iDEC: Indexable Distance Estimating Codes for Approximate Nearest Neighbor Search

Long Gong† Huayi Wang† Mitsunori Ogihara‡ Jun Xu†

†Georgia Institute of Technology, USA ‡University of Miami, USA

{gonglong,huayiwang}@gatech.edu jx@cc.gatech.edu ogihara@cs.miami.edu

ABSTRACT

Approximate Nearest Neighbor (ANN) search is a fundamental algorithmic problem, with numerous applications in many areas of computer science. In this work, we propose *indexable distance estimating codes (iDEC)*, a new solution framework to ANN that extends and improves the locality sensitive hashing (LSH) framework in a fundamental and systematic way. Empirically, an iDEC-based solution has a low index space complexity of $O(n)$ and can achieve a low average query time complexity of approximately $O(\log n)$. We show that our iDEC-based solutions for ANN in Hamming and edit distances outperform the respective state-of-the-art LSH-based solutions for both in-memory and external-memory processing. We also show that our iDEC-based in-memory ANN-H solution is more scalable than all existing solutions. We also discover deep connections between Error-Estimating Codes (EEC), LSH, and iDEC.

PVLDB Reference Format:
Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. iDEC: Indexable Distance Estimating Codes for Approximate Nearest Neighbor Search. *PVLDB*, 13(9): 1483-1497, 2020.
DOI: https://doi.org/10.14778/3397230.3397243

1. INTRODUCTION

Approximate Nearest Neighbor (ANN) search is a fundamental algorithmic problem, with numerous applications in many areas of computer science, including informational retrieval [44, 47], recommendations [17, 56, 62], near-duplication detections [52, 66], string similarity join [45, 76], etc. In this problem, given a query (data) object α, we search in a massive dataset $\mathcal{D}$, containing some n objects, for one or more objects that are *among the closest* to α according to some distance metric.

A natural solution to the exact Nearest Neighbor (NN) search would be exhaustive search (aka brute-force linear scan), where every object in $\mathcal{D}$ is compared with α. The query processing cost of this natural, perhaps naïve solution is $O(dn)$, where d is the dimension of the metric space that objects in $\mathcal{D}$ lie in. When d is small (say $d < 20$), under some mild assumptions on the data distribution in $\mathcal{D}$, the exact NN query can be answered, in $O(\log n)$ average time complexity per query, using a multi-dimensional search tree such as the k-d tree [13]. However, when the dimension d is large, as is in big data applications, such a search tree performs no better in terms of time complexity than the linear scan, due to the "curse of dimensionality" [21].

Locality sensitive hashing (LSH) [63, Chapter 3] is a celebrated family of ANN solutions. The key part in the design of LSH is its hash function family $\mathcal{F}$ with a nice collision property: Any function f sampled uniformly at random from $\mathcal{F}$ maps two distinct objects to the same hash value with a higher probability if they are close to each other than if they are not. A standard LSH-based algorithm can achieve a query time complexity of roughly $O(n^\rho)$ and a space complexity of roughly $O(n^{1+\rho})$ (in addition to $O(dn)$ needed to store the original database) [8], where ρ $(0 < \rho < 1)$ is the *quality* of the LSH family. While it is certainly desired that the quality ρ is made as small in value (high in quality) as possible, there is often a lower bound on ρ that can usually be expressed as a function of the desired approximation ratio c defined in Definition 2. For example, it was proven in [58] that, when the dimension (d in our case) is large, there is a lower bound of roughly $1/c$ for any LSH function designed for ANN in the Hamming space (ANN-H). In other words, any LSH-based ANN-H solution has time complexity $\Omega(n^{1/c})$ and space complexity $\Omega(n^{1+1/c})$.

1.1 Our Solution Approach: iDEC

We propose *indexable distance estimating codes (iDEC)*, a new solution framework to ANN. iDEC extends and improves the LSH framework in a fundamental and systematic way. Empirically, an iDEC-based in-memory solution has a low index space complexity of $O(n)$ and can achieve a low average query time complexity of approximately $O(\log n)$.

An inspiration for iDEC as well as its construction comes from the theory of error estimating codes (EEC) [20, 37–39, 77]. EEC are designed for (approximately) estimating the number of bit errors that have occurred to a packet (a binary string), during its transmission over a (noisy) wireless link. In a rough sense, EEC schemes allow for the estimation of the Hamming distance between the transmitted packet and the received packet. However, EEC schemes in their current forms are not well suited for database applications in general and ANN searches in particular, because they do not provide efficient indexing *per se* in the following sense: To use an EEC scheme for ANN search, there is only one readily available algorithm presently, which is to measure

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397243

the distance between the query point α and every point β in the database $\mathcal{D}$, by collating the EEC codeword of α and those of β. This algorithm is precisely the linear scan.

The *first contribution* of this work is to discover deep connections between iDEC, EEC, and ANN/LSH. We have made two such discoveries. Our first discovery is that EEC schemes can be converted into LSH schemes (codes) for ANN-H (ANN in the Hamming distance), as will be shown in §3. This discovery is significant for two reasons. First, it immediately makes EEC schemes efficiently indexable in the LSH way and hence applicable to ANN searches. Second, the state-of-the-art EEC schemes [37, 38] offer extremely short codewords that translate into relatively (compared to other LSH "codes") small index sizes when used in the ANN context. Our second discovery is that the decoding of most LSH "codes" can be viewed as hard-decision decoding, as will be explained in §2.2. This discovery motivated us to investigate whether we can perform a soft-decision decoding of LSH "codes". The investigation led to our *second contribution*: the iDEC algorithmic framework.

When describing EEC in the following, we focus entirely on its applications to ANN. iDEC generalizes EEC in two ways. First, whereas EEC measures by definition only the Hamming distance, our new codes can measure other distances, such as Euclidean, edit, and angular distances. Hence we call them distance estimating codes (DECs) instead. Second, although we can convert a DEC scheme to an LSH code and perform a hard-decision decoding of the latter in the LSH way, we discover a new and better way of decoding DECs: soft-decision decoding. As will be explained next, the soft-decision decoding of DECs makes them extremely efficiently searchable, much more so than when they are (hard-decision) decoded in the LSH way. For this reason, we add the letter "i" before DEC to emphasize this superior indexing capability.

Before delving into our other contributions, we briefly describe how the iDEC framework and its soft-decision decoding approach work in our iDEC-based solution for ANN-H; how they work in other iDEC-based solutions is similar. In ANN-H, the database $\mathcal{D}$ contains a large set of points in a d-dimensional space $S_1 = \{0,1\}^d$, where the dimension d can be large. The basic idea behind iDEC is to map each point in S_1 to a point in a low-dimensional space $S_2 = Z^m$ (Z is the set of all integers) whose dimension m is small (typically between 6 and 12), using a random projection function $\Psi(\cdot)$ that lies at the heart of three EEC schemes [37–39]. As shown in [37], this $\Psi(\cdot)$ preserves statistical closeness in the following sense: for any $x, y \in \{0,1\}^d$, we have $E[\|\Psi(x), \Psi(y)\|_2^2] = m * \|x, y\|_H$, where $\|\cdot,\cdot\|_2$ and $\|\cdot,\cdot\|_H$ are the Euclidean and the Hamming distances respectively. Hence, our insight is that if a point β is among the closest points in $\mathcal{D}$ (high-dimensional) to the query point α in the Hamming distance, then $\Psi(\beta)$ should also statistically be among the closest to $\Psi(\alpha)$ in Euclidean distance, among the points in $\Psi(\mathcal{D})$ (low-dimensional).

