b-Bit Sketch Trie: Scalable Similarity Search on Integer Sketches

Shunsuke Kanda
RIKEN Center for Advanced Intelligence Project
Tokyo, Japan
shunsuke.kanda@riken.jp

Yasuo Tabei
RIKEN Center for Advanced Intelligence Project
Tokyo, Japan
yasuo.tabei@riken.jp

Abstract—**Recently, randomly mapping vectorial data to strings of discrete symbols (i.e., *sketches*) for fast and space-efficient similarity searches has become popular. Such random mapping is called *similarity-preserving hashing* and approximates a similarity metric by using the Hamming distance. Although many efficient similarity searches have been proposed, most of them are designed for binary sketches. Similarity searches on integer sketches are in their infancy. In this paper, we present a novel space-efficient trie named *b-bit sketch trie* on integer sketches for scalable similarity searches by leveraging the idea behind *succinct data structures* (i.e., space-efficient data structures while supporting various data operations in the compressed format) and a favorable property of integer sketches as fixed-length strings. Our experimental results obtained using real-world datasets show that a trie-based index is built from integer sketches and efficiently performs similarity searches on the index by pruning useless portions of the search space, which greatly improves the search time and space-efficiency of the similarity search. The experimental results show that our similarity search is at most one order of magnitude faster than state-of-the-art similarity searches. Besides, our method needs only 10 GiB of memory on a billion-scale database, while state-of-the-art similarity searches need 29 GiB of memory.**

Index Terms—**Succinct Trie, Succinct Data Structures, Scalable Similarity Search, Similarity-preserving Hashing**

I. INTRODUCTION

The similarity search of vectorial data in databases has been a fundamental task in recent data analysis, and it has various applications such as near duplicate detection in a collection of web pages [1], context-based retrieval in images [2], and functional analysis of molecules [3]. Recently, databases in these applications have become large, and vectorial data in these databases also have been high dimensional, which makes it difficult to apply existing similarity search methods to such large databases. There is thus a strong need to develop much more powerful methods of similarity search for efficiently analyzing databases on a large-scale.

A powerful solution to address this need is *similarity-preserving hashing*, which intends to approximate a similarity measure by randomly mapping vectorial data in a metric space to strings of discrete symbols (i.e., *sketches*) in the Hamming space. Early methods include Sim-Hash for cosine similarity [4], which intends to build binary sketches from vectorial data for approximating cosine similarity. Quite a few similarity searches for binary sketches have been proposed thus far [5]–[11]. Recently, many types of similarity-preserving hashing

algorithms intending to build sketches of non-negative integers (i.e., *b-bit sketches*) have been proposed for efficiently approximating various similarity measures. Examples are b-bit minwise hashing (minhash) [12]–[14] for Jaccard similarity, 0-bit consistent weighted sampling (CWS) for min-max kernel [15], and 0-bit CWS for generalized min-max kernel [16]. Thus, developing scalable similarity search methods for b-bit sketches is a key issue in large-scale applications of similarity search.

Similarity searches on binary sketches are classified into two approaches: *single-* and *multi-indexes*. Single-index (e.g., [9], [11]) is a simple approach for similarity searches and builds an inverted index whose key is a sketch in a database and value is the identifiers with the same sketch. The similarity search for a query sketch is performed by generating all the sketches similar to the query as candidate solutions and then finding the solution set of sketches equal to a generated sketch by retrieving the inverted index. Typically, the hash table data structure is used for implementing the inverted index. A crucial drawback of single-index is that the query time becomes large for sketches with a large alphabet and a large Hamming distance threshold because the number of generated sketches is exponentially proportional to the alphabet size of sketches and the Hamming distance threshold.

To overcome the drawback of the single-index approach, the multi-index approach [17] has been proposed as a generalization of single-index for faster similarity searches, and it has been studied well for the past few decades [5]–[8], [10], [18]. Multi-index divides input sketches into several blocks of short sketches of possibly different lengths and builds inverted indexes from the short sketches in each block, where the key of an inverted index is a short sketch in each block, and its value is the identifier of the sketch with the short sketch. As in single-index, the inverted indexes are implemented using a hash table data structure. The similarity search of a query consists of two steps: filtering and verification. The filtering step divides the query into several short sketches in the same manner and finds short sketches similar to a short sketch of the query in each block by retrieving the inverted index. After that, the verification step removes false positives (i.e., a pair of short sketches in a block is similar but the corresponding pair of their original sketches is dissimilar) from those candidates by computing the Hamming distance between the pair of

every candidate and query in the verification step. Although the candidate solutions for short sketches in each block are generated in multi-index and are retrieved by the inverted index as in single-index, the number of candidate solutions generated in each block becomes smaller, resulting in faster similarity searches especially when a large threshold is used.

Many methods using the multi-index approach have been proposed for scalable similarity searches for binary sketches. Although some recent methods [5], [10], [18] have successfully improved the verification step in the multi-index approach, they have a serious issue when applied to b-bit sketches because the computation time of the filtering step is exponentially proportional to the alphabet size (i.e., the value of b) in b-bit sketches. Although Zhang et al. [19] have tried to improve the filtering step for b-bit sketches, their method has a scalability issue. Since many similarity preserving hashing algorithms for b-bit sketches have been proposed for approximating various similarity measures, an important open challenge is to develop a scalable similarity search for b-bit sketches.

Trie [20] is an ordered labeled tree data structure for a set of strings and supports various string operations such as string search and prefix search with a wide variety of applications in string processing such as string dictionaries [21], n-gram language models [22], and range query filtering [23]. A typical pointer-based representation of trie consumes a large amount of memory. Thus, recent researches have focused on space-efficient representations [23]–[25]. To date, trie has been applied only to the limited application domains listed above. However, as we will see, there remains great potential for a wide variety of applications.

Contribution: In this paper, we present a novel trie representation for b-bit sketches, which we call *b-bit Sketch Trie (bST)*, to enhance the search performance of single-index and multi-index. We design bST by leveraging a *succinct data structure* [24] (i.e, a compressed data structure while supporting fast data operations in the compressed format) and a favorable property behind b-bit sketches as fixed-length strings. Our similarity search method represents a database of b-bit sketches in bST and solves the Hamming distance problem for a given query by traversing the trie while pruning unnecessary portions of the search space. We experimentally test our similarity search using bST's ability to retrieve massive databases of b-bit sketches similar to a query and show that our similarity search performs superiorly with respect to scalability, search performance, and space-efficiency.

II. Similarity Search Problem

We formulate the similarity search problem for b-bit sketches. A b-bit sketch is an L-dimensional vector of integers, each of which is within range $[1, 2^b]$, and it is also equivalent to a string of length L over alphabet $\Sigma = [1, 2^b]$. We also denote elements in Σ (i.e., characters) by the small English letters (e.g., a, b, and c) in the examples of this paper. A database of b-bit sketches consists of n b-bit sketches $s_1, s_2, \ldots, s_n$, where $s_i \in \Sigma^L$. Given b-bit sketch q and threshold τ as a query, the task of the similarity search is to report all the identifiers I of b-bit sketches $s_1, s_2, \ldots, s_n$ whose Hamming distance to q is no more than τ, i.e., $I = \{i : \mathsf{ham}(s_i, q) \le \tau\}$, where $\mathsf{ham}(\cdot, \cdot)$ denotes the Hamming distance between two strings (i.e., the number of positions at which the corresponding characters between two strings are different).

III. Related Works

Many methods for similarity searches on binary sketches have been proposed, and they are classified into two approaches: single- and multi-indexes. Theoretical aspects of similarity search have also been argued [26]–[30]. The following subsections review practical similarity search methods.

