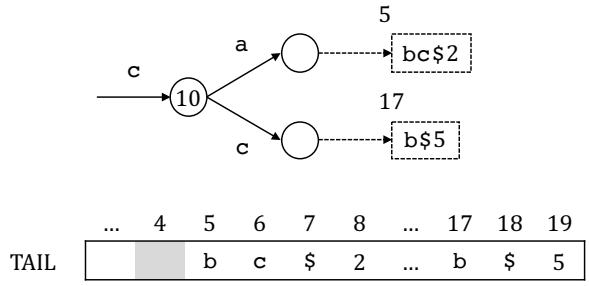


FIGURE 6.2: Result of INSERT(abccb$, 5) in the dictionary shown in FIGURE 6.1.

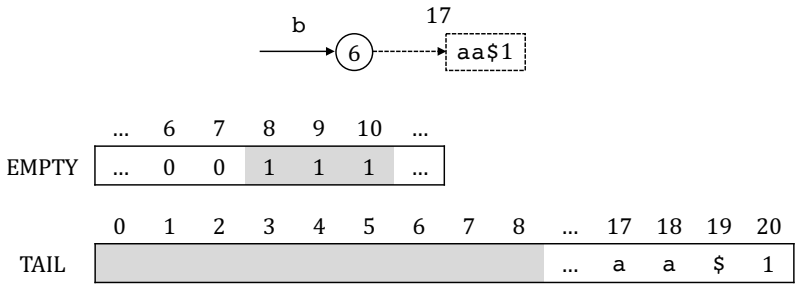


FIGURE 6.3: Result of DELETE(abcabc$) in the dictionary shown in FIGURE 6.1.

6.1.1 Fast Update Methods

Two bottlenecks are typically observed when updating double-array 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 double array [2] performs this search by scanning BASE and CHECK elements linearly in $\mathcal{O}(N)$ time; however, this time turns into a critical problem for a large dictionary. Therefore, general implementations use the *empty-link method (ELM)* [99, 143]. 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:

$$\text{BASE}[r_i] = \begin{cases} r_{i+1} & (1 \le i < m) \\ r_1 & (i = m) \end{cases} \text{ and } \text{CHECK}[r_i] = \begin{cases} r_{i-1} & (1 < i \le m) \\ r_m & (i = 1) \end{cases}.$$

In other words, the successor and predecessor of r_i are obtained from $\text{BASE}[r_i]$ and $\text{CHECK}[r_i]$, respectively. The ELM can scan R in $\mathcal{O}(m)$ time when the first empty element is maintained as the list head. For example, in FIGURE 6.1, empty elements 3, 9, and 11 are

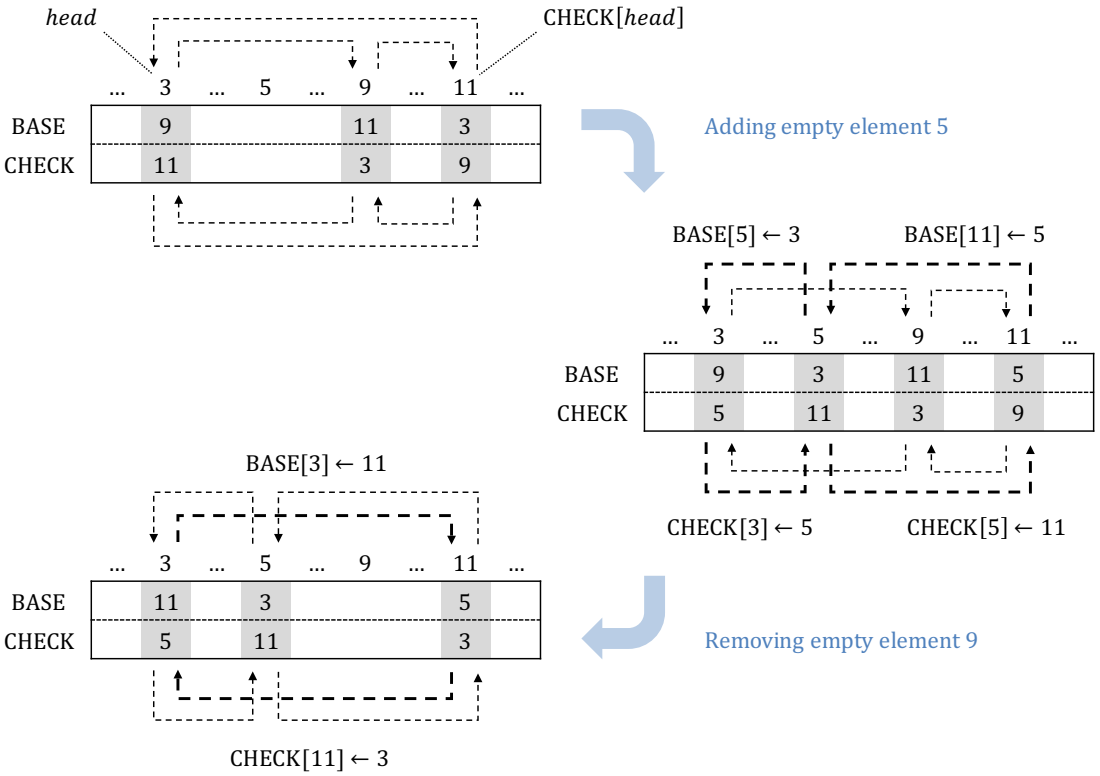


FIGURE 6.4: Illustration of updates with the ELM.

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 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.

Example 6.3. FIGURE 6.4 *shows an example of updating a double array 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* CHECK[*head*]. *When a new empty element 5 is added, it is inserted between the first and last elements as* BASE[5] $\leftarrow$ 3, CHECK[3] $\leftarrow$ 5, BASE[11] $\leftarrow$ 5, *and* 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* CHECK[9] = 3 *and the next element* BASE[9] = 11 *are re-chained as* BASE[3] $\leftarrow$ 11 *and* CHECK[11] $\leftarrow$ 3. *The updates are performed in constant time.*

Node-Link Method Let EDGES(s) be an operation returning a set of edge labels from node s. For example, EDGES(1) = {$, b} in FIGURE 6.1. A simple implementation performs EDGES(s) in $\mathcal{O}(\sigma)$ 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)* [147]. NLM uses additional CHILD and SIB arrays such that CHILD[s] stores the edge label between node s and its first child, and 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 BASE[s] $\oplus$ CHILD[s] and BASE[CHECK[s]] $\oplus$ SIB[s], respectively. In other words, $E \leftarrow$ EDGES(s) is performed in $\mathcal{O}(|E|)$ time. 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.

Example 6.4. FIGURE 6.1 *shows* CHILD *and* SIB *for the trie.* EDGES(1) = {$, b} *is performed as follows. The first edge label* $ *is given by* CHILD[1] = $*; the first child 4 is calculated by* BASE[1] $\oplus$ CHILD[1] = $4 \oplus$ $ = 4*; the second edge label* b *is given by* SIB[4] = b*; and the second child 6 is calculated by* BASE[1] $\oplus$ SIB[4] = $4 \oplus$ b = 6*. It is evident that node 6 is the last child from* SIB[6] = $ = CHILD[1].