Based on this insight, the query processing algorithm is simply to search in $\Psi(\mathcal{D})$ for the nearest neighbors of $\Psi(\alpha)$ in the Euclidean distance. When points in $\Psi(\mathcal{D})$ are organized as a multi-dimensional search tree such as k-d tree [13], this query can be answered with an average time complexity of only $O(\log n)$, since the dimension m is a small constant. This decoding approach is soft-decision in nature because, among the points in $\Psi(\mathcal{D})$, we are looking not for the exact match with $\Psi(\alpha)$ (hard-decision decoding), but for its nearest neighbors.

This solution also has a low index space complexity of $O(n)$, since each vector in $\Psi(\mathcal{D})$ is stored only once. Furthermore, the constant factor in this big-O is small due to the aforementioned high space efficiency of the underlying EEC scheme. Thanks to this extremely low index space requirement, our iDEC-based solutions can perform *in-memory* processing of ANN queries over very large datasets, more so than all other solutions. However, if necessary (say when the dataset is massive), we can easily modify our solutions for external-memory query processing, by using a different search tree that minimizes I/O costs, such as R-tree [35], as will be described in §4.5.

One recent ANN solution, called SRS [68], bears some similarity to our iDEC framework. SRS also maps a high-dimensional $\mathcal{D}$ to a low-dimensional $\Psi(\mathcal{D})$ (using a very different Ψ), indexes $\Psi(\mathcal{D})$ using a multi-dimensional search tree (in this case cover tree [14] or R-tree [35]), and searches the tree for the nearest neighbors of $\Psi(\alpha)$. However, the iDEC framework generalizes and extends SRS in two fundamental ways. First, whereas SRS is designed only for ANN in the Euclidean distance, the iDEC framework is designed to handle various distance metrics including Hamming, Euclidean, edit, and angular. Second, the iDEC framework naturally subsumes EEC, a body of literature that has never been considered relevant to ANN/LSH. This newfound connection between the EEC and the ANN/LSH literatures allows both literatures to inform each other, which may lead to new deep discoveries in the future. More detailed comparisons concerning their designs, query time performances, and index space sizes will be performed in §5 and §7.3.

1.2 Our ANN Solutions

Under this algorithmic framework of soft-decision decoding, we propose novel iDEC-based solutions to two classical ANN problems. They are the *third* and the *fourth contributions* of this work. Both solutions have a low time complexity of approximately $O(\log n)$ and a low index space complexity of $O(n)$. Our first solution is for ANN-H, which is known to be a hard problem: the lower-bound time complexity for any classical LSH-based solution is $O(n^{1/c})$ as mentioned earlier. Our solution outperforms the state-of-the-art LSH-based ANN-H solutions in terms of both query time complexity and index space complexity, for both in-memory and external-memory query processing. Furthermore, it is much easier for our in-memory solution to scale to massive datasets than for all existing solutions (including those that are not LSH-based), since our solution has much less peak memory usage and much shorter indexing time.

The other solution is for an even harder problem: ANN in the edit distance, which we denote as ANN-E. The solution contains two iDEC scheme variants, whose respective innovations are two different multiset features, namely q-gram multiset and sliding-context multiset, that can support accurate ANN-E query processing by preserving statistical closeness: if two strings are close in edit distance then the two corresponding multisets are close in L_1 distance statistically. The L_1 ANN problem is much easier to solve even when the feature dimension is high, as will be shown in §6. We will show both variants significantly outperform the state-of-the-art ANN-E solutions in terms of both index space complexity and query time complexity.

Summary of Contributions.

- We discover deep connections between iDEC, EEC, and ANN/LSH (§3).
- We propose a novel ANN solution framework called iDEC that extends and improves the LSH framework in a fundamental and systematic way (§4).
- We propose a scalable iDEC-based solution for ANN-H that has low index space and query time complexities (§5).
- We propose two novel multiset features, on which, we build two efficient iDEC-based solutions for ANN-E (§6).

2. BACKGROUND

2.1 Problem Description

Let $\mathcal{D}$ be a database whose objects belong to a space $\mathcal{H}$ with a distance metric $\|\cdot,\cdot\|$. Definition 1 formally defines c-ANN, the approximate nearest neighbor (ANN) search problem with an approximation ratio of c ($c \geq 1$).

Definition 1 (c-ANN)**.** *Given a query object $\alpha \in \mathcal{H}$, and $c \geq 1$, find an object $\beta \in \mathcal{D}$ such that $\|\alpha, \beta\| \leq c\|\alpha, \beta^*\|$, where β^* is the nearest neighbor of α in $\mathcal{D}$.*

Our iDEC-based schemes, to be described in §5 and §6, solve this c-ANN problem in Hamming and edit distances respectively. In contrast, the LSH schemes [23,41] were originally designed to solve a different and weaker problem called (c, r)-ANN problem which, defined in Definition 2 next, has an additional parameter r denoting the search radius. Using LSH, one can solve c-ANN by breaking it down into a series of (c, r)-ANN problems in which the radius r takes exponentially increasing values $1, c, c^2, \cdots$ respectively.

Definition 2 ((c, r)-ANN)**.** *Given a query object $\alpha \in \mathcal{H}$, and $c \geq 1$, $r > 0$, find an object in $\mathcal{D}$ having distance at most cr from α, if there is an object in $\mathcal{D}$ having distance at most r from α.*

2.2 LSH and Hard-Decision Decoding

In this section, we formally introduce the concept of LSH (in Definition 3) and explain why its "decoding" process is hard-decision in nature.

Definition 3 (Locality-Sensitive Hashing (LSH) [41])**.** *Let r, s, p_1, p_2 be positive constants such that $r < s$ and $p_2 < p_1 \leq 1$. Let $\mathcal{H}$ be a space with a distance metric $\|\cdot,\cdot\|$. Let U be a certain universe. The hash function family $\mathcal{F} = \{h : \mathcal{H} \to U\}$ is called (r, s, p_1, p_2)-sensitive if for all $\alpha, \beta \in \mathcal{H}$:*

$$\begin{cases} \mathbf{Pr}_{h\in\mathcal{F}}[h(\alpha) = h(\beta)] \geq p_1 & \text{if } \|\alpha, \beta\| \leq r, \\ \mathbf{Pr}_{h\in\mathcal{F}}[h(\alpha) = h(\beta)] \leq p_2 & \text{if } \|\alpha, \beta\| > s. \end{cases} \quad (1)$$

The quality of an LSH family is measured by $\rho(\mathcal{F}) \triangleq \frac{\log 1/p_1}{\log 1/p_2}$.

We now provide a succinct and simplified description of how a classical LSH scheme is "decoded". Despite the simplification, its query time complexity of $O(n^\rho)$ and index space of $O(n^{1+\rho})$ (achieved when k defined below is set to $O(\log_{1/p_2} n)$ and n^ρ hash tables are used) match those of the classical LSH for solving (c, r)-ANN, where ρ is the quality of the LSH family as defined above. In a classical LSH scheme, typically a group of $k > 1$ LSH functions h_1, h_2, $\cdots$, h_k are used to map each point β in $\mathcal{D}$ to a k-dimensional vector of hash values $\langle h_1(\beta), h_2(\beta), \cdots, h_k(\beta)\rangle$. These n points in $\mathcal{D}$ are stored in a hash table using their corresponding vectors as hash keys. Then given a query point α, the search procedure is to probe all points in the hash bucket that the vector $\langle h_1(\alpha), h_2(\alpha), \cdots, h_k(\alpha)\rangle$ would be inserted into, in the hope that one or more points very close to α are mapped to the same vector and hence inserted into this bucket. We regard this search procedure as hard-decision decoding, since the query point α is "decoded" to precisely the vector $\langle h_1(\alpha), h_2(\alpha), \cdots, h_k(\alpha)\rangle$ and only the corresponding bucket is probed.