A. Single-Index Approach

Single-index (e.g., [9], [11]) is a simple approach for the similarity search. This approach typically builds an inverted index implemented using a hash table data structure. From a database of sketches $s_1, s_2, \ldots, s_n$, it builds an inverted index whose key is sketch s_i in the database and value is the set of identifiers with the same sketch. The similarity search for given query q and threshold τ is performed by generating the set Q of all the sketches q' similar to query q (i.e., $Q = \{q' \in \Sigma^L : \mathsf{ham}(q, q') \le \tau\}$) and then finding the solution set I of sketches equal to generated sketch $q' \in Q$ (i.e., $I = \{i : \exists q' \in Q \text{ s.t. } s_i = q'\}$) by retrieving the inverted index. Each element in set Q is called *signature*, and single-index using the hash table data structure is referred to as *single-index hashing (SIH)*.

The search time of SIH is evaluated by using the retrieval time for the signatures in Q and access time for solution set I (see Appendix A for detailed analysis), and it is linearly proportional to L and exponentially proportional to τ and b. Since the number of signatures can exceed that of sketches in the database for large parameters b, L, and τ, a naive linear search can be faster than SIH. In particular, for $b > 1$ (i.e., non-binary sketches), the time performance is much more sensitive to τ, resulting in difficulty in applying SIH to b-bit sketches when a large τ is used.

B. Multi-Index Approach

The multi-index approach partitions sketch s_i for each $i = 1, 2, \ldots, n$ in a database into m blocks (i.e., substrings) $s_i^1, s_i^2, \ldots, s_i^m$ of lengths $L^1, L^2, \ldots, L^m$, respectively. The m blocks are disjoint. The approach builds inverted index X^j using the j-th block $s_1^j, s_2^j, \ldots, s_n^j$ for each $j = 1, 2, \ldots, m$, where the key of X^j is the j-th block s_i^j for each $i = 1, 2, \ldots, n$, and the value of X^j is the set of identifiers of the original sketch with the same block s_i^j.

A query is searched in two steps: filter and verification. Given query sketch q and threshold τ, q is partitioned into m blocks $q^1, q^2, \ldots, q^m$ of lengths $L^1, L^2, \ldots, L^m$, respectively. Thresholds $\tau^1, \tau^2, \ldots, \tau^m$ no more than τ are assigned to m blocks. Note that $\tau^j = \lfloor \tau/m \rfloor$ for $j = 1, 2, \ldots, m$ is traditionally used to avoid false negatives on the pigeonhole principle (e.g., [5], [19]). At the filter step, the set C^j

of candidate solutions for q^j for each $j = 1, 2, \ldots, m$ is obtained by retrieving inverted index X^j. As in the single-index approach, the set Q^j of all the sketches similar to q^j (i.e. $Q^j = \{q' \in \Sigma^{L^j} : \mathsf{ham}(q^j, q') \leq \tau^j\}$) for each $j = 1, 2, \ldots, m$ is generated, and C^j is computed by retrieving inverted index X^j for each $q' \in Q^j$. The verification step removes false positives (i.e., $\{i : \mathsf{ham}(s_i, q) \geq \tau, i \in C^j\}$) by computing the Hamming distance between the pair of sketches s_i for each $i \in C^j$ and query q. See Appendix A for detailed analysis of the search performance of the multi-index approach.

Quite a few similarity searches based on the multi-index approach have been proposed. We briefly review some state-of-the-art methods as follows. Gog and Venturini [5] proposed an efficient multi-index implementation method, which exploits succinct data structures [24] and triangle inequality. Qin et al. [10] generalized the pigeonhole principle and proposed a method that assigns variable thresholds τ^j for each $j = 1, 2, \ldots, m$ by considering the distribution of sketches. Qin and Xiao [18] proposed the pigeonring principle to shorten the verification time by exploiting the sum of thresholds assigned to adjacent blocks and constraining the number of candidate solutions (i.e., the size of C^j). The efficiency of the pigeonring principle is verified for long sketches (e.g., $L \geq 256$), enabling the assignment of a sufficient number of blocks [18].

Multi-index hashing (MIH) [9] is a state-of-the-art multi-index approach using the hash table data structure for implementing an inverted index. MIH partitions sketches s_i into m blocks of equal length (i.e., $L^j = \lfloor L/m \rfloor$) and assigns threshold $\tau^j = \lfloor \tau/m \rfloor - 1$ to the first $\tau - m \lfloor \tau/m \rfloor + 1$ blocks and $\tau^j = \lfloor \tau/m \rfloor$ to the other blocks.

HmSearch [19] is a representative multi-index approach originally designed for b-bit sketches. HmSearch partitions sketches into blocks by using the pigeonhole principle where the length of blocks is determined such that threshold τ^j for each block is zero or one. To avoid generating many signatures at the filter step, HmSearch registers all the signatures generated from each sketch in a database to the inverted index, resulting in a large memory consumption. Although many similarity search methods applicable to b-bit sketches have been proposed ([6], [7], [31]), Zhang et al. [19] show that HmSearch performs best.

Despite the importance of similarity searches for b-bit sketches, no previous work has been able to achieve both fast similarity search and space-efficiency. The problem behind the existing methods consists of (i) inefficient similarity searches because of a large number of generated signatures, (ii) a large verification cost in the multi-index approach, and (iii) a large space consumption for storing multiple inverted indexes. We solve this problem by presenting a trie-based similarity search method and a space-efficient trie representation, named bST, tailored for b-bit sketches. Our method with bST can be used instead of the inverted index using the hash table data structure in the single- and multi-index approaches. Since our method solves the similarity search problem for b-bit sketches by traversing the trie without signature generation, fast similarity
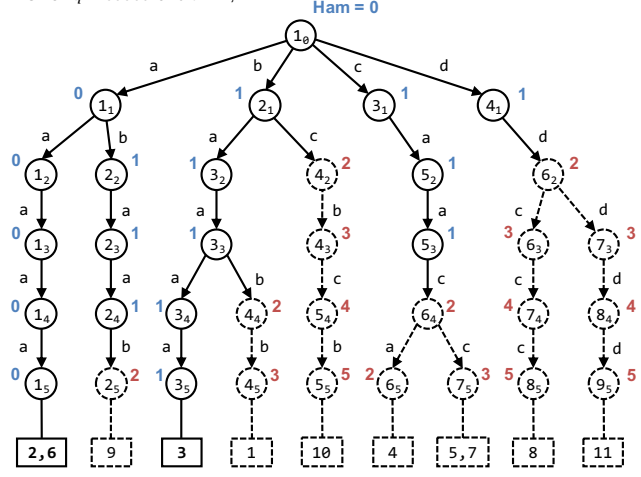

Fig. 1. Illustration of a trie built from eleven 2-bit sketches of `baabb`, `aaaaa`, `baaaa`, `caaca`, `caacc`, `aaaaa`, `caacc`, `ddccc`, `abaab`, `bcbcb`, and `ddddd`. $L = 5$. Each leaf has the sketch ids associated with sketches. The Hamming distance between query $q = $ `aaaaa` and the prefix of each node is represented by red/blue numbers. Solid circles (dashed circles) represent traversed nodes (pruned nodes) for q and $\tau = 1$.

searches can be performed even for a large threshold τ. Since bST compactly stores a massive database of b-bit sketches, bST enables scalable similarity searches. The details of the proposed method are presented in the next sections.