6.1.2 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. [99] 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. [115] 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. [143] presented an adaptive method that uses elements with fewer siblings than the rearmost element as well as empty and single

Algorithm 6.1 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 6.2 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
```

elements. Available literature [143] 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 6.1. PACK searches the relocation addresses of the compression elements using EXCHECK, which is shown in Algorithm 6.2. EXCHECK then returns the BASE value indicating the addresses. Note that EXCHECK in Algorithm 6.2 uses ELM. Other operations used in PACK and EXCHECK are described as follows.

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

Essentially, PACK improves the load factor α by repeating the following steps. EXCHECK obtains the relocation addresses of the compression elements; SHELTER solves

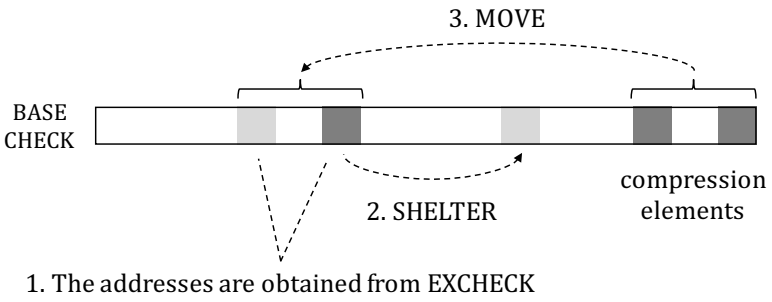


FIGURE 6.5: Illustration of steps of PACK.

collisions during relocation; and MOVE eliminates empty elements by relocating the compression elements. These steps are depicted in FIGURE 6.5. 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 [143] for more details.

TAIL Dorji et al. [40] 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. 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 $k-1$ elements until an empty space is found. Here, k denotes the suffix length.

FIGURE 6.6 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.

6.2 Practical Rearrangement Methods

While the existing methods described in Section 6.1.2 can maintain high load factors α and β, these methods present practical problems related to time and functionality.

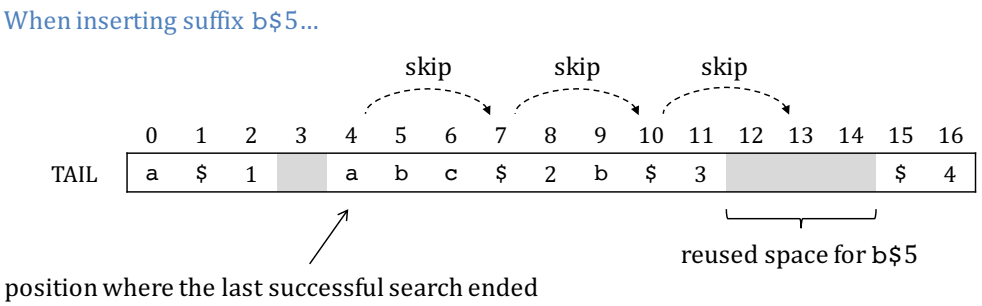


FIGURE 6.6: Illustration of Dorji's second method demonstrating the insertion of suffix `b$` and value 5.

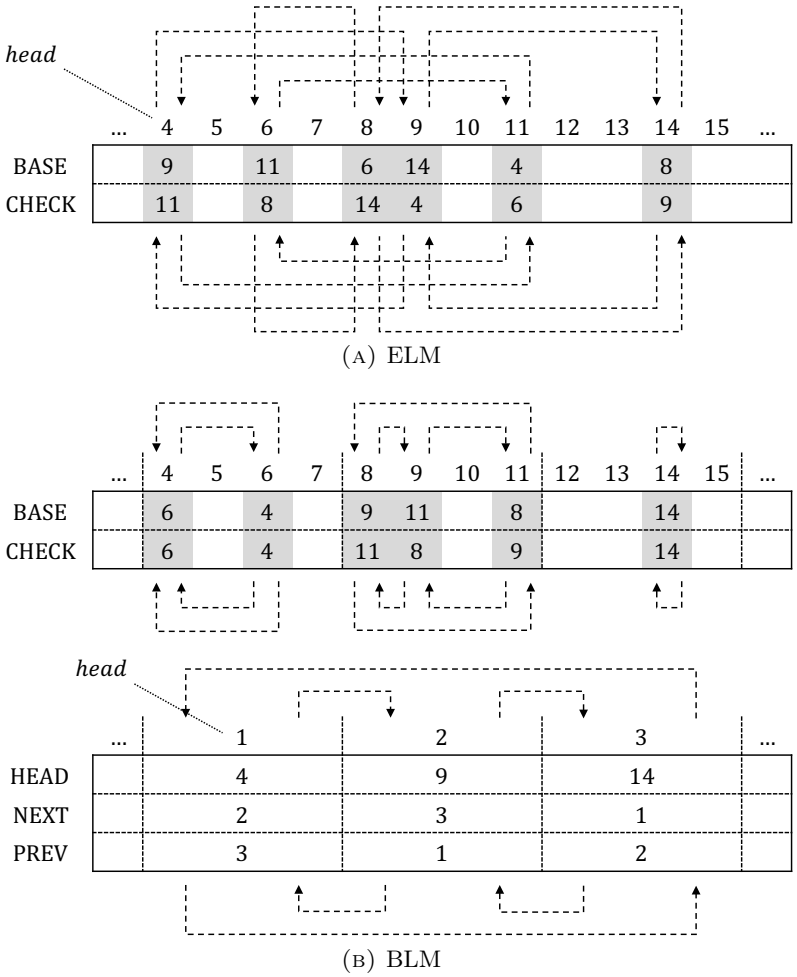
- The combination of DELETE and PACK requires significantly more time compared to simple DELETE since the algorithm is complex and involves many operations. In addition, empty elements are reused during INSERT. Therefore, maintaining the high load factor α by sacrificing DELETE time does not have many advantages.

- Dorji's first method cannot implement trie dictionaries, such as that shown in FIGURE 6.1, 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 [40] 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.

To address these problems, the simplest solution is to rearrange the double array 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 6.2.1 and 6.2.2 describe rearranging BASE and CHECK. TAIL is simply rearranged by transferring suffixes for all leaves to a new TAIL.

6.2.1 Cache-friendly implementation of the ELM

While the ELM supports scanning R in $\mathcal{O}(m)$ time, the scan order is random in BASE and CHECK, as shown in FIGURE 6.4. 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 size 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



(A) ELM

(B) BLM

FIGURE 6.7: Illustrations of the ELM and BLM.

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 6.7 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 6.3 shows EXCHECK using BLM. In Algorithm 6.3, 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.

Algorithm 6.3 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
```

For the block length, $\ell = 2^{\lceil \log \sigma \rceil}$ 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 $\lceil \log \sigma \rceil$ 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 $\lfloor r/\ell \rfloor = \lfloor base/\ell \rfloor$. ISTARGET searches relocation addresses using the $base$; thus, addresses $v \leftarrow base \oplus c$ 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 $\lfloor r_i/\ell \rfloor$ 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 6.3, 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 $\ell = 256$, 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 [147] 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 [105, 129] 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.

[1] Although the literature [147] is written in Japanese, Yoshinaga and Kitsuregawa [149] have introduced the proposed methods in English.