2.3 ToW-Based Error-Estimating Codes

The basic idea behind EEC is that the transmitter sends, along with a packet, a set of codewords that can be viewed as a succinct description of the packet. The receiver can estimate the number of bit errors that have occurred to the received packet (equivalently its Hamming distance to the transmitted packet) during the transmission from the discrepancies (caused by the bit errors) between the received packet and the succinct description. In the seminal EEC scheme [20], this problem was formulated and solved as a coding theory problem, in which each codeword is, as usual, the parity of a group of bits randomly sampled from the packet. In the subsequent EEC schemes [37–39] however, this problem was formulated instead as the design of a two-party computation protocol between the sender and the receiver [73] for estimating this Hamming distance. These schemes achieved much higher coding efficiency using synopsis data structures (aka *sketch* [55]).

The EEC schemes proposed in [37–39] were all adapted from the Tug-of-War (ToW) sketch [7] for estimating the second moment, equivalently the square of the L_2 (Euclidean) norm, of a data stream. ToW is a suitable sketch for this two-party computation because, when binary strings x and y are viewed as binary vectors, their Hamming distance, denoted as $||x, y||_H$, is equal to $||x, y||_2^2$, the square of their Euclidean distance. Given a d-dimensional vector x (not necessarily binary from this point on), a ToW-based EEC codeword of x, denoted as $\psi(x)$, is defined as

$$\psi(x) \triangleq \langle x, R\rangle \quad (2)$$

where R is a d-dimensional vector whose d scalars are *i.i.d.* random variables each of which takes the value $+1$ or -1 each with probability $1/2$. In the sequel, we denote the family of such R vectors as $\mathcal{R}$, and the family of ψ functions as $\psi_\mathcal{R}$. It is shown in [37] that $\psi(\cdot)$, which is a random projection function [6, 70], weakly preserves pairwise distances in the following sense: Given any two points $x, y \in \{0, 1\}^d$, we have

$$\mathbf{E}[(\psi(x) - \psi(y))^2] = \|x, y\|_2^2 = ||x, y||_H. \quad (3)$$

3. EEC-DERIVED LSH

In this section, we describe our *first contribution*: the conversion of a ToW-based EEC codeword (function) family $\psi_\mathcal{R}$ to an LSH family in Hamming space. Each LSH function in this family is defined as $h_{\psi,b}(\cdot) = \lfloor \frac{\psi(\cdot)+b}{w} \rfloor$, where ψ is drawn uniformly at random from the family $\psi_\mathcal{R}$ and b is a random offset drawn uniformly from $[0, w]$; here w is a predefined constant called *bucket width* in [40]. Note that in [23], the LSH function takes the same form except that there $\psi(\cdot)$ is a stable distribution sketch function, whereas here $\psi(\cdot)$ is a ToW-based EEC codeword function.

Now we calculate the quality of this LSH family by deriving the collision probability of two points $\alpha, \beta \in \{0,1\}^d$ given such an LSH function. Let $s = \|\alpha, \beta\|_H$. For a $\psi(\cdot)$ with the underlying random "ToW vector" R, $\psi(\alpha) - \psi(\beta) = \langle \alpha - \beta, R \rangle$ has the same distribution as the "distance" from the starting point in an s-step simple random walk. More precisely, it is distributed as $Y_s = \sum_{i=1}^{s} X_i$ where X_i takes value either 1 or -1 with equal probability (of 0.5). Using the results in [23], we obtain the collision probability $p(s) = \sum_{t=0}^{w} (1 - \frac{t}{w}) \mathbf{Pr}[|Y_s| = t]$, where

$$\mathbf{Pr}[|Y_s| = t] = \begin{cases} \binom{s}{s/2} \left(\frac{1}{2}\right)^s & \text{if } t = 0 \\ \binom{s}{(s+t)/2} \left(\frac{1}{2}\right)^s + \binom{s}{(s-t)/2} \left(\frac{1}{2}\right)^s & \text{otherwise} \end{cases}.$$

Here we define $\binom{n}{x} = 0$ when x is not an integer.

It is not hard to check that when w is an even integer, the collision probability $p(s)$ decreases monotonically with $s = \|\alpha, \beta\|_H$. Thus, this hash family can be viewed as a (r_1, r_2, p_1, p_2)-sensitive LSH, where $p_1 = p(r_1), p_2 = p(r_2)$. For example, given $w = 4$, when $r_1 = 5, r_2 = 10$, we have $p_1 = 0.5469$, $p_2 = 0.4189$, and $\rho = 0.6937$.

Note that this ρ is larger (*i.e.*, lower in quality) than the lower bound (*i.e.*, best possible) quality value of roughly $1/2$ $(= r_1/r_2)$. We do not know whether this quality can be further improved. Nonetheless, we have made no attempt at optimizing it for two reasons. First, this conversion is itself a significant discovery as explained earlier. Second, as we will soon show, the soft-decision decoding of LSH "codes" anyway leads to much better average query time complexity and index space complexity, which makes it irrelevant to further optimize this quality.

4. IDEC FRAMEWORK

In this section, we present our *second contribution*: the iDEC algorithmic framework. This framework is shared by all iDEC-based ANN solutions. Each solution however uses a different random projection function $\psi_i(\cdot)$, which we will introduce next.

4.1 Dimensionality Reduction

As mentioned earlier, all iDEC-based solutions take the following dimensionality reduction approach. It maps the original dataset $\mathcal{D}$ that lies in a high-dimensional space S_1 with distance metric ν_1 to $\Psi(\mathcal{D})$ that lies in a low-dimensional space S_2 (say of m dimensions) with distance metric ν_2. It does so using a random projection vector function $\Psi(\cdot)$ that is statistically closeness-preserving in the sense that for any two points α and β in S_1 that are close together according to ν_1, the images $\Psi(\alpha)$ and $\Psi(\beta)$ (both in S_2) are also statistically close together according to ν_2. Note that ν_1 and ν_2 can be different. For example, in the iDEC-based solution for ANN-H that we will describe in §5, ν_1 is Hamming distance $\|\cdot,\cdot\|_H$ whereas ν_2 is Euclidean distance $\|\cdot,\cdot\|_2$. The vector function $\Psi(\cdot)$ is defined as $\langle \psi_1(\cdot), \cdots, \psi_m(\cdot) \rangle$ where $\psi_1(\cdot)$, $\psi_2(\cdot)$, ..., $\psi_m(\cdot)$ are independent random projection functions sampled uniformly randomly from the corresponding family. Each $\psi_i(\cdot)$ needs to be statistically closeness-preserving so that $\Psi(\cdot)$ is also.

4.2 Indexing

This statistically closeness preservation property implies that if the point β is among the closest points in S_1 to the query point α, then $\Psi(\beta)$ should statistically be among the closest to $\Psi(\alpha)$ in S_2. An ANN query over the high-dimensional dataset $\mathcal{D}$ is thus converted to one over the low-dimensional dataset $\Psi(\mathcal{D})$. By building a multi-dimensional index $\mathcal{T}$ for the low-dimensional dataset $\Psi(\mathcal{D})$, the latter ANN query can be computed very efficiently.