IV. TRIE-BASED SIMILARITY SEARCH

A key idea behind our similarity search is to build bST so that it supports data operations in the inverted index and to solve the similarity search problem on b-bit sketches by traversing bST for computing Hamming distances. bST can be used instead of the inverted index in the single-index approach and in the filtering step in the multi-index approach, which result in a fast and space-efficient similarity search for b-bit sketches. In this section, we first introduce a data structure of trie and then present a similarity search on a trie using pointers (PT).

A. Data Structure

Trie is an ordered labeled tree representing a set of sketches (see Figure 1). Each node is associated with the common prefix of a subset of input sketches, and the root node (i.e., the node without a parent) is not associated with any prefix. Each leaf (i.e., each node without any children) is associated with input sketches of the same characters and contains their identifiers. Each edge has a character of sketches as a label. All outgoing edges of a node are labeled with distinct characters.

The set of prefixes at each level ℓ consists of substrings of length ℓ and is assumed to be lexicographically sorted in ascending order. Each node (represented by solid/dashed circles in Figure 1) is associated with the unique node identifier (id) that is represented as notation u_ℓ by level ℓ and lexicographic order u of the prefix from left to right at level ℓ. Note that level ℓ is in $[0, L]$, i.e., the root id is 1_0. The u-th prefix at level ℓ is denoted by $\mathsf{str}(u_\ell)$.

Algorithm 1 Similarity search. q: query sketch, τ: Hamming distance threshold, u_ℓ: u-th node at level ℓ in PT, $dist$: the Hamming distance at u_ℓ for q, and I: solution set of ids.

1: Initialize $I \leftarrow \emptyset$
2: SIMSEARCH(1_0, 0)
3: **procedure** SIMSEARCH(u_ℓ, $dist$)
4:     **if** $dist > \tau$ **then** ▷ Hamming distance is more than τ
5:         **return**
6:     **end if**
7:     **if** $\ell = L$ **then**   ▷ Reach leaf node
8:         Add the ids associated with u_ℓ to I
9:         **return**
10:     **end if**
11:     Compute the set K of pairs $(v_{\ell+1}, c)$ by children(u_ℓ)
12:     **for** each pair $(v_{\ell+1}, c)$ in K **do**
13:         **if** $c \neq q[\ell + 1]$ **then**
14:             SIMSEARCH($v_{\ell+1}$, $dist + 1$)
15:         **else**
16:             SIMSEARCH($v_{\ell+1}$, $dist$)
17:         **end if**
18:     **end for**
19: **end procedure**

For the trie in Figure 1, node 3_3 represents the lexicographic order of $\mathsf{str}(3_3) = \mathtt{baa}$, which is the third node by following $\mathsf{str}(1_3) = \mathtt{aaa}$ and $\mathsf{str}(2_3) = \mathtt{aba}$.

B. Similarity Search

The similarity search for query sketch q and threshold τ on a trie represented by PT is performed by traversing the trie from root 1_0 to every leaf u_L in a depth-first manner while computing the Hamming distance between q and the prefix at each node. Algorithm 1 shows the pseudo-code of the similarity search on PT. First, we initialize solution set I to an empty set. We start at root $u_\ell = 1_0$ on PT and distance count $dist = 0$. The similarity search is recursively performed by the following three steps: (i) given node u_ℓ, we compute children(u_ℓ) that is a function returning set K of all the pairs $(v_{\ell+1}, c)$ of child $v_{\ell+1}$ and edge label c connecting u_ℓ and $v_{\ell+1}$, (ii) we recursively go down to each child $v_{\ell+1}$ in K if $\mathsf{ham}(\mathsf{str}(v_{\ell+1}), q[1..(\ell+1)])$ is no more than τ, where $q[i..j]$ denotes the substring $q[i]q[i+1]\ldots q[j]$ for $1 \leq i \leq j \leq L$, and (iii) if $v_{\ell+1}$ is a leaf, we add the ids associated with $v_{\ell+1}$ to solution set I. In step (ii), we stop going down to all the descendants under node $v_{\ell+1}$ if $\mathsf{ham}(\mathsf{str}(v_{\ell+1}), q[1..(\ell+1)])$ is more than τ without missing all the solutions.

Let t^{tra} be the number of traversed nodes. The time complexity is $\mathcal{O}(t^{tra} + |I|)$, where $|I|$ denotes the cardinality of set I. Let t be the number of nodes in a trie. t^{tra} can be much smaller than t when a small threshold τ is used. However, the space complexity is $\mathcal{O}(t \log t + t \cdot b)$ bits, and it is not feasible for PT to represent tries built from massive databases.

A crucial observation of the similarity search in Algorithm 1 is that trie should support the children operation for the implementation. Thus, we present bST supporting children for the scalable similarity search in the next section.

C. Review on Succinct Tries

A solution for compactly representing tries is to leverage succinct trie representations. We review two representative succinct tries in this subsection.

Level-order unary degree sequence trie (LOUDS-trie) [24], [25] is a succinct representation for tries and has a wide variety of applications (e.g., [32], [33]). LOUDS-trie represents a trie using $(b + 2) \cdot t + o(t)$ bits of space, which is much smaller than $\mathcal{O}(t \log t + t \cdot b)$ bits of space for PT, while supporting trie operations (e.g., computations of a child or the parent for a node) in constant time. *Fast succinct trie (FST)* [23] is a practical variant of LOUDS-trie. FST consists of two LOUDS structures: one is fast and the other is space-efficient. FST divides a trie into two layers at a certain level and builds the fast structure at the top layer and the space-efficient structure at the bottom layer. Although the space and time complexities of FST are the same as those of LOUDS-trie, FST is smaller and much faster in practice due to the layer-wise representation of trie.

LOUDS-trie and FST are not effective to manage b-bit sketches because they are designed for general strings and do not consider the favorable properties of b-bit sketches (e.g., strings of fixed-length L and the fixed-size 2^b of each character). In the next section, we present a novel trie representation bST, which is designed for storing b-bit sketches space-efficiently.

V. b-BIT SKETCH TRIE (bST)

In this section, we present bST, a space-efficient representation of trie for large sets of b-bit sketches, that supports the children function for scalable similarity searches. The bST design leverages an idea behind data distribution of b-bit sketches and *succinct data structures* [24], which are compressed data structures while supporting various data operations in the compressed format.

b-bit sketches are random strings such that the character at each position is equally distributed. This is a major difference from strings in the natural language where a character in each position appears with a bias. Thus, a trie built from b-bit sketches has the property that the higher the level the trie nodes are at, the more children they have. We design a compact trie representation as bST by leveraging this property.

The key ideas behind bST are (i) to divide a trie topology into three layers including subtries from the top level to the bottom level according to node density, where the node density in a layer is defined as the proportion of the number of nodes at the top level to the number of nodes at the bottom level, and (ii) to apply an optimal encoding into each set of subtries in each layer (see Figure 2).