Algorithm 6.4 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 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, ...
```

6.2.2 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 double-array dictionary from scratch is a very simple approach. Generally, this is called *static reconstruction* since the double-array dictionary is built on the condition that all registered keys are pre-given.

Algorithm 6.4 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).

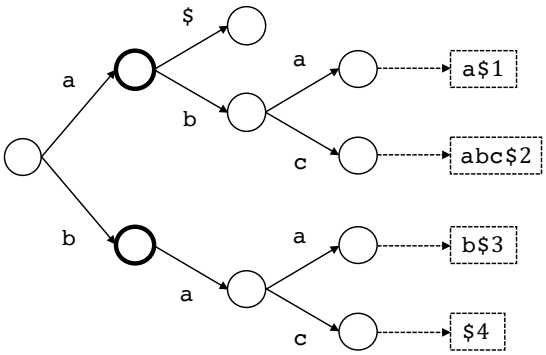
In the algorithm, node addresses are redefined in the depth-first order using a stack. Although a queue can be substituted, node addresses are redefined in the breadth-first order. Generally, such node arrangement causes frequent cache misses in node-to-node traversal near leaves. Thus, most public implementations of a static double-array 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 6.2.3.

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 [147] show that static double-array dictionaries enable faster retrieval than double-array 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 6.3.1 provides estimations of each working space.

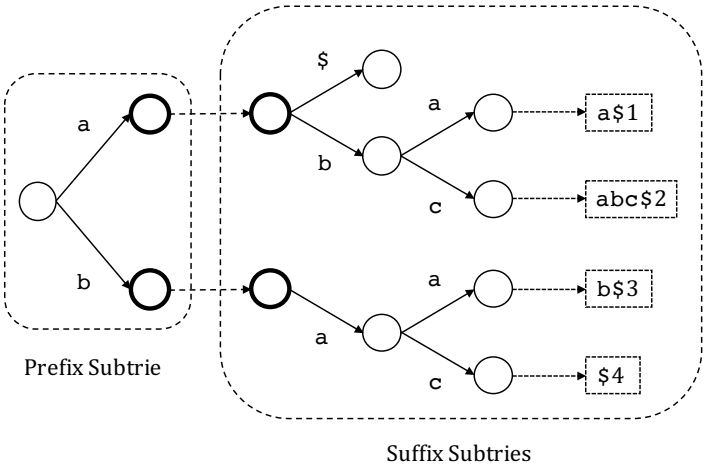
6.2.3 Concurrent Rearrangement

We can shorten rearrangement time by concurrent processing. The proposed method called the *trie division method (TDM)* enables concurrent rearrangement by dividing a trie into a *prefix subtrie* and *suffix subtries*, as shown in FIGURE 6.8. 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 6.8b. 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 6.9 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`



(A) Without the TDM



(B) With the TDM

FIGURE 6.8: Illustrations of the TDM for the MP-trie in FIGURE 6.1.

as division nodes. The TDM is similar to the technique used in the *double-array language models* [111, 137] for static tries with a large alphabet.

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 6.9, 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 6.9. 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.

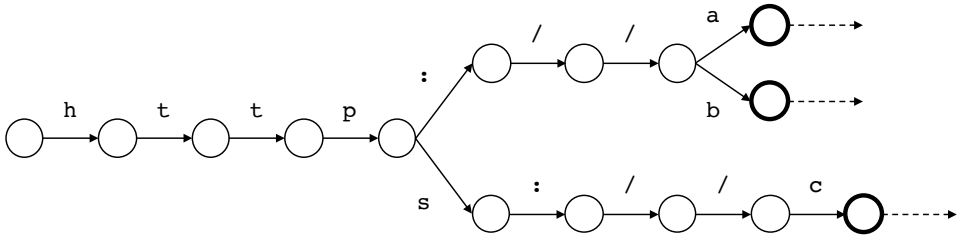


FIGURE 6.9: Illustration of a prefix subtrie preregistering `http://` and `https://`.

6.3 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/kampersanda/ddd.

6.3.1 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 GeoNames dump [51].
- ENWIKI: All page titles from the English Wikipedia of Feb. 2015 [133].
- NWC: Japanese word N-grams in the Nihongo Web Corpus 2010 [141].
- INDOCHINA: URLs of a 2004 crawl by the UbiCrawler [22] on the country domains of Indochina [82].

TABLE 6.1 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, *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.

TABLE 6.1: Information about 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 6.2: 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

Data structures For rearrangement methods, we evaluated all possible combinations shown in TABLE 6.2 (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 6.8b. TABLE 6.3 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.

TABLE 6.3: 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

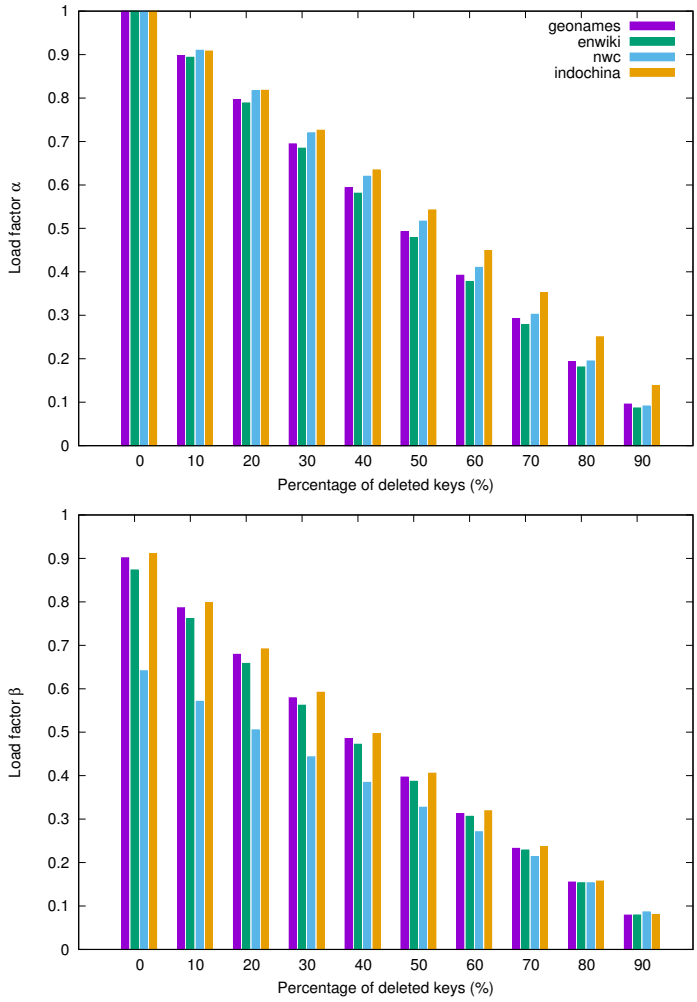


FIGURE 6.10: Load factors of PE for each dataset.

We prepared ten patterns of load factors for each dictionary by randomly inserting all keys and by randomly deleting the keys. FIGURE 6.10 shows the load factors α and β of PE for each dataset before rearrangement. The dictionaries become increasingly sparse. In addition, FIGURE 6.11 shows that the TAIL length also becomes increasingly large, similar to the example shown in FIGURE 6.3. 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 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

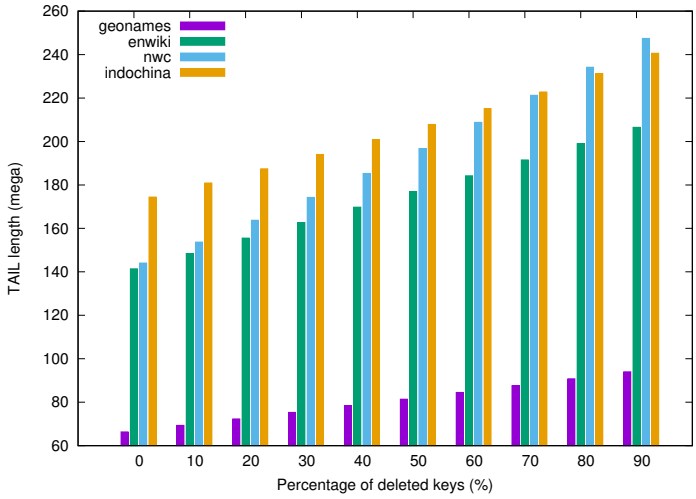


FIGURE 6.11: TAIL lengths of PE for each dataset.

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 6.3.2. PACK rearranges nodes in $\mathcal{O}(N)$ words. REBUILD rearranges nodes in $\mathcal{O}(N(1+\alpha))$ words because the original node components use $\mathcal{O}(N)$ words and the new node components use $\mathcal{O}(N \cdot \alpha)$ words. For TAIL rearrangement, both of PACK and REBUILD use $\mathcal{O}(M) + \mathcal{O}(M \cdot \beta) = \mathcal{O}(M(1+\beta))$ words. The above is summarized as follows. PACK uses $\mathcal{O}(N + M(1+\beta))$ words and REBUILD uses $\mathcal{O}(N(1+\alpha) + M(1+\beta))$ words. 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.

6.3.2 Results

Figures 6.12, 6.13, and 6.14 show the experimental results for rearrangement times, working spaces, and search times, respectively.[2] The y-axis in FIGURE 6.12 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 6.12 omits the results of combinations

[2]The figures are located at the end of this chapter.

using REBUILD and BLM (i.e., RB, RBN, RBT, and RBNT) as there were no distinct differences compared to the NLM versions. FIGURE 6.13 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 6.14 omits the results of combinations using NLM and BLM (except 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 35x, 42x, 6x, and 9x 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 260x 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 1.5x faster than PE, PBN is 2.1x faster than PEN, PBT is 1.4x faster than PET, and PBNT is 1.9x faster than PENT. These results are the outcome of the rate of single nodes. When PACK relocates a single element where $|E| = 1$ in Algorithm 6.1, EXCHECK can directly select an empty element $r \leftarrow head$ in Algorithm 6.2 or $r \leftarrow \text{HEAD}[head]$ in Algorithm 6.3 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 218x, 260x, 32x, and 3x, respectively. For NWC and INDOCHINA, the ratios are relatively small, since TDM did not work efficiently as seen from the imbalances in TABLE 6.3. 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 8.4x, 9.2x, 3.4x, and 2.4x, 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 6.10, 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 1.44x, 1.45x, 1.67x, and 1.35x 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 6.1. 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 1.04x, 1.04x, 1.05x, and 1.05x, 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 1.26x and 1.23x, respectively. When comparing RE and RET with GEONAMES and ENWIKI, TDM reduces the working spaces by up to 1.19x and 1.23x, 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 1.04x and 1.05x, respectively. When comparing RE and RET with NWC and INDOCHINA, TDM reduces the working spaces by up to 1.08x and 1.19x, 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 1.5x, 1.8x, 2.1x, and 2.9x 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 dictionary 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 strings and add `http://www` to the preregistered prefixes when storing URLs.