Several candidate multi-dimensional tree data structures exist for implementing $\mathcal{T}$, including k-d tree [13]), ball tree [59], vp-tree (vantage point tree) [74], and cover tree [14] that are optimized for in-memory operations, and R-tree [35] that is optimized for external-memory operations. In principle, an iDEC-based solution works with any of these trees. In this work, for in-memory operations the k-d tree is used because it delivers better query processing performance than the other three when the accuracy requirement is not very high, as will be shown in §4.4; for external-memory operations, the R-tree is used like in most other external-memory solutions. We will describe how our ANN-H solution works with R-tree in §4.5 for external-memory operations and evaluate its performance in §7. Except that, below we will focus mostly on how our solutions work with a k-d tree for in-memory operations. In all our solutions, the dimension of k-d tree is m, the same as the dimension of S_2. It is important to keep m small, since the time complexity of the search (in the k-d tree) grows exponentially with m.

The construction of the index $\mathcal{T}$ consists of two steps. The first is the computation of $\Psi(\alpha)$ for all $\alpha \in \mathcal{D}$, which requires $O(dmn)$ time. The second is the indexing of the n points with m-dimensions in a k-d tree, which requires $O(mn \log n)$ time [29]. Since m is a small constant, we remove it and obtain the time complexity of $O(dn + n \log n)$ for indexing. The space complexity of $\mathcal{T}$ is $O(n)$.

4.3 Query Processing

Algorithm 1: iDEC Query Procedure

input : Query point α; multi-dimensional index $\mathcal{T}$ for $\Psi(\mathcal{D})$
output: α's ANN $\beta^* \in \mathcal{D}$

1 Search $\mathcal{T}$ for the set of t-NNs of $\Psi(\alpha)$, denoted as $\mathcal{P}$
2 $\beta^* \leftarrow \underset{\beta \in \Psi^{-1}(\mathcal{P})}{\arg\min} \|\beta, \alpha\|$

Algorithm 1 presents the generic pseudo-code of query processing under the iDEC framework. The algorithm is simply to first search $\mathcal{T}$ for $\mathcal{P}$, the set of t exact nearest neighbors (t-NN) of $\Psi(\alpha)$ in $\Psi(\mathcal{D})$ (Line 1 of Algorithm 1), and then find among their inverses $\Psi^{-1}(\mathcal{P})$ the point β^* that is closest to α (Line 2 of Algorithm 1). The rationale is that if β^* is indeed the nearest point to α in $\mathcal{D}$, then due to the statistical closeness preservation property of $\Psi(\cdot)$, with a decent probability p, $\Psi(\beta^*) \in \mathcal{P}$ when t is large enough. How large t should be depends on p, n, and the data distribution in S_1, as we will elaborate next.

Algorithm 1 can be viewed as decoding $\Psi(\alpha)$, the iDEC codeword for α. This decoding is soft-decision in nature because, among the vectors in $\Psi(\mathcal{D})$, we are not looking for the exact match with $\Psi(\alpha)$ (hard-decision decoding) as in classical LSH schemes, but rather its nearest neighbors.

Query Time Complexity. Through the following analysis, we will show that, under the iDEC framework, the average time complexity of answering an ANN query is roughly $O(\log n)$. We will treat the dimension m as a small constant

in the analysis. In Algorithm 1, the most expensive step computationally is the t-NN query (Line 1). It was shown in [29] that a t-NN query has an average time complexity of $O(\log n + (t^{1/m}+1)^m)$, under some mild assumptions on the distribution of points in the dataset. A rough explanation of this complexity is that it takes $O(\log n)$ steps to walk down the k-d tree to find a leaf node (corresponding to a tiny cell) that is quite close to α and may contain one or more t-NN candidates; then roughly $(t^{1/m}+1)^m$ cells neighboring this cell need to be searched for all other possible t-NN candidates. The term $(t^{1/m}+1)^m$ can be considered $O(t)$ since $(t^{1/m}+1)^m \leq (t^{1/m}+t^{1/m})^m = 2^m t$ and m is a small constant. Hence the complexity reduces to $O(\log n + t)$.

We note that to guarantee a certain query quality (approximation ratio) for the ANN query, this t value grows as $O(n)$ in theory. However, with real-world datasets, t is typically small even when n is gigantic. For example, our iDEC-based solution for ANN-H can achieve an average approximation ratio at least 2.89 with t being about 258, even when n is around 1 billion, as shown in §7.3.3. Hence, we conclude that, in practice, t can be considered roughly $O(\log n)$, and so can the average query time complexity.

4.4 Why k-d Tree?

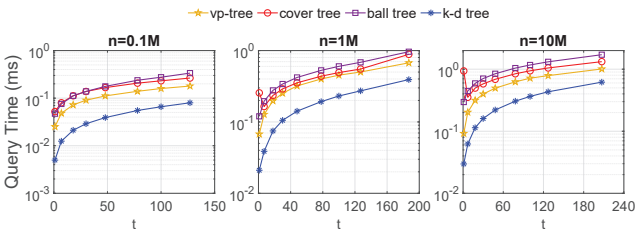

Figure 1: Query time vs. t for t-NN search.

As mentioned earlier, among the four candidate search trees optimized for in-memory operations, we use the k-d tree in the iDEC-based solutions. We make this design decision because we have found, via experiments on several datasets that vary in dimensions, sizes, and value ranges of coordinates (of data points), that overall k-d tree has the best t-NN query performance among them, when t is $O(\log n)$ as used in the iDEC-based solutions.

We now provide some evidence on the outperformance of k-d tree over the others. In the interest of space, we only present our experimental results on three 6-dimensional synthetic datasets (mimicking real-world image datasets $\Psi(D)$ for $m = 6$) in which each coordinate of any point is an integer random variable following a discrete uniform distribution $\mathcal{U}\{-100, 100\}$; we only consider integer coordinates because coordinates of any point in $\Psi(D)$ are all integers in both iDEC-based solutions. We have compared the k-d tree with the other three candidates, namely, vp-tree (vantage point tree) [74], cover tree [14], and ball tree [59]. For each of the four trees compared, we run t-NN search with 200 queries over the three synthetic datasets. For all four trees, the average query time (in milliseconds) as a function of t, with t varying roughly from 1 to $10 \log n$, is shown in Figure 1. Figure 1 clearly shows that the query time of k-d tree is significantly (up to 12 times) shorter than those of the other three trees, for all t values in this range.

4.5 iDEC for External-Memory Operations

When a dataset $\mathcal{D}$ is massive, the combined size of the original database $\mathcal{D}$ and its index (including both the image $\Psi(\mathcal{D})$ and the k-d tree $\mathcal{T}$) can be too large to fit in main memory. In this case, iDEC-based solutions have to operate with both $\mathcal{D}$ and its index residing on disk. As mentioned earlier, k-d tree is no longer suitable since it is optimized for in-memory operations. Instead, R-tree [53] is used in the iDEC-based solutions for external-memory operations, since it is the most commonly used technique for indexing and retrieving multi-dimensional data residing on disk.

Space and Time Complexities. The index space and the indexing time complexities for external-memory iDEC are $O(n)$ and $O(dn + n \log n)$ respectively, the same as those of in-memory iDEC. The I/O cost and the (overall) time complexity per query using R-tree are both $O(n)$ with a small constant factor, as shown in [68].

5. IDEC FOR ANN-H

In this section, we describe our *third contribution*: an iDEC-based solution to ANN-H. In ANN-H, $\mathcal{D}$ consists of binary strings, members of $\{0,1\}^d$ for some fixed d, with the Hamming distance as the distance metric. ANN-H is uniquely important in the ANN/LSH literature for the following reason. Since ANN-H deals with only binary symbols '0' and '1', the literature often treats it as a "clean" case where ANN with "messier" (but not necessarily more challenging) metrics (*e.g.*, the angular distance) can be first converted to it and then solved. For example, a standard approach to document similarity search was to first map each document in $\mathcal{D}$ to a binary vector with the celebrated SimHash [19] as Ψ and organize the converted vectors into a suffix-tree-like index [52].