Formally, ℓ_1 (ℓ_2) is the top level (the bottom level) in a layer, and $\ell_1 < \ell_2$. The node density $D(\ell_1, \ell_2)$ in the layer from ℓ_1 to ℓ_2 is defined as

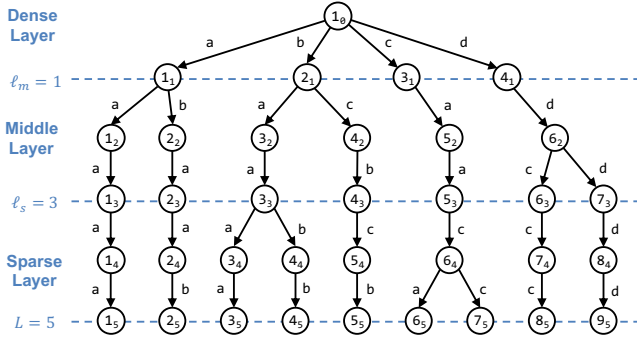$$D(\ell_1, \ell_2) = \frac{t_{\ell_2}}{t_{\ell_1}}, \tag{1}$$

Fig. 2. Illustration of the division of a trie topology into three layers of dense, middle, and sparse layers.

children(2_1) = {(3_2, a), (4_2, c)} is computed as follows:
i. Compute range [5, 8] as (2−1)·4+1 = 5 and 2·4 = 8
   (the range [5,8] corresponds to the children of node 2_1)
ii. Compute the number of 1s at position (5-1) as rank(H_2, 5-1) = 2
iii. Compute the number of 1s at position 8 as rank(H_2, 8) = 4
iv. Compute pair (3_2, a) as select(H_2, 3) mod 4 = 5 mod 4 = a, and
    pair (4_2, c) as select(H_2, 4) mod 4 = 7 mod 4 = c

Fig. 3. Illustration of TABLE representation for nodes at level 2 of the trie in Figure 2 and computation example of children(2_1).

where t_{ℓ_1} (t_{ℓ_2}) is the number of nodes in the top level ℓ_1 (the number of nodes in the bottom level ℓ_2). The layers consist of (i) *dense layer*, (ii) *sparse layer*, and (iii) *middle layer* and are determined according to node densities as follows: (i) the dense layer is a layer from the top level (i.e., level $\ell = 0$) to the maximum level ℓ_m satisfying density condition $D(0, \ell_m) = 2^{b\ell_m}$ for given b, (ii) the sparse layer is a layer from the bottom level L (i.e., all the nodes are leaves at level L) to the minimum level ℓ_s ($\geq \ell_m$) satisfying density condition $D(\ell_s, L) < \lambda$ for parameter $\lambda \in (0, 1)$, and (iii) the middle layer is the remaining layer except for the dense and sparse layers (i.e., the layer between level ℓ_m to level ℓ_s).

We present novel compact representations for subtries in dense, sparse, and middle layers in the following subsections.

Rank and Select Data Structures: bST leverages rank and select data structures [24] that are succinct data structures on a bit array and supports rank and selection operations on bit array B of length N as follows:

- rank(B, i) returns the number of occurrences of bit 1 in $B[1..i]$.
- select(B, i) returns the position in B of the i-th occurrence of bit 1; however, if i exceeds the number of 1s in B, it always returns $N + 1$.

Given a bit array $B = [01101011]$, rank($B, 5$) = 3 and select($B, 4$) = 7.

The operations can be performed in $\mathcal{O}(1)$ time by using auxiliary data structures of only $o(N)$ additional bits [24]. In our experiments, we implemented rank and select using the *succinct data structure library* [34].

A. Representation for Dense Layer

The dense layer between level 0 and level ℓ_m includes a complete 2^b-ary trie of height ℓ_m and with $2^{b\ell_m}$ leaves. A characteristic property of the complete 2^b-ary trie is that we can compactly represent it by storing only level information ℓ_m rather than complete information on the trie such as topology and edge labels. Thus, the space usage for storing a complete 2^b-ary trie with height ℓ_m is $\mathcal{O}(\log \ell_m)$.

Given node u_ℓ such that $\ell < \ell_m$, children(u_ℓ) computes 2^b pairs $(v_{\ell+1}, c)$ of children $v_{\ell+1}$ and edge labels c. The

operation returns $\{((v + 1)_{\ell+1}, 1), ((v + 2)_{\ell+1}, 2), \ldots, ((v + 2^b)_{\ell+1}, 2^b)\}$ where $v = (u - 1) \cdot 2^b$.

For the trie in Figure 2, children(1_0) returns $\{(1_1, \text{a}), (2_1, \text{b}), (3_1, \text{c}), (4_1, \text{d})\}$. This is because children(1_0) is computed as $\{(1_1, 1), (2_1, 2), (3_1, 3), (4_1, 4)\}$, and 1, 2, 3, and 4 correspond to a, b, c, and d, respectively.

B. Representation for Middle Layer

The middle layer includes dense and sparse nodes. We present two representations of *TABLE* and *LIST* for the nodes at the middle layer. Either TABLE or LIST is adaptively selected according to the node density at each level and is applied to the nodes.

TABLE Representation: We use bit array H_ℓ of length $2^b \cdot t_{\ell-1}$ for compactly representing the nodes at level $\ell \in [\ell_m + 1, \ell_s]$. The $((u-1) \cdot 2^b + c)$-th position on H_ℓ (i.e., $H_\ell[(u-1) \cdot 2^b + c]$) is 1 if and only if each node $u_{\ell-1}$ at level $\ell - 1$ has a child with edge label c. Thus, each position on H_ℓ represents whether there exists edge label c connecting from node $u_{\ell-1}$ at level $\ell - 1$ to a child at level ℓ.

Array H_2 in Figure 3 shows the TABLE representation of nodes at level 2 of the trie in Figure 2. Node 2_1 in the trie has edge labels of a and c connecting 2_1's children of 3_2 and 4_2, respectively. Thus, the first and second positions in H_2 are 5 and 7, respectively.

Given node $u_{\ell-1}$, children($u_{\ell-1}$)(= K) is computed as follows: (i) compute range $[i, j]$ on H_ℓ as $i \leftarrow (u-1) \cdot 2^b + 1$ and $j \leftarrow u \cdot 2^b$, (ii) compute the number of 1s at position $(i-1)$ as $x \leftarrow$ rank($H_\ell, i - 1$) if $i > 1$ or $x \leftarrow 0$ otherwise, (iii) compute the number of 1s at position j as $y \leftarrow$ rank(H_ℓ, j), and (iv) compute the set K of pairs of child id v_ℓ and edge label c as $c \leftarrow$ select(H_ℓ, v) mod 2^b for all $v \in [x + 1, y]$.

The algorithm is made possible because range $[i, j]$ corresponds to the children of $u_{\ell-1}$, and there is a 1-to-1 correspondence between the 1s on $H_\ell[i..j]$ and the children of $u_{\ell-1}$.

The bottom of Figure 3 shows an example for computing children(2_1). In this example, $2^b = 4$. Given node 2_1, range $[i, j]$ is computed as $i \leftarrow (2-1) \cdot 4 + 1 = 5$ and $j \leftarrow 2 \cdot 4 = 8$. The number of 1s at the fourth position is computed as $x \leftarrow$ rank($H_2, 5 - 1$) = 2. The number of 1s at the eighth position is computed as $y \leftarrow$ rank($H_2, 8$) = 4. Pair (3_2, a) in K is computed as $x + 1 = 3$ and select($H_2, 3$) mod

children(6_2) = {(6_3, c), (7_3, d)} is computed as follows:
  i.   Compute range [6, 7] as select(B_3, 6) = 6 and select(B_3, 6+1)−1 = 7
  ii. Compute pairs (6_3, C_3[6]) = (6_3, c) and (7_3, C_3[7]) = (7_3, d)

Fig. 4. Illustration of LIST representation for nodes at level 3 of the trie in Figure 2 and computation example of children(6_2).

Paths from node 5_3 to leaves 6_5 and 7_5 are restored as follows:
  i.   Compute leaf id range [6, 7] as select(D, 5) = 6 and select(D, 5+1)−1 = 7
  ii.   Restore path ca from node 5_3 to leaf 6_5 as P[(6−1)·2+1..6·2] = P[11..12] = ca
  iii. Restore path cc from node 5_3 to leaf 7_5 as P[(7−1)·2+1..7·2] = P[13..14] = cc

Fig. 5. Representation of subtries in sparse layer of the trie in Figure 2.