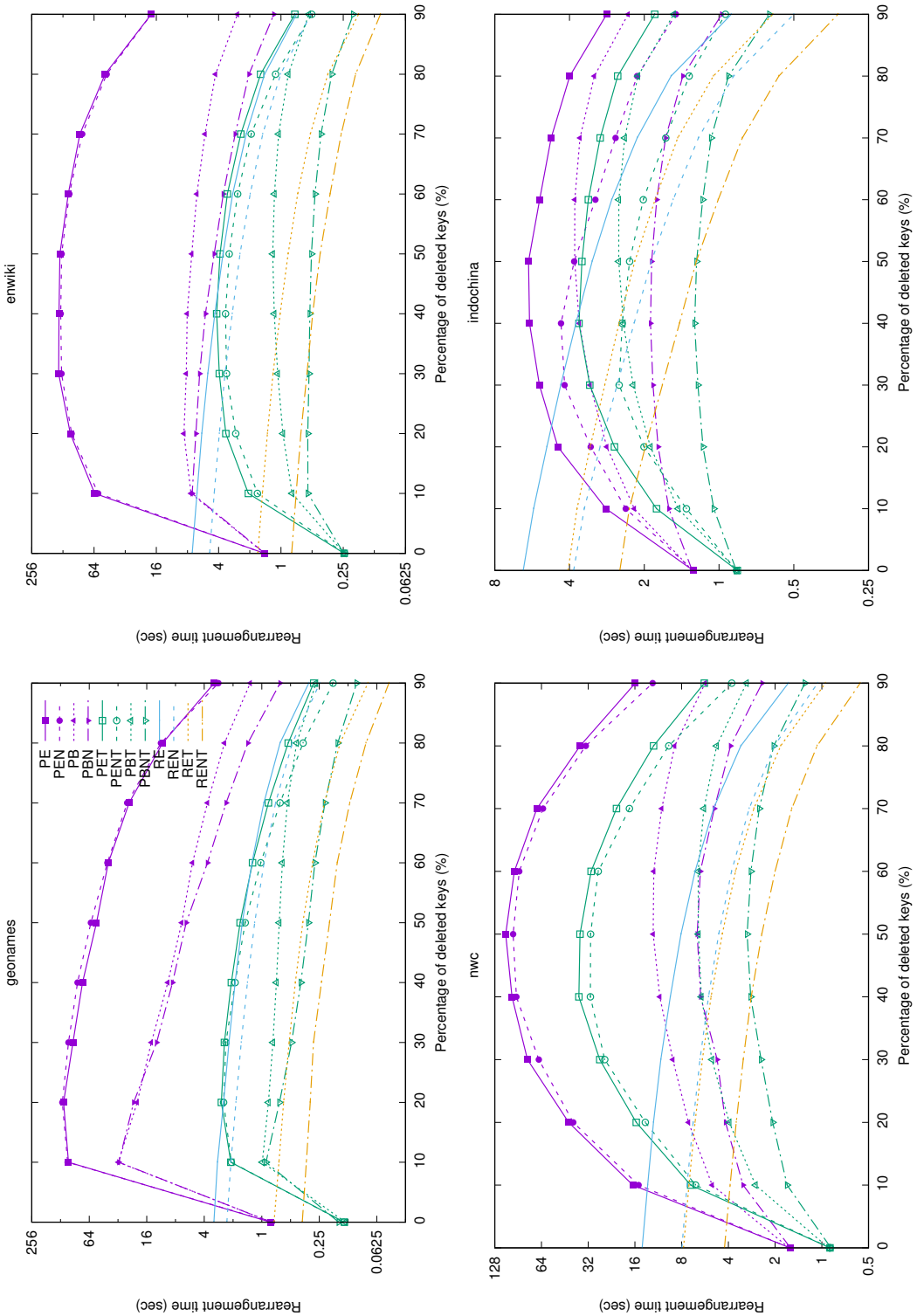


FIGURE 6.12: Experimental results of rearrangement time.

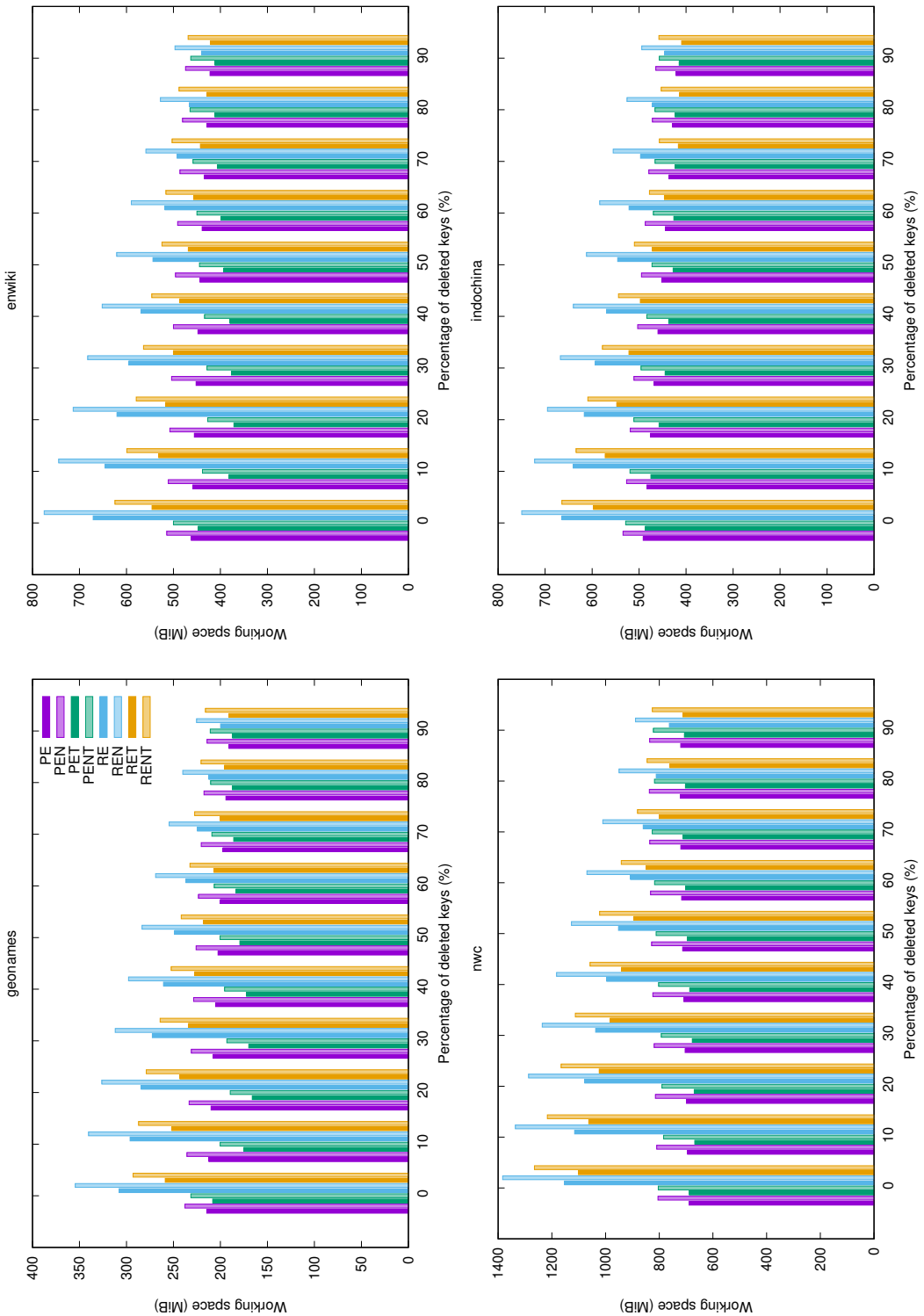


FIGURE 6.13: Experimental results of working space.

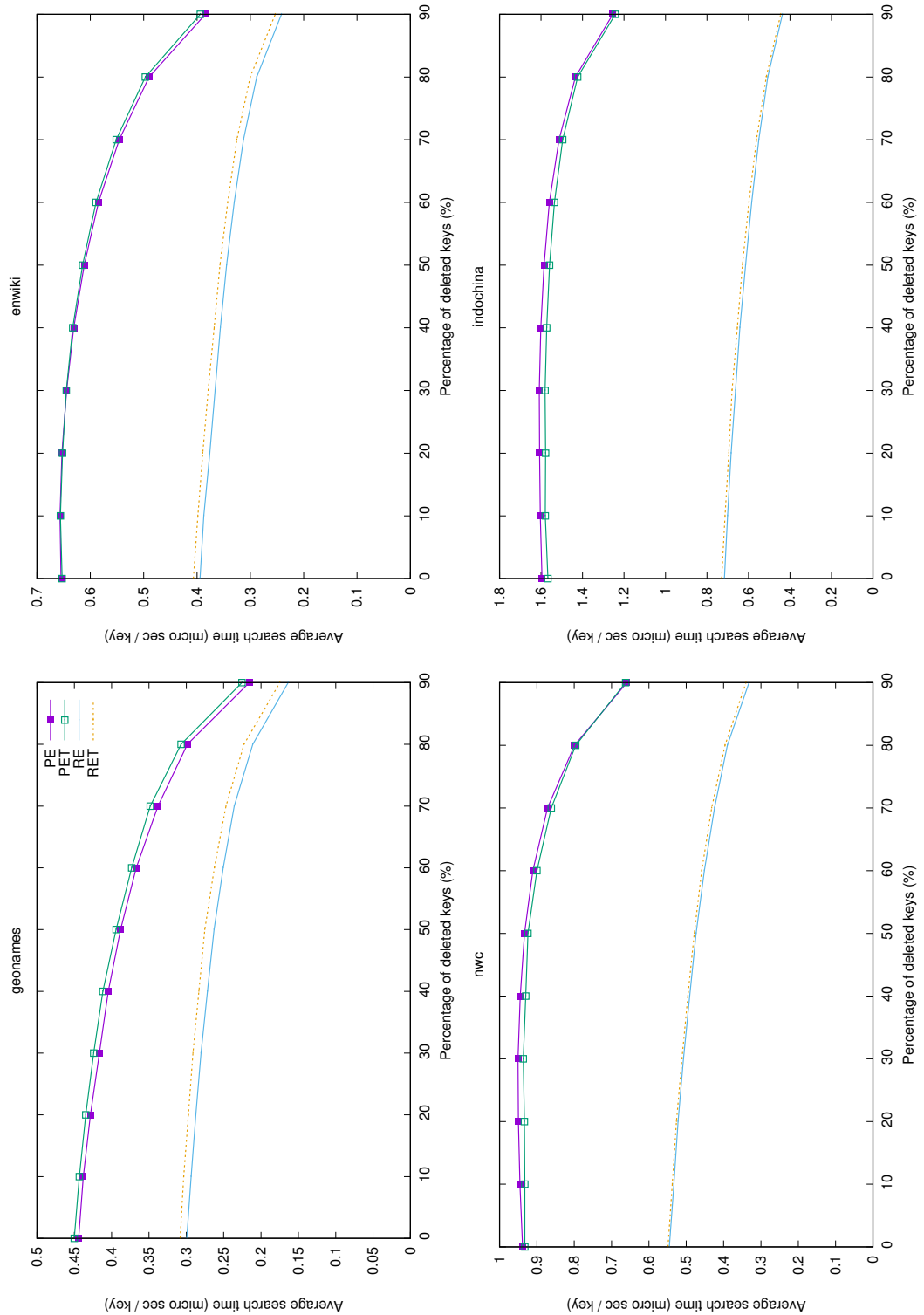


FIGURE 6.14: Experimental results of SEARCH time.

Chapter 7

Conclusion

In this thesis, we have presented novel data structures for implementing efficient string dictionaries. We conclude our work, and remark future perspectives and tasks for each chapter, as follows.

Static Compressed Double-Array Tries In Chapter 3, we introduced problems of double-array tries to implement string dictionaries for large datasets, and developed novel compressed double-array structures, named DALF and XCDA. DALF is based on CDA and compresses the BASE array using linear functions. While the original BASE array practically takes 32 or 64 bits for each element, DALF can reduce the space to 8 bits; however, DALF cannot support the ACCESS operation because of based on CDA. XCDA solves the problem with a different compression approach. It compresses the original BASE and CHECK arrays by using the XOR transformation and directly accessible codes. XCDA can implement both the LOOKUP and ACCESS operations, while experimentally achieving the same space efficiency with DALF.

Although we discussed string dictionary implementation, the double array is a versatile data structure and can be used to various applications such as multiple string matching [110, 146], lexical analyzers [87], and N-gram language models [137]. Essentially, the idea of XCDA will contribute to implementing the applications in compact space.

Static Dictionary Compression Using Dictionary Encoding In Chapter 4, we introduced alternative strategy for string dictionary compression using dictionary encoding, instead of the existing strategy with powerful text compression such as Re-Pair. Moreover, we proposed several dictionary structures for the dictionary encoding, denoted as auxiliary string dictionaries. Our strategy and dictionary structures were evaluated through experiments using real-world datasets. The results have shown that our strategy can construct high-performance string dictionaries in a short time. In particular, using the proposed auxiliary string dictionaries, named RPDT and FBC, is comprehensively superior to using Re-Pair. On the other hand, since efficient techniques of Re-Pair compression have been proposed [19, 20], the new techniques should be also evaluated in the future.

Whereas we addressed string dictionary compression in this work, our strategy and the auxiliary string dictionaries can be used to compress other data structures partly containing strings. The future tasks will investigate such data structures and conduct application experiments.

Dynamic Path-Decomposed Tries In Chapter 5, we presented DynPDT, which is a new data structure of space-efficient dynamic string dictionaries through incremental path decomposition. Our experimental results showed that DynPDT uses much smaller space than existing dynamic string dictionaries, but its time performance is not very high. The main cause is that the node-to-node traversal using m-Bonsai is slow compared with pointer-based representations. Another disadvantage is that the hash table cannot be easily resized owing to open addressing. Those disadvantages will be improved by engineering an alternative trie representation.

A remaining issue is that succinctness of DynPDT is not provided in this thesis. Unfortunately, we are not aware of any succinct dynamic string dictionary, and implementing such data structures is an open problem [117].

Practical Rearrangement of Dynamic Double-Array Tries In Chapter 6, we have discussed practical rearrangement approaches of dynamic double-array dictionaries and presented some efficient methods. Our experimental results demonstrated that the proposed methods can support much faster rearrangement compared to existing methods. Our rearrangement requires a few seconds for large datasets; thus, the time will not become a problem in practical applications.

Whereas the improvement of our methods is very large, we have not shown any contribution to real systems. The future tasks will investigate applications of the rearrangement.

Bibliography

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. "Integrating compression and execution in column-oriented database systems". In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. ACM. 2006, pp. 671–682.

[2] Jun'ichi Aoe. "An efficient digital search algorithm by using a double-array structure". In: *IEEE Transactions on Software Engineering* 15.9 (1989), pp. 1066–1077. DOI: 10.1109/32.31365.

[3] Jun'ichi Aoe, Katsushi Morimoto, and Takashi Sato. "An efficient implementation of trie structures". In: *Software: Practice and Experience* 22.9 (1992), pp. 695–721. DOI: 10.1002/spe.4380220902.

[4] Jun'ichi Aoe, Katsushi Morimoto, Masami Shishibori, and Ki-Hong Park. "A trie compaction algorithm for a large set of keys". In: *IEEE Transactions on Knowledge and Data Engineering* 8.3 (1996), pp. 476–491.

[5] Márcio Drumond Araujo, Gonzalo Navarro, and Nivio Ziviani. "Large text searching allowing errors". In: *Proceedings of the 4th South American Workshop on String Processing (WSP)*. 1997, pp. 2–24.

[6] Diego Arroyuelo, Pooya Davoodi, and Srinivasa Rao Satti. "Succinct dynamic cardinal trees". In: *Algorithmica* 74.2 (2016), pp. 742–777. DOI: 10.1007/s00453-015-9969-x.