As mentioned earlier, an iDEC-based solution is completely specified when the underlying ψ_i is. In our ANN-H solution, each ψ_i is an instance of the ToW function ψ defined in (2) that maps any two strings close in Hamming distance to two integer vectors close statistically in Euclidean distance as characterized precisely in (3). Hence in this solution the corresponding t-NN search (Line 1 in Algorithm 1) is in the Euclidean distance.

We note that our ANN-H solution, without any modification, can be used for ANN-L_2 (Euclidean distance) also. On the other hand, the aforementioned SRS scheme [68], which is designed for ANN-L_2, can be used for ANN-H as well. Here we briefly compare the designs and the relative strengths and weaknesses of SRS against our solution. Like the ψ_i used in our solution, the ψ_i used in SRS also takes the form of $\langle \cdot, R \rangle$, but its R is a Gaussian random vector, not a ToW (*i.e.*, ± 1) random vector. Hence when SRS is used for ANN-H, its ψ_i will map a binary (integer) vector to a (generally) non-integer real number. This "digital-to-analog conversion" would increase its index size, since a real (floating-point) number takes up more storage space than an integer. We will elaborate on this effect in §7.3. On the other hand, when our ANN-H solution is used for ANN-L_2, the scalars of the data points (vectors) are in general non-integer real numbers, so the ψ_i in our solution, despite using an integral R, would also map each data point to a non-integer real number in general. In this case, the "digital-to-analog conversion" is forced upon our solution and erases these advantages over SRS.

6. IDEC FOR ANN-E

In this section, we describe the *fourth contribution* of this work: two iDEC scheme variants for ANN-E (E for edit distance), namely multiset-iDEC and context-iDEC. The edit distance between two strings α and β is defined as the minimum number of symbol insertions, deletions, and replacements that are needed to change α to β. As mentioned earlier, in both variants, the respective innovations are two different multiset "features" that statistically capture the closeness, in edit distance, of any two strings. Unlike in ANN-H, in ANN-E, we drop the following two requirements. First, the alphabet, or the set of symbols used in the strings, is no longer in general binary. Second, we no longer require every string to have the same length.

ANN-E is in general much more difficult than ANN-H, since even when symbols are binary, in ANN-E we compare two binary strings α and β not only for bit flipping errors, but also bit insertions and deletions.

6.1 Multiset-iDEC

The first variant, which we call multiset-iDEC, is to convert a string α to a multiset of q-grams [63] (also called q-shingles in the literature), which we denote as $M^{(q)}(\alpha)$. For example, given the 20-bit binary string $\alpha =$ "10100010000010001100" and with gram size $q=3$, the corresponding multiset $M^{(q)}(\alpha)$ contains elements "101" (the first 3 bits), "010" (the next 3 bits), $\cdots$, and "100" (the last 3 bits). It can be shown that $M^{(q)}(\alpha) = \{$"000" $\times$ 5, "001" $\times 3$, "010" $\times 3$, "011" $\times 1$, "100" $\times 4$, "101" $\times 1$, "110" $\times$ 1, "111" $\times$ 0$\}$, in which the number after "$\times$" is the multiplicity of the 3-gram. With the implicit understanding that the eight possible 3-gram values are ordered lexicographically, we can drop the 3-gram values and more succinctly represent the multiset $M^{(q)}(\alpha)$ by its multiplicity vector $(5, 3, 3, 1, 4, 1, 1, 0)$. This multiplicity vector also succinctly encodes $M^{(q)}(\alpha)$'s multiplicity function $\mu(\cdot)$ in the sense $\mu($"000"$) = 5$, $\mu($"001"$) = 3$, $\mu($"010"$) = 3$, and so on. Hence we consider multiplicity vector and multiplicity function the same thing in the sequel.

The rationale for converting a string into such a multiset is that if two strings α and β are close to each other in edit distance, then their corresponding multisets should with high probability be close to each other in L_1 distance, defined next. Let M_1 and M_2 be two multisets both defined over the universe $\mathcal{A}$, and $\mu_1(\cdot)$ and $\mu_2(\cdot)$ be their respective multiplicity functions (vectors). Then the L_1 distance between M_1 and M_2, denoted as $\|M_1, M_2\|_1$, is the same as the L_1-difference between their multiplicity functions (vectors) $\mu_1(\cdot)$ and $\mu_2(\cdot)$. The latter is defined as $\sum_{a\in\mathcal{A}} |\mu_1(a) - \mu_2(a)|$.

Each bit flipping, insertion, or deletion error that "modifies" a string α into another string β adds roughly q to the L_1 distance between $M^{(q)}(\alpha)$ and $M^{(q)}(\beta)$. In other words,

$$q * ||\alpha, \beta||_E \approx \|M^{(q)}(\alpha), M^{(q)}(\beta)\|_1 \tag{4}$$

where $||\cdot, \cdot||_E$ denotes the edit distance. Hence, the mapping (by $M^{(q)}(\cdot)$) of a string α to its q-gram multiset $M^{(q)}(\alpha)$ is statistically closeness-preserving, a property we need and will use in designing the random projection function $\psi_i(\cdot)$.

When the dimension (of the multiplicity vectors) of the q-gram multisets (equivalently the size of their universe) is small, the function $M^{(q)}(\cdot)$ can be used directly as the random projection vector function $\Psi(\cdot)$. More precisely, given a query string α, we simply search for t-NN, in L_1 distance, of (the multiplicity vector of) $M^{(q)}(\alpha)$ in $M^{(q)}(\mathcal{D}) \triangleq \{M^{(q)}(\beta) \mid \beta \in \mathcal{D}\}$ organized as a k-d tree, taking advantage of the statistically closeness preserving property of $M^{(q)}(\cdot)$ shown in formula (4). This way of using raw q-gram multisets as $\Psi(\cdot)$ has the advantage that there will be no further information loss beyond that was incurred when converting a string to its q-gram multiset.

In most ANN search applications, however, the q-gram multisets have a high dimension. For example, the UNIREF dataset, which we use in our evaluations, has an alphabet size of 25, its q-gram multisets have a dimension of $25^2 = 625$, even when $q = 2$. In this case, we need to reduce this dimension, using sketching functions $\xi_i(\cdot)$, $i = 1, 2, \cdots, m$. In other words, each $\psi_i(\cdot)$ in Ψ is defined as $\xi_i(M^{(q)}(\cdot))$.

Here we need to sketch the L_1 distance (RHS of (4)), not the squared L_2 distance as in ANN-H, so using the standard ToW sketch (function) as ξ_i is no longer appropriate. Instead each ξ_i here is a random instance of a variant of ToW sketch proposed in [27] for estimating the L_1 distance of two multisets. Each ξ_i has the following property: For any two multisets M_1 and M_2, we have $E[\|\xi_i(M_1) - \xi_i(M_2))\|_2^2] = \|M_1, M_2\|_1$. Hence, for any two strings α and β, we have $E[\|\psi_i(\alpha) - \psi_i(\beta)\|_2^2] = \|M^{(q)}(\alpha) - M^{(q)}(\beta)\|_1$, of which the RHS is close to $q * \|\alpha, \beta\|_E$ according to the approximation formula (4). The implied ANN query procedure is that, given a query string α, to search for the t-NN (Line 1 of Algorithm 1), in L_2 (Euclidean) distance, of $\Psi(\alpha)$ in $\Psi(\mathcal{D})$.