$4 = 5 \mod 4 = $ a, and pair $(4_2, c)$ in K is computed as $y = 4$ and select($H_2, 4$) $\mod 4 = 7 \mod 4 = $ c. Thus, $K = \{(3_2, a), (4_2, c)\}$.

The space usage of TABLE for nodes at level ℓ is $2^b \cdot t_{\ell-1} + o(2^b \cdot t_{\ell-1})$ bits.

LIST Representation: LIST represents nodes at level $\ell \in [\ell_m + 1, \ell_s]$ using array C_ℓ of length t_ℓ and bit array B_ℓ of length t_ℓ. For each node u_ℓ at level ℓ, $C_\ell[u]$ stores the edge label between u_ℓ and its parent. $B_\ell[u]$ stores bit 1 (i.e., $B_\ell[u] = 1$) if u_ℓ is the first of its siblings (i.e., u is the smallest id for its siblings) or bit 0 otherwise. Arrays C_3 and B_3 in Figure 4 are the LIST representation at level 3 of the trie in Figure 2.

Given node $u_{\ell-1}$, children($u_{\ell-1}$)($= K$) is computed as follows: (i) compute range $[i, j]$ on C_ℓ and B_ℓ as $i \leftarrow$ select(B_ℓ, u) and $j \leftarrow$ select($B_\ell, u+1$) -1 and (ii) compute the set K of pairs of child id v_ℓ and edge label c as $c \leftarrow C_\ell[v]$ for all $v \in [i, j]$.

The bottom of Figure 4 shows an example for computing children(6_2). $K = \{(6_3, c), (7_3, d)\}$. Given node 6_3, range $[i, j]$ is computed as $i \leftarrow$ select($B_3, 6$) $= 6$ and $j \leftarrow$ select($B_3, 7$) $-1 = 7$. Pairs $(6_3, c)$ and $(7_3, d)$ in K are computed as $C_3[6] = $ c and $C_3[7] = $ d, respectively.

The space usage of LIST for nodes at level ℓ is $(b+1) \cdot t_\ell + o(t_\ell)$ bits. When all t nodes are represented by LIST, the space usage is $\sum_{\ell=1}^{L}\{(b+1) \cdot t_\ell + o(t_\ell)\} = (b+1) \cdot t + (t)$ bits and is more space-efficient than the LOUDS-trie of $(b+2) \cdot t + o(t)$ bits of space.

Selection of TABLE and LIST Representations: Either the TABLE representation or LIST representation is adaptively applied to the series of nodes according to node density (Eq. 1) at each level. For the selection, we ignore the auxiliary spaces for rank and select because they are negligible. Since the space usages for the TABLE representation (the LIST representation) at level ℓ are $2^b \cdot t_{\ell-1}$ $((b+1) \cdot t_\ell)$, we use threshold $2^b/(b+1)$ for the node density. If $D(\ell-1, \ell) > 2^b/(b+1)$, TABLE is more space efficient and is applied; otherwise, LIST is applied.

C. Representation for Sparse Layer

The sparse layer between level ℓ_s and level L includes a set of subtries, each of which has a height of $L - \ell_s$. We collapse the subtries into their root-to-leaf paths and handle them as strings rather than trie structures. The representation for the sparse layer uses two arrays of P and D. P is an array of length $(L - \ell_s) \cdot t_L$ such that P[$(L - \ell_s) \cdot (v-1) + 1..(L - \ell_s) \cdot v$] stores the edge labels on the path from the root in the subtrie
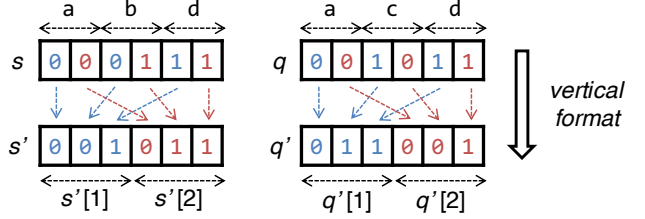

Fig. 6. Illustration of the sketches s' and q' in vertical format for 2-bit sketches of length 3. $s = $ abd, and $q = $ acd.

containing leaf v_L. D is a bit array of length t_L such that D[v] stores 1 if and only if leaf v_L is the leftmost leaf in the subtrie. Figure 5, which shows arrays P and D, is an example of the representation for the sparse layer.

The paths from a given node u_{ℓ_s} to the leaves can be restored using the select operation on D in the following four steps: (i) $i \leftarrow$ select(D, u), (ii) $j \leftarrow$ select(D, $u+1$) -1, (iii) compute P[$(L - \ell_s) \cdot (i-1) + 1..(L - \ell_s) \cdot j$], and (iv) return subarrays of P[$(L - \ell_s) \cdot (i-1) + 1..(L - \ell_s) \cdot j$] and of every length $(L - \ell_s)$ as the paths. Leaf ids of the subtrie with root u_{ℓ_s} are computed as v_L for $v \in [i, j]$.

Figure 5 shows an example of restoring the paths ca and cc from node 5_3 to leaves 6_5 and 7_5, respectively. Given node 5_3, $i \leftarrow$ select(D, 5) $= 6$ and $j \leftarrow$ select(D, 6) $-1 = 7$ are computed. Leaf ids of the subtrie with root 5_3 are 6_5 and 7_5. In this example, $L - \ell_s = 5 - 3 = 2$. The path from root 5_3 to leaf 6_5 is P[$(2 \cdot (6-1) + 1)..(2 \cdot 6)$] = P[11..12] = ca. The path from root 5_3 to leaf 7_5 is P[$(2 \cdot (7-1) + 1)..(2 \cdot 7)$] = P[13..14] = cc.

After restoring paths in a subtrie by using P and D, we can simulate traversing the subtrie by computing the Hamming distance between the query and the paths. Therefore, the fast computation of ham is crucial in the sparse layer, which is explained in the next paragraph.

Hamming Distance Computation Approach: Let us consider computing ham(s, q) for b-bit sketches s and q, each of length L. When the operation is naively performed by comparing s and q character by character, the computation time is $\mathcal{O}(L)$.

Zhang et al. [19] proposed a faster computation approach by exploiting a vertical layout and the bit-parallelism offered by CPUs. For this approach, we consider the binary representation of a character in sketches, e.g., a = 00, b = 01, c = 10, and d = 11 for $b = 2$. The approach encodes s into s' in a *vertical* format, i.e., the i-th significant L bits of each character of s are stored in $s'[i]$ of consecutive L bits. The resulting s' is an

array of length b in which each element is in L bits. Figure 6 shows an example of 2-sketches each of length 3, which are represented in the vertical format.

Given sketches s' and q', we can compute $\mathsf{ham}(s,q)$ as follows. Initially, we prepare a bitmap $bits$ of L bits in which all the bits are set to 0. For each $i = 1, 2, \ldots, b$, we iteratively perform $bits \leftarrow bits \lor (s'[i] \oplus q'[i])$, where $\lor$ and $\oplus$ denote bitwise OR and XOR operations, respectively. For the resulting $bits$, $\mathsf{popcnt}(bits)$ is the same as $\mathsf{ham}(s,q)$, where popcnt counts the number of 1s and belongs to the instruction sets of any modern CPU. For example, for s' and q' in Figure 6, the resulting $bits$ becomes 010 as $(s'[1] \oplus q'[1]) \lor (s'[2] \oplus q'[2]) = (001 \oplus 011) \lor (011 \oplus 001) = 010 \lor 010 = 010$. $\mathsf{popcnt}(010)$ is the same as $\mathsf{ham}(\texttt{abd}, \texttt{acd})$ (i.e., one). The operations $\lor$, $\oplus$, and popcnt can be performed in $\mathcal{O}(1)$ time per machine word. Let w be the machine word size; we can compute $\mathsf{ham}(s,q)$ in $\mathcal{O}(b\lceil L/w\rceil)$ time.