[7] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. "Succinct trees in practice". In: *Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX)*. 2010, pp. 84–97.

[8] Julian Arz and Johannes Fischer. "LZ-compressed string dictionaries". In: *Proceedings of the Data Compression Conference (DCC)*. 2014, pp. 322–331. DOI: 10.1109/DCC.2014.36.

[9] Nikolas Askitis. "Efficient data structures for cache architectures". PhD thesis. RMIT University, 2007.

[10] Nikolas Askitis and Ranjan Sinha. "Engineering scalable, cache and space efficient tries for strings". In: *The VLDB Journal* 19.5 (2010), pp. 633–660. DOI: 10.1007/s00778-010-0183-9.

[11] Nikolas Askitis and Ranjan Sinha. "HAT-trie: A cache-conscious trie-based data structure for strings". In: *Proceedings of the 30th Australasian Conference on Computer Science*. Vol. 62. 2007, pp. 97–105.

[12] Nikolas Askitis and Justin Zobel. "Cache-conscious collision resolution in string hash tables". In: *Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE)*. 2005, pp. 91–102.

[13] Ricardo Baeza-Yates and Gonzalo Navarro. "Block addressing indices for approximate text retrieval". In: *Journal of the Association for Information Science and Technology* 51.1 (2000), pp. 69–82.

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

[15] Doug Baskins. *A 10-minute description of how Judy arrays work and why they are so fast*. 2002. URL: http://judy.sourceforge.net/doc/10minutes.htm.

[16] Holger Bast, Christian W Mortensen, and Ingmar Weber. "Output-sensitive autocompletion search". In: *Information Retrieval* 11.4 (2008), pp. 269–286. DOI: 10.1007/s10791-008-9048-x.

[17] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. "Representing trees of higher degree". In: *Algorithmica* 43.4 (2005), pp. 275–292.

[18] Tim Berners-Lee, James Hendler, and Ora Lassila. "The Semantic Web". In: *Scientific American* 284.5 (2001), pp. 28–37.

[19] Philip Bille, Inge Li Gørtz, and Nicola Prezza. "Practical and effective Re-Pair compression". In: *CoRR* abs/1704.08558 (2017). arXiv: 1704.08558. URL: http://arxiv.org/abs/1704.08558.

[20] Philip Bille, Inge Li Gørtz, and Nicola Prezza. "Space-efficient Re-Pair compression". In: *Proceedings of the Data Compression Conference (DCC)*. IEEE. 2017, pp. 171–180.

[21] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. "Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks". In: *Proceedings of the 20th International Conference on World Wide Web (WWW)*. 2011, pp. 587–596.

[22] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. "Ubicrawler: A scalable fully distributed Web crawler". In: *Software: Practice and Experience* 34.8 (2004), pp. 711–726. DOI: 10.1002/spe.587.

[23] Brazil Inc. *Groonga: An open-source fulltext search engine and column store*. 2017. URL: http://groonga.org/.

[24] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. "Compact representation of Web graphs with extended functionality". In: *Information Systems* 39 (2014), pp. 152–174.

[25] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. "DACs: Bringing direct access to variable-length codes". In: *Information Processing & Management* 49.1 (2013), pp. 392–404.

[26] Nieves R Brisaboa, Rodrigo Cánovas, Francisco Claude, Miguel A Martínez-Prieto, and Gonzalo Navarro. "Compressed string dictionaries". In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA)*. Vol. 6630. Springer, 2011, pp. 136–147.

[27] Michael Burrows and David J Wheeler. *A block-sorting lossless data compression algorithm*. Tech. rep. Digital Equipment Corporation, 1994.

[28] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. "The smallest grammar problem". In: *IEEE Transactions on Information Theory* 51.7 (2005), pp. 2554–2576.

[29] Francisco Claude and Gonzalo Navarro. "Fast and compact Web graph representations". In: *ACM Transactions on the Web* 4.4 (2010), Article 16.

[30] John G Cleary. "Compact hash tables using bidirectional linear probing". In: *IEEE Transactions on Computers* 33.9 (1984), pp. 828–834.

[31] Joshimar Cordova and Gonzalo Navarro. "Simple and efficient fully-functional succinct trees". In: *Theoretical Computer Science* 656 (2016), pp. 135–145.

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

[33] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*. Addiso Wesley, 2009.

[34] Jan Daciuk, Jakub Piskorski, and Strahil Ristov. "Natural language dictionaries implemented as finite automata". In: *Scientific applications of language methods*. Ed. by Carlos Martín-Vide. World Scientific, 2010. Chap. 4, pp. 133–204.

[35] Armon Dadgar. *libart: Adaptive Radix Trees implemented in C*. 2014. URL: https://github.com/armon/libart.

[36] John J Darragh, John G Cleary, and Ian H Witten. "Bonsai: A compact representation of trees". In: *Software: Practice and Experience* 23.3 (1993), pp. 277–291. DOI: 10.1002/spe.4380230305.

[37] Rene De La Briandais. "File searching using variable length keys". In: *Proceedings of the Western Joint Computer Conference*. ACM. 1959, pp. 295–298.

[38] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. "Engineering the LOUDS succinct tree representation". In: *Proceedings of the 5th International Workshop on Experimental and Efficient Algorithms (WEA)*. 2006, pp. 134–145.

[39] Debora Donato, Luigi Laura, Stefano Leonardi, Ulrich Meyer, Stefano Millozzi, and Jop F Sibeyn. "Algorithms and experiments for the Webgraph". In: *Journal of Graph Algorithms and Applications* 10.2 (2006), pp. 219–236.

[40] 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". In: *Information Processing & Management* 46.5 (2010), pp. 502–513. DOI: 10.1016/j.ipm.2009.08.004.

[41] John A Dundas. "Implementing dynamic minimal-prefix tries". In: *Software: Practice and Experience* 21.10 (1991), pp. 1027–1040. DOI: 10.1002/spe.4380211004.

[42] Peter Elias. "Efficient storage and retrieval by content and address of static files". In: *Journal of the ACM* 21.2 (1974), pp. 246–260.

[43] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Cambridge, MA: Memorandum 61, Computer Structures Group, MIT, 1971.