6.2 Context-iDEC

The multiset-iDEC has a shortcoming: If we partition a string α into a few long substrings and permute their orders, the resulting string β is likely very far from β in edit distance, yet $M^{(q)}(\beta)$ is pretty close to $M^{(q)}(\alpha)$ in L_1 distance. This β is a spurious pattern with respect to α. For many applications we can think of, either very few such spurious patterns should exist "naturally", or they should be discovered and returned as query answers nonetheless. However, for certain applications such as time series analysis, the order each symbol appears in a string matters.

Our second approach, called context-iDEC, is to convert a fixed-length context for each symbol in the string into a multiset. The multisets of all contexts are then unioned together to arrive at a final sliding-context multiset. The context-iDEC scheme is robust against most of conceivable spurious patterns because the sliding windows of (overlapping) contexts implicitly encode the exact order in which these symbols occur in the string.

We now formally define the concept of a context. Let $s = s_1 s_2 \ldots s_\ell$ be a string of length ℓ. Let i be an integer such that $w < i \leq \ell - w$, where w $(w > 0)$ is the window size of a *left* or *right context*. For each symbol s_i in the string, we define its *left context*, denoted as $C^{(L)}$, as the w-symbol-long substring preceding the center symbol s_i. In other words, $C^{(L)} \triangleq s_{i-w} s_{i-w+1} \ldots s_{i-1}$. Similarly, we define its *right context*, denoted as $C^{(R)}$, as the w-symbol-long substring following the center symbol s_i. In other words, $C^{(R)} \triangleq s_{i+1} s_{i+2} \ldots s_{i+w}$. We call $C^{(L)} \| s_i \| C^{(R)}$ (where "$\|$" is the string concatenation operator), the $(2w+1)$-symbol-long substring concatenating the left context of s_i, the center symbol s_i, and the right context of s_i together, the *context*

of s_i, which we denote as $C_i^{(s)}$. Clearly, the string s has a total of $\ell-(2w+1)+1 = \ell-2w$ sliding contexts. These sliding contexts are the "raw materials" to be processed further into the context-iDEC "feature" for the string s.

Before we describe context-iDEC, we explain why these sliding contexts are good "raw materials". We say that two $(2w+1)$-symbol-long contexts, taken from two different strings, are similar (or identical if indeed so) if their center symbols are *identical*, their left contexts are identical or similar, and so are their right contexts. Conceivably, if two strings α and β are close to each other in edit distance, most of their contexts should be pairwise similar or identical. Hence, if a suitable feature can be found to capture the similarity or lack there of between any two sets of sliding contexts (of two different strings), then a good index can possibly be built on this feature for ANN-E.

We now describe how the context-iDEC (feature vector) of the string s in the example above is extracted from its $\ell-2w$ sliding contexts. Each $(2w+1)$-symbol-long context is mapped into a multiset of $(q+1)$-grams, each of which is a concatenation of a q-gram in its left or right context and the center symbol, as follows. Given any context $C_i^{(s)} = C^{(L)} \| s_i \| C^{(R)}$, its multiset, denoted as $M^{(q)+1}(C_i^{(s)})$, is defined as the union of the q-gram multisets of its left context $C^{(L)}$ concatenating with the center symbol s_i and the q-gram multisets of its right context $C^{(R)}$ concatenating with the center symbol s_i. More precisely, $M^{(q)+1}(C_i^{(s)}) \triangleq \big(M^{(q)}(C^{(L)}) \| s_i\big) \cup \big(s_i \| M^{(q)}(C^{(R)})\big)$. Given a multiset M of strings and a symbol a, their concatenation $M\|a$ consists of all strings of form $e\|a$ where e is a string from M. More precisely, $M\|a \triangleq \{(e\|a) \times \mu(e) \,|\, \forall (e \times \mu(e)) \in M\}$, where $\mu(\cdot)$ (defined in §6.1) is the multiplicity function of M. Similarly, $a\|M \triangleq \{(a\|e) \times \mu(e) \,|\, \forall (e \times \mu(e)) \in M\}$.

The $(q+1)$-gram multiset (function) $M^{(q)+1}(\cdot)$ is intuitively a good "mini feature" since, like our explanations for the approximation formula (4), two contexts that are similar or identical (i.e., close to each other in edit distance) should have similar or identical $(q+1)$-gram multisets (i.e., close to each other in L_1 distance). The context-iDEC (feature vector) of s, denoted as $M^{(SC)}(s)$, is defined as the union of the $(q+1)$-gram multisets of all its contexts $C_{w+1}^{(s)}, C_{w+2}^{(s)}, \cdots, C_{\ell-w}^{(s)}$. More precisely, we have

$$M^{(SC)}(s) \triangleq \bigcup_{j=w+1}^{\ell-w} M^{(q)+1}(C_i^{(s)}).$$

Since each such $(q+1)$-gram multiset is a good "mini feature", the resulting context-iDEC $M^{(SC)}(\cdot)$ is intuitively a good feature in the sense if two sets of sliding contexts are close to each other (likely because the corresponding strings are close to each other in edit distance as explained above), their corresponding $M^{(SC)}(\cdot)$ multisets should be close to each other in L_1 distance.

Like in multiset-iDEC, we can use the context-iDEC feature $M^{(SC)}(\cdot)$ directly as $\Psi(\cdot)$, when the dimension of the multiset is small. Otherwise (which is usually the case), we need to reduce the dimension, again using $\{\xi_i\}_{i=1}^m$ that are instances of ToW-for-L_1 sketching functions. That is, $\psi_i(\cdot) \triangleq \xi_i(M^{(SC)}(\cdot))$ for $i = 1, 2, \cdots, m$. Therefore, ψ_i has the same statistical closeness-preserving property $\mathbf{E}[\|\psi_i(\alpha) - \psi_i(\beta)\|_2^2] = \|M^{(SC)}(\alpha) - M^{(SC)}(\beta)\|_1$ as that in multiset-iDEC.

7. EVALUATION

In this section, we evaluate the performance of our iDEC-based algorithms for ANN-H and ANN-E against the respective state-of-the-art solutions. After we explain performance metrics in §7.1 and implementation details in §7.2, we describe benchmark algorithms, evaluation datatsets, and experimental results on ANN-H in §7.3 and on ANN-E in §7.4. In the rest of this section, we refer to an iDEC-based algorithm simply as iDEC whenever it is clear from the context which algorithm we are referring to.

7.1 Performance Metrics

We evaluate the performance of ANN algorithms in three aspects: scalability, query efficiency, and query quality. To measure scalability (how well an algorithm can scale to very large datasets), we use *index size* (excluding the size of the original database) and *indexing time* (time needed to build the index). Each index size or indexing time value presented in this section is the average over five independent runs. To measure query efficiency, we use *query time*; for external-memory experiments, we use also the *I/O cost* as quantified by the number of disk accesses. To measure *query quality*, we use *approximation ratio* in most situations. However, some ANN algorithms can return "a null answer", in which case the approximation ratio becomes ∞. For example, a classical LSH-based algorithm (e.g., [41]) returns "null" when all buckets that the query point is hashed into are empty. For this reason, we also use *recall*.

The concepts of approximation ratio and recall are defined as follows. Suppose an ANN algorithm returns a data object β given a query point α and β^* is the true nearest neighbor of α in $\mathcal{D}$. Then the approximation ratio is defined as $\|\alpha, \beta\| / \|\alpha, \beta^*\|$. As mentioned in §2, in c-ANN (defined in Definition 1), we consider the answer β to be correct and correspondingly the recall value for this query to be 1, if $\|\alpha, \beta\| \leq c \cdot \|\alpha, \beta^*\|$; otherwise we consider the recall value to be 0.