We conducted preliminary experiments to compare the computation speeds of the naive and vertical-format approaches. From the result for 32-dimensional 4-bit sketches, the computation time of the vertical-format approach was more than an order of magnitude faster than that of the naive approach. Therefore, we apply the vertical-format approach to P representation.

VI. Experiments

In this section, we demonstrate the effectiveness of similarity searches using bST through experiments using real-world datasets. The source code implementing our bST is available at https://github.com/kampersanda/integer_sketch_search.

A. Setup

We used four real-world datasets as shown in Table I. *Review* is 12,886,488 book reviews in English from Amazon [35]. We eliminated the stop words from the reviews and then represented each of reviews as a 9,253,464 dimensional fingerprint where each dimension of the fingerprint represents the presence or absence of a word. We used b-bit minhash [14] to convert each binary vector into a 2-bit sketch of 16 dimensions. *CP* consists of 216,121,626 compound-protein pairs each of which is represented as a binary vector of 3,621,623 dimensions. We used b-bit minhash to convert each binary vector into a 2-bit sketch of 32 dimensions. *SIFT* consists of 128 dimensional SIFT descriptors built from the BIGANN dataset [36] of one billion images. We used 0-bit CWS [15] to convert each feature into a 4-bit sketch of 32 dimensions. *GIST* consists of 384 dimensional GIST descriptors built from 79,302,017 tiny images [37]. We used 0-bit CWS to convert each descriptor into a 8-bit sketch of 64 dimensions. Following [14], [16], we used parameter settings of $b = 2$ for b-bit minhash and $b = 4$ or 8 for 0-bit CWS.

We randomly sampled 1,000 vectors from each dataset for queries. We evaluated the search time for τ in the range from 1 to 5 and chose parameters L for each dataset in consideration of the number of solutions obtained. Table II shows the average

TABLE I
SUMMARY OF DATASETS.

	n	Hashing	L	b
Review	12,886,488	b-bit minhash	16	2
CP	216,121,626	b-bit minhash	32	2
SIFT	1,000,000,000	0-bit CWS	32	4
GIST	79,302,017	0-bit CWS	64	8

TABLE II
AVERAGE NUMBER OF SOLUTIONS.

	$\tau = 1$	$\tau = 2$	$\tau = 3$	$\tau = 4$	$\tau = 5$
Review	12	28	181	1,273	7,671
CP	418	622	1,473	2,831	5,201
SIFT	174	1,057	5,603	26,840	111,727
GIST	168	1,664	10,787	51,085	189,188

number of solutions for each τ, and a substantial number of solutions are obtained.

We conducted all experiments on one core of quad-core Intel Xeon CPU E5–2680 v2 clocked at 2.8 Ghz in a machine with 256 GB of RAM, running the 64-bit version of CentOS 6.10 based on Linux 2.6.

B. Comparison of Succinct Tries

We compared bST with the state-of-the-art succinct tries of LOUDS-trie and FST in combination with the single-index approach. LOUDS-trie was implemented using the TX library downloadable from https://github.com/hillbig/tx-trie, and FST was implemented using the SuRF library downloadable from https://github.com/efficient/SuRF. Since the single-index approach needs the inverted index for similarity searches, it enables us to fairly evaluate the search performance and space usage of data structures for implementing an inverted index. Thus, we evaluated the performance of similarity searches on single-index using a succinct trie on each dataset. We fixed parameter $\lambda = 0.5$. Parameters ℓ_m and ℓ_s were used as $(\ell_m, \ell_s) = (8, 11)$ for Review, $(\ell_m, \ell_s) = (9, 14)$ for CP, $(\ell_m, \ell_s) = (0, 21)$ for SIFT, and $(\ell_m, \ell_s) = (0, 49)$ for GIST.

LOUDS-trie and FST were applicable to sketches whose length was less than 2^{32} because of the implementation issues with the TX and SuRF libraries, respectively. Thus, they were not applicable to SIFT whose length was 32 billion.

Table III shows the experimental results of search time and space usage. bST was much faster than LOUDS-trie and FST by a large margin. bST was at most 6.2 times faster than LOUDS-trie and was at most 3.8 times faster than FST on Review. bST was at most 5.0 times faster than LOUDS-trie and was at most 4.4 times faster than FST on CP. bST was much more space-efficient than LOUDS-trie and FST. bST was 2.6 times smaller than LOUDS-trie and was 1.9 times smaller than FST on Review. bST was 2.4 times smaller than LOUDS-trie and was 1.8 times smaller than FST on CP.

Those results show that bST as an engineered representation for b-bit sketches was much more efficient than LOUDS-trie and FST with respect to search performance and space usage.

| | Review | | | | | |
	$\tau=1$	$\tau=2$	$\tau=3$	$\tau=4$	$\tau=5$	Space
bST	**0.006**	**0.05**	**0.37**	**2.0**	**8.3**	**9**
LOUDS	0.036	0.32	2.23	11.6	45.2	24
FST	0.021	0.19	1.42	7.7	31.5	18

| | CP | | | | | |
	$\tau=1$	$\tau=2$	$\tau=3$	$\tau=4$	$\tau=5$	Space
bST	**0.019**	**0.14**	**0.88**	**4.9**	**23**	**308**
LOUDS	0.090	0.67	4.37	23.1	99	741
FST	0.084	0.60	3.88	20.0	82	548

| | SIFT | | | | | |
	$\tau=1$	$\tau=2$	$\tau=3$	$\tau=4$	$\tau=5$	Space
bST	**0.22**	**3.4**	**30**	**171**	**690**	**6,082**
LOUDS	–	–	–	–	–	–
FST	–	–	–	–	–	–

| | GIST | | | | | |
	$\tau=1$	$\tau=2$	$\tau=3$	$\tau=4$	$\tau=5$	Space
bST	**0.32**	**3.8**	**26**	**105**	**304**	**1,072**
LOUDS	0.75	9.5	65	279	905	1,329
FST	0.53	7.0	49	206	633	1,163

	Review	CP	SIFT	GIST
SI-bST	**48**	**1,057**	**9,802**	**1,338**
MI-bST ($m=2$)	126	3,232	23,159	5,513
SIH	172	2,329	32,727	4,501
MIH ($m=2$)	125	4,633	28,876	6,128
MIH ($m=3$)	160	3,997	26,665	5,744
HmSearch ($\tau=1,2$)	866	53,097	–	48,456
HmSearch ($\tau=3,4$)	860	29,396	–	27,337
HmSearch ($\tau=5$)	860	28,866	–	25,305

bST was the only method applicable to SIFT. The similarity search performance for bST is shown in the next subsection.

C. Comparison of Similarity Search Methods

We compared single-index and multi-index using bST with representative similarity search methods of SIH, MIH, and HmSearch. Single-index and multi-index using bST for implementing an inverted index are referred to as SI-bST and MI-bST, respectively. HmSearch is the state-of-the-art similarity search for b-bit sketches and is reviewed in Section III.

SIH and MIH are single-index and multi-index using the hash table for implementing an inverted index, respectively. Since SIH and MIH were designed for binary sketches (i.e., $b=1$) as in [9], we modified them for integer sketches (i.e., $b>1$). Since the similarity search of SIH with large τ and b took a large amount of time, we limited the execution time to within 10 seconds per query.