[44] Paolo Ferragina and Giovanni Manzini. "Indexing compressed text". In: *Journal of the ACM* 52.4 (2005), pp. 552–581.

[45] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. "Compressed representations of sequences and full-text indexes". In: *ACM Transactions on Algorithms* 3.2 (2007), p. 20.

[46] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. "On searching compressed string collections cache-obliviously". In: *Proceedings of the 27th Symposium on Principles of Database Systems (PODS)*. 2008, pp. 181–190.

[47] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S Muthukrishnan. "Structuring labeled trees for optimal succinctness, and beyond". In: *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2005, pp. 184–193.

[48] Edward Fredkin. "Trie memory". In: *Communications of the ACM* 3.9 (1960), pp. 490–499. DOI: 10.1145/367390.367400.

[49] Masao Fuketa, Hiroya Kitagawa, Takuki Ogawa, Kazuhiro Morita, and Jun'ichi Aoe. "Compression of double array structures for fixed length keywords". In: *Information Processing & Management* 50.5 (2014), pp. 796–806. DOI: 10.1016/j.ipm.2014.04.004.

[50] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. "An empirical study of real-world SPARQL queries". In: *Proceedings of the 1st International Workshop on Usage Analysis and the Web of Data (USEWOD)*. 2011.

[51] *GeoNames dump*. URL: http://download.geonames.org/export/dump/.

[52] Gaston H Gonnet and Ricardo Baeza-Yates. *Handbook of algorithms and data structures: in Pascal and C*. 2nd. Boston, Massachusetts, USA: Addison-Wesley, 1991.

[53] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. "Practical implementation of rank and select queries". In: *Poster Proceedings of the 4th Workshop on Experimental and Efficient Algorithms (WEA)*. 2005, pp. 27–38.

[54] Google Inc. *Sparsehash*. 2015. URL: https://github.com/sparsehash/sparsehash.

[55] Ronald Lewis Graham, Donald E Knuth, and Oren Patashnik. *Concrete mathematics: A foundation for computer science*. 2nd. Addison-Wesley, 1994.

[56] Roberto Grossi and Giuseppe Ottaviano. "Design of practical succinct data structures for large data collections". In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA)*. 2013, pp. 5–17.

[57] Roberto Grossi and Giuseppe Ottaviano. "Fast compressed tries through path decompositions". In: *ACM Journal of Experimental Algorithmics* 19.1 (2014), Article 1.8. DOI: 10.1145/2656332.

[58] Roberto Grossi and Giuseppe Ottaviano. "The wavelet trie: Maintaining an indexed sequence of strings in compressed space". In: *Proceedings of the 31st ACM symposium on Principles of Database Systems (PODS)*. 2012, pp. 203–214.

[59] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 3.2 (2005), pp. 158–182.

[60] Kenneth Heafield. "KenLM: Faster and smaller language model queries". In: *Proceedings of the 6th Workshop on Statistical Machine Translation (WMT)*. Association for Computational Linguistics. 2011, pp. 187–197.

[61] Harold Stanley Heaps. *Information retrieval: Computational and theoretical aspects*. Orlando, FL, USA: Academic Press, Inc., 1978.

[62] Steffen Heinz, Justin Zobel, and Hugh E Williams. "Burst tries: A fast, efficient data structure for string keys". In: *ACM Transactions on Information Systems* 20.2 (2002), pp. 192–223. DOI: 10.1145/506309.506312.

[63] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. "WebBase: A repository of Web pages". In: *Computer Networks* 33.1 (2000), pp. 277–293.

[64] Bo-June Paul Hsu and Giuseppe Ottaviano. "Space-efficient data structures for top-k completion". In: *Proceedings of the 22nd International Conference on World Wide Web (WWW)*. 2013, pp. 583–594.

[65] Te C Hu and Alan C Tucker. "Optimal computer search trees and variable-length alphabetical codes". In: *SIAM Journal on Applied Mathematics* 21.4 (1971), pp. 514–532.

[66] David A Huffman. "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[67] Guy Jacobson. "Space-efficient static trees and graphs". In: *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE. 1989, pp. 549–554.

[68] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. "Linked dynamic tries with applications to LZ-compression in sublinear time and space". In: *Algorithmica* 71.4 (2015), pp. 969–988. DOI: 10.1007/s00453-013-9836-6.

[69] Daniel C Jones. *hat-trie: an efficient trie implementation*. Version 0.1.1. 2017. URL: https://github.com/dcjones/hat-trie.

[70] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. "Compressed double-array tries for string dictionaries supporting fast lookup". In: *Knowledge and Information Systems* 51.3 (2017), pp. 1023–1042. ISSN: 02193116. DOI: 10.1007/s10115-016-0999-8.

[71] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. "Efficient string dictionary compression using string dictionaries". In: *DBSJ Japanese Journal* 16-J (2018), Article 7.

[72] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. "Practical implementation of space-efficient dynamic keyword dictionaries". In: *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE)*. 2017, pp. 221–233.

[73] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. "Practical string dictionary compression using string dictionary encoding". In: *Proceedings of the 3rd International Conference on Big Data Innovations and Applications (Innovate-Data)*. 2017, pp. 1–8.

[74] Shunsuke Kanda, Masao Fuketa, Kazuhiro Morita, and Jun'ichi Aoe. "A compression method of double-array structures using linear functions". In: *Knowledge and Information Systems* 48.1 (2016), pp. 55–80. DOI: 10.1007/s10115-015-0873-0.

[75] Shunsuke Kanda, Yuma Fujita, Kazuhiro Morita, and Masao Fuketa. "Practical rearrangement methods for dynamic double-array dictionaries". In: *Software: Practice and Experience* 48.1 (2018), pp. 65–83.

[76] Yusaku Kaneta. "Faster practical block compression for rank/select dictionaries". In: *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE)*. 2017, pp. 234–240.

[77] Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. "Efficient implementation of rank and select functions for succinct representation". In: *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms (WEA)*. Springer, 2005, pp. 315–327.