Each query time, I/O cost, approximation ratio or recall value presented in this section is the average over many queries. In ANN-H evaluations, we will mainly use approximation ratio as the quality metric, since a few benchmark algorithms (*e.g.,* QALSH and C2LSH) are optimized for providing theoretical guarantees in terms of the approximation ratio. In ANN-E evaluations, we will only use recall with $c = 1.5$, since the benchmark algorithm there suffers from the aforementioned "null answer problem". Finally, we note that Mean Average Precision (MAP) [9], another query quality metric widely used in the ANN literature recently, is equivalent to the concept of recall when $c = 1$ in the c-ANN search.

7.2 Implementation Details

Our iDEC-based algorithms are implemented in C++. For k-d trees, we use an efficient open-source C++ implementation [15]. For R-trees, we use the source code provided by the authors of [68]. For computing the exact edit distance in ANN-E, we use the C++ library called edlib [67]. For all benchmark algorithms (to be described next), we use the most efficient open-source implementation or the source code provided by the corresponding authors. All source codes are in C++ except that the index construction of OPQ is implemented in MATLAB. We compile all C++ source codes with g++ 7.5 with -O3 and MATLAB source

codes with MATLAB 9.1. All ANN-H experiments are done on a workstation with Intel(R) Core(TM) i7-9800X 3.8 GHz CPU, 128 G DRAM, and 4 TB hard disk drive (HDD). All ANN-E experiments are done on a workstation with Intel(R) Core(TM) i7-6700K 4.0 GHz CPU and 64 G DRAM. Both machines run Ubuntu 18.04.

7.3 Performance Evaluation for ANN-H

7.3.1 Benchmark Algorithms and Parameters

We compare iDEC with five state-of-the-art ANN search algorithms: SRS [24, 68], Optimized Product Quantization (OPQ) [33], Hierarchical Navigable Small World (HNSW) [51], Query-Aware LSH (QALSH) [40] and C2LSH [32]. For SRS, OPQ, and HNSW, we have tuned their parameters for the best respective performances. For QALSH and C2LSH, we have set their parameters according to the recommendations provided by the respective authors. The parameter settings of all these algorithms, including those of iDEC, are detailed as follows.

iDEC. We use the iDEC-based algorithm for ANN-H described in §5, with the dimension m of the image $\Psi(\mathcal{D})$ set to 6 in all experiments, since, through experimental evaluation for m varying between 4 and 16 on various datasets, we find that m=6 appears to strike the best tradeoff between the query quality and the query efficiency. The values of m in ANN-E evaluation shown later in §7.4 are obtained through a similar tuning process. We measure the query performances under different values of t (the number of NNs searched in Line 1 of Algorithm 1). As explained earlier in §4.2, iDEC uses k-d trees and R-trees for in-memory and external-memory operations, respectively.

SRS. SRS [24, 68] is an LSH-based algorithm for ANN-L_2 (in Euclidean distance). As explained in §5, it can be used for AHH-H without modification. Using cover trees and R-trees respectively, SRS can operate both in-memory [24] and external-memory [68].

We use m=6 for SRS, which is obtained through the same tuning process as that for iDEC. We do not use early terminations and instead stop the query process when t (same semantics as t in Line 1 of Algorithm 1) NNs are checked out. Like with iDEC, we measure its query performances under different values of t.

OPQ. OPQ [33] is a quantization-based in-memory-only ANN algorithm. We fix the length of PQ encoding for each vector at 8 bytes. We use $2^6, 2^7, 2^8, 2^7, 2^8$, and 2^7 centroids for Audio, Mnist, Enron, GIST1M, SIFT1M, and GloVe respectively. For the two large datasets, we are not able to build their indices since both caused OPQ to run out of memory (and crash) during index construction. We measure its query performances for different numbers of NN candidates to be checked out.

HNSW. HNSW [51] is a graph-based in-memory-only ANN algorithm. We measure its query performances with the parameter M (the number of "neighbors" per point) varying from 8 to 48. For each value of M, we select the *efConstruction* parameter (varying from 100 to 600) that strikes the best tradeoff between the indexing efficiency and query quality. Same as in OPQ, we are not able to build their indices for the two large datasets due to running out of memory. We note that the index size and the indexing time of HNSW on a dataset are generally different when the parameters M or *efConstruction* are different. For fairness (to HNSW), we report the smallest index size and the shortest indexing time of HNSW with the above parameter settings on each of the 6 datasets in Table 2 and Table 3, respectively.

QALSH. QALSH [40] is an LSH-based algorithm optimized for external-memory operations in Euclidean space. We set the approximation ratio to 2, and use e^{-1} for the error probability and $100/n$ for the false positive percentage.

C2LSH. C2LSH [32] is another LSH-based algorithm optimized for external-memory operations in Euclidean space. We use the collision threshold version of C2LSH. We set the approximation ratio, the error probability, and the false positive percentage to the same values as those in QALSH. An additional parameter here that does not need to be set in QALSH is the interval size, which we set to 1.

Common Parameters. The disk page size is fixed at 4,096 bytes for all algorithms on all datasets in external-memory experiments.

7.3.2 Evaluation Datasets

We use 8 publicly available datasets of diverse dimensions, sizes (number of points), and types (image, audio, and textual), that are widely used in the ANN literature. All these datasets are originally in Euclidean space. Like in most existing ANN-H work, we binarize each of them into a dataset in a d-dimensional Hamming space (*i.e.*, $\{0,1\}^d$) whose d is a multiple of 64 that is closest to the dimension of the original dataset, following a standard method used in [57]. We then remove all duplicate points resulting from the binarization process. Finally, we sample, uniformly at random without replacement, a certain number of points from a binarized dataset as the query workload. For each of the 8 binarized datasets, Table 1 shows the number of points in it (excluding those in the query workload), its dimension, its "type", and the query time t_{BF} (BF stands for "Brute Force") that a brute-force linear scan takes on it. For small (first 3), medium (next 3), and large datasets (last 2), the number of queries (i.e., size of the query workload) is set to 200, 1,000, and 10,000, respectively. The binarized Enron dataset was converted from the feature vectors of 1,369 dimensions extracted by the authors of [68]. We drop the word "binarized" in the sequel with the understanding that all datasets we refer to by names have been binarized.

Table 1: Datasets and t_{BF} (in milliseconds).

	dataset	n	d	type	t_{BF}
small	Audio [71]	54,187	192	audio	0.6
	Mnist [72]	69,000	768	image	2.3
	Enron [5]	94,851	1,344	text	6.3
medium	GIST1M [4]	982,661	960	image	39.4
	SIFT1M [4]	993,244	128	image	6.4
	GloVe [61]	1,192,505	128	text	7.3
large	GIST80M [28]	72,908,143	384	image	1,083.8
	SIFT1B [4]	995,534,710	128	image	5,375.1

7.3.3 In-Memory Experimental Results

In this section, we present the in-memory experimental results of iDEC, SRS, OPQ, and HNSW on the 8 datasets shown in Table 1. QALSH and C2LSH are not compared with, because as explained in §7.3.1, they are implemented and optimized only for external-memory operations.

Index Size. Table 2 shows index sizes of four compared algorithms on all datasets (here and in Table 3, CR means crash due to lack of memory during index construction). As shown in Table 2, the index sizes of iDEC are at least 1.3 and 2.7 times smaller than those of SRS and HNSW respectively, but are roughly twice those of OPQ. However, the index size does not tell the whole story, especially in the cases of HNSW and OPQ, since their memory footprints are much larger than their respective index sizes in the index construction process. For example, our measurements show that the peak memory usages[1] of HNSW and OPQ during index construction are one and two orders of magnitude larger than their index sizes, respectively.