The implementations of SIH and MIH are also contained in our library. The implementation of HmSearch is available at https://github.com/kampersanda/hmsearch.

For MI-bST and MIH, we tested the number of blocks $m \in \{2,3,4\}$ for each threshold τ and chose the best value of m achieving the fastest similarity search. MI-bST with $m=2$ was the fastest for all pairs of datasets and thresholds. MIH with $m=3$ was the fastest for $\tau \in [4,5]$ on GIST, and MIH with $m=2$ was the fastest for the other pairs.

Figure 7 shows the average similarity search time in milliseconds (ms) per query for each method. Since SIH had been designed for binary sketches, its performance was evaluated only with them [9], [11]. In contrast, when integer sketches were used, SIH did not perform well even for small τ for each dataset. SIH did not finish within 10 seconds for a τ of no less

than four on SIFT and no less than two on GIST because the large numbers of signatures were generated on those datasets.

Although MIH had also been designed for binary sketches as well as SIH, it performed well for large τ in contrast to SIH. On the other hand, MIH was slower than SIH for small τ (e.g., $\tau=1$) on each dataset.

Although HmSearch was a similarity search on the multi-index approach with engineered assignments of the number of blocks and thresholds, it was slower than MIH on Review, CP, and GIST. In addition, the space usage of HmSearch exceeded the memory limitation of 256 GB on SIFT. Those results on b-bit sketches were consistent with those results on binary sketches, which were shown in [38].

For each $\tau \leq 4$, SI-bST was the fastest among all the methods on each dataset, while only SIH was competitive compared to SI-bST for $\tau=1$ on Review and SIFT. For $\tau=5$, MI-bST and MIH were the fastest and competitive except on GIST, while SI-bST was the fastest on GIST. The results show our SI-bST and MI-bST were the fastest for similarity searches on huge datasets of b-bit sketches.

Table IV shows the experimental results of space-efficiency. HmSearch, MIH, and MI-bST were similarity searches on the multi-index approach. HmSearch consumed a large amount of memory and consumed approximately 860 MiB memory for Review, more than 256 GiB memory for SIFT, and at least 25 GiB memory for GIST. MIH was more space-efficient than HmSearch, but MIH's space-usage was problematic for large datasets. In particular, MIH consumed more than 26 GiB of memory for SIFT and more than 5.7 GiB of memory for GIST. MI-bST was the most space-efficient among all methods on the multi-index approach for each dataset. MI-bST consumed 3.2 GiB for CP, 23 GiB for SIFT, and 5.4 GiB for GIST. Although SIH was a similarity search on the single-index approach, SIH was not space-efficient, and it consumed 2.3 GiB for CP , 32 GiB for SIFT, and 4.5 GiB for GIST. SI-bST was the most space-efficient among all the methods for each dataset. SI-bST consumed only 1.0 GiB for CP, 9.6 GiB for SIFT, and 1.3 GiB for GIST.

For thresholds $\tau \leq 4$, the results of the comparison of similarity searches showed that SI-bST was the best among all the methods in terms of search time and space usage. For $\tau=5$, MI-bST can be used instead of SI-bST for fast and space-efficient similarity searches.
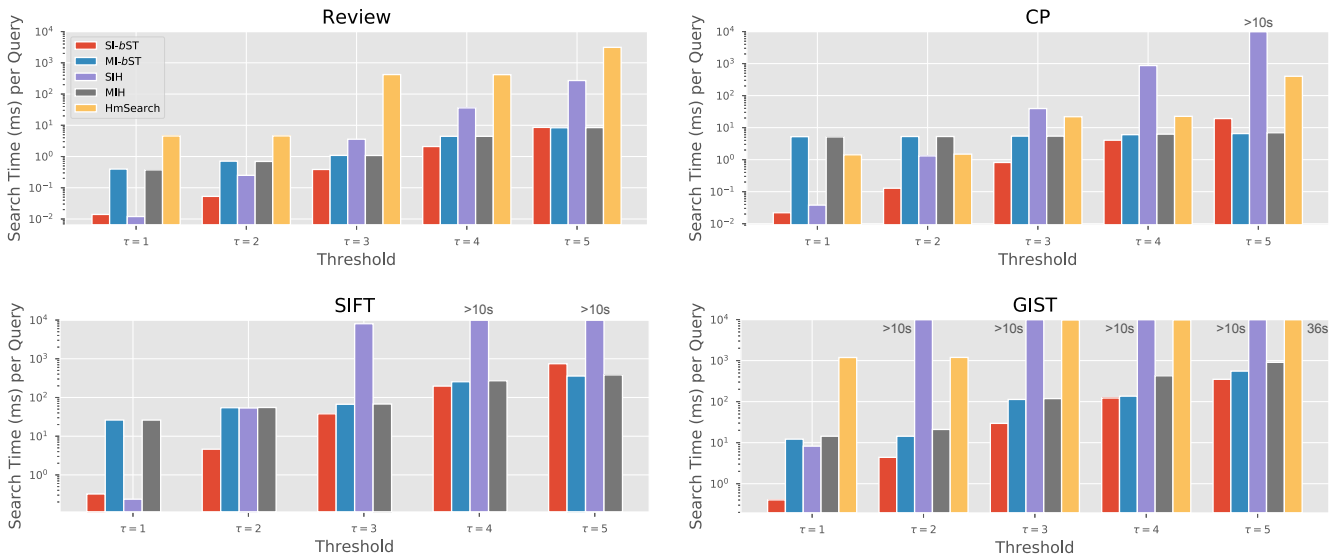
Fig. 7. Average search time in milliseconds per query (log scale). The bars denote SI-bST (red), MI-bST (blue), SIH (purple), MIH (gray), and HmSearch (yellow) starting from the left. The upper limit of results the figure plots is 10 seconds since we aborted the similarity search process of SIH if the time exceeds 10 seconds. Other than SIH, only the result of HmSearch for GIST when $\tau = 5$ exceeded 10 seconds (it was 36 seconds).

VII. CONCLUSION

We presented bST, a novel succinct representation of trie for fast and space-efficient similarity searches on b-bit sketches. Our experimental results using real-world datasets demonstrated that bST outperformed other state-of-the-art succinct tries in terms of search time and memory.

Subsequently, we presented SI-bST and MI-bST, single-index and multi-index using bST implementing an inverted index. Our experimental results demonstrated that SI-bST was the fastest for thresholds $\tau \leq 4$. For $\tau = 5$, MI-bST was the alternative to SI-bST. In addition, the space-efficiency of SI-bST was the best among all the methods, and it consumed 10 GiB of memory for storing a billion-scale database, while a state-of-the-art method consumed 29 GiB of memory.

REFERENCES

[1] M. Henzinger, "Finding near-duplicate web pages: a large-scale evaluation of algorithms," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2006, pp. 284–291.

[2] J. Song, Y. Yang, Y. Yang, Z. Huang, and H. T. Shen, "Inter-media hashing for large-scale retrieval from heterogeneous data sources," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 785–796.

[3] J.-I. Ito, Y. Tabei, K. Shimizu, K. Tsuda, and K. Tomii, "PoSSuM: a database of similar protein–ligand binding and putative pockets," *Nucleic Acids Research*, vol. 40, pp. D541–D548, 2012.

[4] A. Gionis, P. Indyk, R. Motwani, and Others, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, vol. 99, no. 6, 1999, pp. 518–529.

[5] S. Gog and R. Venturini, "Fast and compact Hamming distance index," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2016, pp. 285–294.