[78] Jon Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. "The Web as a graph: measurements, models, and methods". In: *Proceedings of the 5th Annual International Conference Computing and Combinatorics (COCOON)*. Springer, 1999, pp. 1–17.

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

[80] Taku Kudo, Kaoru Yamamoto, and Yuji Matsumoto. "Applying conditional random fields to Japanese morphological analysis". In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2004, pp. 230–237.

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

[82] *Laboratory for Web Algorithmics - Datasets*. 2011. URL: http://law.di.unimi.it/datasets.php.

[83] N Jesper Larsson and Alistair Moffat. "Off-line dictionary-based compression". In: *Proceedings of the IEEE* 88.11 (2000), pp. 1722–1732.

[84] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". In: *Proceedings of the IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 38–49. DOI: 10.1109/ICDE.2013.6544812.

[85] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. "Speeding up queries in column stores – A case for compression". In: *Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery (DAWAK)*. Springer. 2010, pp. 117–129.

[86] Cláudio L Lucchesi and Tomasz Kowaltowski. "Applications of finite automata representing large vocabularies". In: *Software: Practice and Experience* 23.1 (1993), pp. 15–30.

[87] Atsushi Maeda and Kouta Mizushima. "A compressed-array representation of automata and its application to programming language". In: *Proceedings of the 49th IPSJ Programming Symposium*. 2008, pp. 49–54.

[88] Veli Mäkinen and Gonzalo Navarro. "Dynamic entropy-compressed sequences and full-text indexes". In: *ACM Transactions on Algorithms* 4.3 (2008), Article 32.

[89] Udi Manber and Gene Myers. "Suffix arrays: A new method for on-line string searches". In: *SIAM Journal on Computing* 22.5 (1993), pp. 935–948.

[90] Christopher D Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. Cambridge, MA, USA: MIT press, 1999.

[91] Frank Manola and Eric Miller, eds. *RDF Primer*. W3C Recommendation, 2014. URL: https://www.w3.org/TR/rdf-primer/.

[92] Giovanni Manzini. "XBWT tricks". In: *Proceedings of the 23th International Symposium on String Processing and Information Retrieval (SPIRE)*. 2016, pp. 80–92.

[93] Miguel A Martínez-Prieto. *libCSD: C++ Library implementing Compressed String Dictionaries*. 2012. URL: https://github.com/migumar2/libCSD.

[94] Miguel A Martínez-Prieto, Javier D Fernández, and Rodrigo Cánovas. "Querying RDF dictionaries in compressed space". In: *ACM SIGAPP Applied Computing Review* 12.2 (2012), pp. 64–77.

[95] Miguel A Martínez-Prieto, Nieves R Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. "Practical compressed string dictionaries". In: *Information Systems* 56 (2016), pp. 73–108. DOI: 10.1016/j.is.2015.08.008.

[96] Shirou Maruyama, Hiroshi Sakamoto, and Masayuki Takeda. "An online algorithm for lightweight grammar-based compression". In: *Algorithms* 5.2 (2012), pp. 214–235.

[97] Shirou Maruyama, Yasuo Tabei, Hiroshi Sakamoto, and Kunihiko Sadakane. "Fully-online grammar compression". In: *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE)*. 2013, pp. 218–229.

[98] Ruslan Mavlyutov, Marcin Wylot, and Philippe Cudre-Mauroux. "A comparison of data structures to manage URIs on the Web of data". In: *Proceedings of the 12th European Semantic Web Conference (ESWC)*. 2015, pp. 137–151. DOI: 10.1007/978-3-319-18818-8_9.

[99] Kazuhiro Morita, Masao Fuketa, Yoshihiro Yamakawa, and Jun'ichi Aoe. "Fast insertion methods of a double-array structure". In: *Software: Practice and Experience* 31.1 (2001), pp. 43–65. DOI: 10.1002/1097-024x(200101)31:1<43::aid-spe356>3.0.co;2-r.

[100] Donald R Morrison. "PATRICIA: practical algorithm to retrieve information coded in alphanumeric". In: *Journal of the ACM* 15.4 (1968), pp. 514–534.

[101] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. "DBpedia SPARQL benchmark – performance assessment with real queries on real data". In: *Proceedings of the 10th International Semantic Web Conference (ISWC)*. 2011, pp. 454–469.

[102] Ingo Müller, Cornelius Ratsch, and Franz Faerber. "Adaptive string dictionary compression in in-memory column-store database systems". In: *Proceedings of the 17th International Conference on Extending Database Technology (EDBT)*. 2014, pp. 283–294.

[103] J Ian Munro and Venkatesh Raman. "Succinct representation of balanced parentheses and static trees". In: *SIAM Journal on Computing* 31.3 (2001), pp. 762–776.

[104] J Ian Munro, Venkatesh Raman, and Adam J Storm. "Representing dynamic binary trees succinctly". In: *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Vol. 1. 2001, pp. 529–536.

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

[106] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[107] Gonzalo Navarro. *Re-Pair compression and decompression*. 2010. URL: https://www.dcc.uchile.cl/~gnavarro/software/.

[108] Gonzalo Navarro and Kunihiko Sadakane. "Fully functional static and dynamic succinct trees". In: *ACM Transactions on Algorithms* 10.3 (2014), p. 16.

[109] Thomas Neumann and Gerhard Weikum. "RDF-3X: a RISC-style engine for RDF". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 647–659.

[110] Janne Nieminen and Pekka Kilpeläinen. "Efficient implementation of Aho-Corasick pattern matching automata using Unicode". In: *Software: Practice and Experience* 37.6 (2007), pp. 669–690.

[111] Jun-ya Norimatsu, Makoto Yasuhara, Toru Tanaka, and Mikio Yamamoto. "A fast and compact language model implementation using double-array structures". In: *ACM Transactions on Asian and Low-Resource Language Information Processing* 15.4 (2016), p. 27. DOI: 10.1145/2873068.

[112] Daisuke Okanohara. *Tx: Succinct trie data structure*. 2010. URL: https://github.com/hillbig/tx-trie.

[113] Daisuke Okanohara. *Ux*. 2013. URL: https://github.com/hillbig/ux-trie.

[114] Daisuke Okanohara and Kunihiko Sadakane. "Practical entropy-compressed rank/select dictionary". In: *Proceedings of the 9th Workshop on Algorithm Engineering & Experimiments (ALENEX)*. 2007, pp. 60–70.

[115] 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". In: *Software: Practice and Experience* 33.13 (2003), pp. 1229–1249. DOI: 10.1002/spe.545.