[6] A. X. Liu, K. Shen, and E. Torng, "Large scale hamming distance query processing," in *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, 2011, pp. 553–564.

[7] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *Proceedings of the 16th International Conference on World Wide Web (WWW)*, 2007, pp. 141–150.

[8] M. Norouzi, A. Punjani, and D. J. Fleet, "Fast search in hamming space with multi-index hashing," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012, pp. 3108–3115.

[9] ——, "Fast exact search in hamming space with multi-index hashing," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 6, pp. 1107–1119, 2014.

[10] J. Qin, Y. Wang, C. Xiao, W. Wang, X. Lin, and Y. Ishikawa, "GPH: Similarity search in Hamming space," in *Proceedings of the 34th International Conference on Data Engineering (ICDE)*, 2018.

[11] A. Torralba, R. Fergus, and Y. Weiss, "Small codes and large image databases for recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008, pp. 1–8.

[12] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the Web," *Computer Networks and ISDN Systems*, vol. 29, no. 8, pp. 1157–1166, 1997.

[13] M. Theobald, J. Siddharth, and A. Paepcke, "Spotsigs: robust and efficient near duplicate detection in large web collections," in *Proceedings of the 31st Annual International ACM SIGIR conference on Research and Development in Information Retrieval*, 2008, pp. 563–570.

[14] P. Li and C. König, "b-Bit minwise hashing," in *Proceedings of the 19th International Conference on World Wide Web (WWW)*, 2010, pp. 671–680.

[15] P. Li, "0-bit consistent weighted sampling," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 665–674.

[16] ——, "Linearized GMM kernels and normalized random fourier features," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 315–324.

[17] D. Greene, M. Parnas, and F. Yao, "Multi-index hashing for information retrieval," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994, pp. 722–731.

[18] J. Qin and C. Xiao, "Pigeonring: A principle for faster thresholded similarity search," in *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*, 2018, pp. 28–42.

[19] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu, "Hmsearch: An efficient hamming distance query processing algorithm," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2013, p. 19.

[20] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[21] S. Kanda, K. Morita, and M. Fuketa, "Compressed double-array tries for string dictionaries supporting fast lookup," *Knowledge and Information Systems*, vol. 51, no. 3, pp. 1023–1042, 2017.

[22] G. E. Pibiri and R. 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.

[23] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "SuRF: Practical range query filtering with fast succinct tries," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 323–336.

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

[25] O. Delpratt, N. Rahman, and R. Raman, "Engineering the LOUDS succinct tree representation," in *Proceedings of the 5th International Workshop on Experimental and Efficient Algorithms (WEA)*, 2006, pp. 134–145.

[26] D. Belazzougui, "Faster and space-optimal edit distance "1" dictionary," in *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2009, pp. 154–167.

[27] D. Belazzougui and R. Venturini, "Compressed string dictionary look-up with edit distance one," in *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2012, pp. 280–292.

[28] R. Cole, L.-A. Gottlieb, and M. Lewenstein, "Dictionary matching and indexing with errors and don't cares," in *Proceedings of the 36th annual ACM Symposium on Theory of Computing (STOC)*, 2004, pp. 91–100.

[29] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong, "Compressed indexes for approximate string matching," *Algorithmica*, vol. 58, no. 2, pp. 263–281, 2010.

[30] A. C. Yao and F. F. Yao, "Dictionary look-up with one error," *Journal of Algorithms*, vol. 25, no. 1, pp. 194–202, 1997.

[31] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, 2008, pp. 257–266.

[32] T. Kudo, T. Hanaoka, J. Mukai, Y. Tabata, and H. 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.

[33] Y. Tabei, "Succinct multibit tree: compact representation of multibit trees by using succinct data structures in chemical fingerprint searches," in *Proceedings of the 12th International Workshop on Algorithms in Bioinformatics (WABI)*, 2012, pp. 201–213.

[34] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA)*, 2014, pp. 326–337.

[35] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM Conference on Recommender Systems*, 2013, pp. 165–172.

[36] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011, pp. 861–864.

[37] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 11, pp. 1958–1970, 2008.

[38] J. Qin, C. Xiao, Y. Wang, and W. Wang, "Generalizing the pigeonhole principle for similarity search in Hamming space," *IEEE Transactions on Knowledge and Data Engineering (Early Access)*, 2019.

APPENDIX

A. Evaluation for Single- and Multi-indexes

We evaluate the time performance of the single-index approach using a hash table data structure (i.e., SIH) for the similarity search per query as cost $cost_S$ in the following equation:

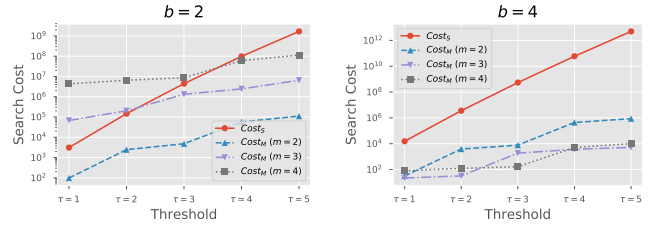$$cost_S = \mathsf{sigs}(b, L, \tau) \cdot L + |I|, \qquad (2)$$

Fig. 8. $cost_S$ and $cost_M$ for $b = 2$ (left) and $b = 4$ (right); the other parameters are chosen as $L = 32$, $b \in \{2, 4\}$, and $m \in \{2, 3, 4\}$.

where $\mathsf{sigs}(b, L, \tau)$ is the number of signatures as follows:

$$\mathsf{sigs}(b, L, \tau) = \sum_{k=0}^{\tau} \binom{L}{k} (2^b - 1)^k. \qquad (3)$$

Since $cost_S$ depends largely on the number of signatures, it is linearly proportional to L and exponentially proportional to τ and b.

We evaluate the time performance of the multi-index approach using hash tables (e.g., MIH) for the similarity search per query as cost $cost_M$ by the summation of the filtering and verification costs as follows:

$$cost_M = \sum_{j=1}^{m} \left\{ \mathsf{sigs}(b, L^j, \tau^j) \cdot L^j + L \cdot |C^j| \right\}, \qquad (4)$$

where term $\sum_{j=1}^{m} \mathsf{sigs}(b, L^j, \tau^j) \cdot L^j$ is the filtering cost for m blocks of query, and term $L \sum_{j=1}^{m} |C^j|$ is the verification cost depending on the total number of candidate solutions and sketch length.

Figure 8 shows the values of costs $cost_S$ and $cost_M$ by fixing the parameters of $(n, L) = (2^{32}, 32)$ and varying the parameters of $m \in \{2, 3, 4\}$ and $\tau \in \{1, 2, \ldots, 5\}$. In that figure, we compute $|I| = \mathsf{sigs}(b, L, \tau) \cdot n/(2^b)^L$ in $cost_S$ and $|C^j| = \mathsf{sigs}(b, L^j, \tau^j) \cdot n/(2^b)^{L^j}$ in $cost_M$ under the assumption that n sketches in a database are uniformly distributed in the Hamming space.

We can see $cost_S$ for the single-index approach exponentially increases for parameters τ and b. Thus, similarity searches on the single-index approach cannot be applied to b-bit sketches and large τ. We can also see that $cost_M$ for the multi-index approach increases for parameters of τ and b, but the increase is relatively small when large m (e.g., $m = 4$) is used. However, when a large number of blocks are used, a large number of candidate solutions are generated, resulting in a large verification cost. In addition, since many inverted indexes are built for large blocks, methods on the multi-index approach consume a large amount of memory.