[116] Giuseppe Ottaviano. *path_decomposed_tries*. 2011. URL: https://github.com/ot/path_decomposed_tries.

[117] Giuseppe Ottaviano. "Space-efficient data structures for collections of textual data". PhD thesis. University of Pisa, 2013.

[118] Adam Pauls and Dan Klein. "Faster and smaller n-gram language models". In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 2011, pp. 258–267.

[119] W Wesley Peterson. "Addressing for random-access storage". In: *IBM Journal of Research and Development* 1.2 (1957), pp. 130–146.

[120] Giulio Ermanno Pibiri and Rossano Venturini. "Efficient data structures for massive n-gram datasets". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2017, pp. 615–624.

[121] *Pizza&Chili corpus*. URL: http://pizzachili.dcc.uchile.cl.

[122] Andreas Poyias, Simon J Puglisi, and Rajeev Raman. "Compact dynamic rewritable (CDRW) arrays". In: *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2017, pp. 109–119.

[123] Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. "m-Bonsai: a practical compact dynamic trie". In: *CoRR* abs/1704.05682 (2017). arXiv: 1704.05682. URL: http://arxiv.org/abs/1704.05682.

[124] Andreas Poyias and Rajeev Raman. "Improved practical compact dynamic tries". In: *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval (SPIRE)*. 2015, pp. 324–336. DOI: 10.1007/978-3-319-23826-5_31.

[125] Eric Prud'hommeaux and Andy Seaborne, eds. *SPARQL Query Language for RDF*. W3C Recommendation, 2008. URL: https://www.w3.org/TR/rdf-sparql-query/.

[126] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets". In: *ACM Transactions on Algorithms* 3.4 (2007), p. 43.

[127] Rajeev Raman and Srinivasa Rao Satti. "Succinct dynamic dictionaries and trees". In: *Proceedings of the 30th International Colloquium on Automata, Languages, and Programming (ICALP)*. 2003, pp. 357–368. DOI: 10.1007/3-540-45061-0_30.

[128] Hiroo Saito, Masashi Toyoda, Masaru Kitsuregawa, and Kazuyuki Aihara. "A large-scale study of link spam detection by graph algorithms". In: *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*. 2007, pp. 45–48.

[129] Sho Seshake and Hisatoshi Mochizuki. "Proposal to construction method of double array with short transition pattern". In: *Proceedings of the 15th Forum on Information Technology (FIT)*. 2016, pp. 89–90.

[130] Alan Silverstein. *Judy IV Shop Manual*. 2002.

[131] Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. "Packed compact tries: A fast and efficient data structure for online string processing". In: *Proceedings of the 27th International Workshop on Combinatorial Algorithms (IWOCA)*. 2016, pp. 213–225.

[132] Takanori Ueda, Koh Satoh, Daichi Suzuki, Kenji Uchida, Kousuke Morimoto, Sayaka Akioka, and Hayato Yamana. "A parallel distributed Web crawler consisting of producer-consumer modules". In: *IPSJ Transactions on Database* 6.2 (2013), pp. 85–97.

[133] *Wikipedia database dumps*. URL: https://dumps.wikimedia.org.

[134] Hugh E Williams and Justin Zobel. "Compressing integers for fast file access". In: *Computer Journal* 42.3 (1999), pp. 193–201.

[135] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: Compressing and indexing documents and images*. San Francisco, CA, USA: Morgan Kaufmann, 1999.

[136] Marcin Wylot, Jigé Pont, Mariusz Wisniewski, and Philippe Cudré-Mauroux. "dipLODocus[RDF] — short and long-tail RDF analytics for massive Webs of data". In: *Proceedings of the 10th International Semantic Web Conference (ISWC)*. 2011, pp. 778–793.

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

[138] Susumu Yata. *Darts-clone: a clone of Darts*. 2011. URL: https://github.com/s-yata/darts-clone.

[139] Susumu Yata. "Dictionary compression by nesting Prefix/Patricia tries". In: *Proceedings of the 17th Annual Meeting of the Association for Natural Language*. 2011.

[140] Susumu Yata. *MARISA: Matching Algorithm with Recursively Implemented StorAge*. 2011. URL: https://github.com/s-yata/marisa-trie.

[141] Susumu Yata. *Nihongo Web Corpus 2010 – N-gram corpus*. 2010. URL: http://s-yata.jp/corpus/nwc2010/ngrams/.

[142] Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, Toru Sumitomo, and Jun'ichi Aoe. "A compact static double-array keeping character codes". In: *Information Processing & Management* 43.1 (2007), pp. 237–247. DOI: 10.1016/j.ipm.2006.04.004.

[143] Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. "An efficient deletion method for a minimal prefix double array". In: *Software: Practice and Experience* 37.5 (2007), pp. 523–534. DOI: 10.1002/spe.778.

[144] Susumu Yata, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. "Cache-efficient double-array". In: *Proceedings of the 5th Forum on Information Technology (FIT)*. 2006, pp. 71–72.

[145] Susumu Yata, Masaki Oono, Kazuhiro Morita, Toru Sumitomo, and Jun'ichi Aoe. "Double-array compression by pruning twin leaves and unifying common suffixes". In: *Proceedings of the 1st International Conference on Computing & Informatics (ICOCI)*. 2006, pp. 1–4.

[146] Susumu Yata, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. "Fast string matching with space-efficient word graphs". In: *Proceedings of the 4th International Conference on Innovations in Information Technology (IIT)*. 2008, pp. 79–83.

[147] Susumu Yata, Masahiro Tamura, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. "Sequential insertions and performance evaluations for double-arrays". In: *Proceedings of the 71st National Convention of Information Processing Society of Japan (IPSJ)*. 2009, pp. 1263–1264.

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

[149] Naoki Yoshinaga and Masaru Kitsuregawa. "A self-adaptive classifier for efficient text-stream processing". In: *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*. 2014, pp. 1091–1102.

[150] Jacob Ziv and Abraham Lempel. "Compression of individual sequences via variable-rate coding". In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536.

[151] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. "Super-scalar RAM-CPU cache compression". In: *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. IEEE. 2006, p. 59.