Table 2: Comparisons of index sizes (in megabytes).

dataset	iDEC	SRS	OPQ	HNSW
Audio	1.6	2.2	0.6	4.4
Mnist	2.0	2.8	0.8	5.6
Enron	2.8	3.8	1.1	7.7
GIST1M	28.0	39.8	11.2	277.6
SIFT1M	28.5	40.2	11.4	80.7
GloVe	34.0	48.2	13.6	96.9
GIST80M	2,073.2	2,868.7	**CR**	**CR**
SIFT1B	19,251.8	**CR**	**CR**	**CR**

Indexing Time. Table 3 suggests that the indexing times of iDEC are at least 5.4 times shorter than those of SRS and are roughly 3 orders of magnitude shorter than those of both OPQ and HNSW.

Table 3: Comparisons of indexing times (in seconds).

dataset	iDEC	SRS	OPQ	HNSW
Audio	0.03	0.2	15.1	11.5
Mnist	0.06	0.9	99.5	36.2
Enron	0.1	2.3	401.5	93.5
GIST1M	0.8	16.8	1,800.9	2,955.8
SIFT1M	0.4	3.1	509.6	526.8
GloVe	0.5	3.8	365.0	1,444.2
GIST80M	111.4	601.0	**CR**	**CR**
SIFT1B	2,210.4	**CR**	**CR**	**CR**

Query Time (Efficiency) vs. Approximation Ratio (Quality). Figure 2 compares the query performances, in terms of this efficiency-quality tradeoff using approximation ratio as the query quality metric, of the 4 algorithms on the 8 datasets; the 5 missing plots in the last two subfigures correspond to the 5 CR (crash) cases shown in Table 2 and Table 3. Figure 2 shows that, on all the 7 datasets compared on, iDEC significantly outperforms SRS except for a few approximation ratio values that are close to 1 (or those close to 2 on GIST80M). It also shows that the two non-LSH-based algorithms, OPQ and HNSW, outperform iDEC significantly on all the 6 datasets (except Enron) compared on. However, due to their three orders of magnitude longer indexing time (as just shown), it is virtually impossible for OPQ and HNSW to in practice outperform iDEC on very large datasets for the following reason. The parameter settings for OPQ and HNSW to attain an optimal efficiency-quality tradeoff point (i.e., a point on the optimal tradeoff curve) depend heavily on the characteristics of the dataset. Hence, given a dataset $\mathcal{D}$, to identify the parameter settings (of OPQ or HNSW) that lead to a near-optimal tradeoff point, with a quality level suitable for the target application, require many indices (of $\mathcal{D}$) to be built during the parameter tuning process. This would be extremely time-consuming on very large datasets, as it would take millions of seconds to build just a single index in memory (assuming there are terabytes of memory available) for a billion-point dataset such as SIFT1B. Furthermore, the parameters may need to be re-tuned whenever the datasets and/or the application requirements change.

Query Time (Efficiency) vs. MAP (Quality). Figure 3 also compares the query performances, in terms of this efficiency-quality tradeoff using MAP as the query quality metric instead, of the 4 algorithms on the three small and the three medium datasets. In the interest of space, those on the two large datasets are not presented here, since they lead to similar conclusions. Comparing Figure 3 with the six sub-figures (from (a) to (f)) in Figure 2 concerning the three small and the three medium datasets, we can see that the relative orders for the performances of these four algorithms using MAP as the query quality metric are roughly consistent with those using approximation ratio. Furthermore, MAP may not be a better query quality metric than approximation ratio in our experimental settings, since multiple points that have different approximation ratios in Figure 2 can share the same MAP value in Figure 3. In other words, MAP appears to have a weaker discriminative power than approximation ratio in our experimental settings.

7.3.4 *External-Memory Experimental Results*

In this section, we present the results of iDEC, SRS, QALSH and C2LSH for external-memory experiments. OPQ and HNSW are not included here, because as explained in §7.3.1, they are implemented and optimized only for in-memory operations.

Index Size. Table 4 compares the index sizes of iDEC with those of SRS, QALSH, and C2LSH. For QALSH and C2LSH, we were not able to obtain their exact index sizes on SIFT1B, since QALSH had a segmentation fault during the index construction and C2LSH did not finish the index construction in a reasonable amount of time. Instead, we calculate the exact numbers of hash tables required by them using the formulae provided in [40] and [32] respectively, which allow us to conservatively (i.e., more than fair to them) estimate their index sizes (the two numbers with asterisks) in Table 4 for SIFT1B based on their actual index sizes for the 7 smaller datasets.

Table 4 shows that iDEC outperforms SRS, and significantly outperforms QALSH and C2LSH, in index size on the 8 datasets. More precisely, the index sizes of iDEC are roughly 2, and at least 12 and 68 times smaller than those of SRS, QALSH, and C2LSH, respectively. The advantage of iDEC over QALSH and C2LSH comes mainly from that iDEC uses a single 6-dimensional R-tree, while QALSH uses between 63 to 125 B^+ trees, and C2LSH needs between 383 to 651 B^+ trees on the 8 datasets. The advantage of iDEC over SRS may appear surprising, since SRS also uses a single 6-dimensional R-tree. However, in iDEC each coordinate of

[1]The peak memory usages were measured using Linux `getrusage` [1] and MATLAB Profiler [2] for HNSW and OPQ, respectively.

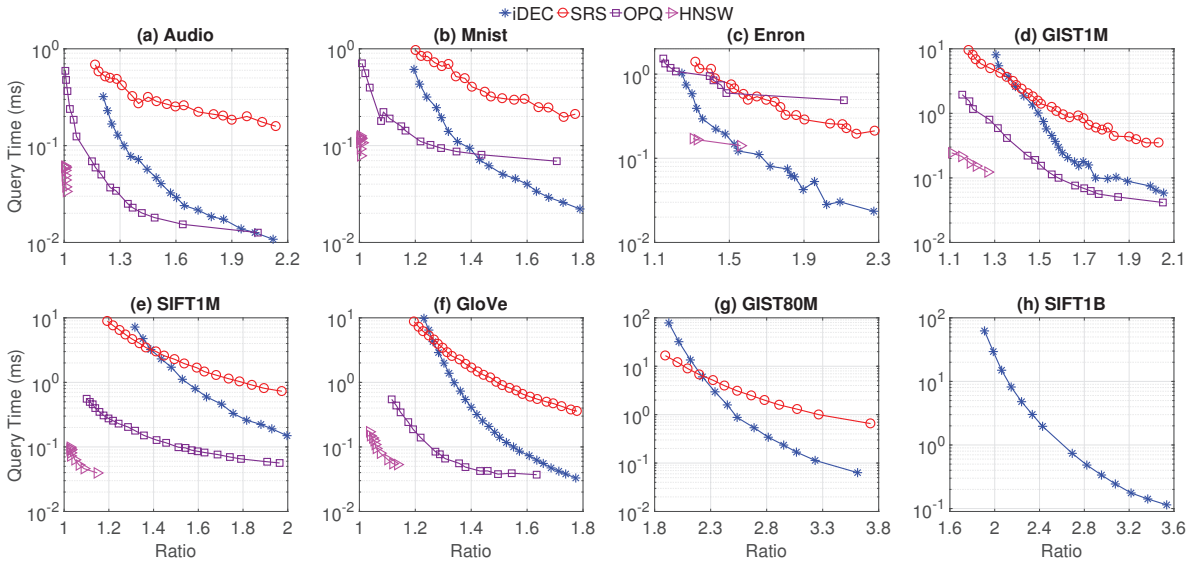

Figure 2: Query time vs. ratio.

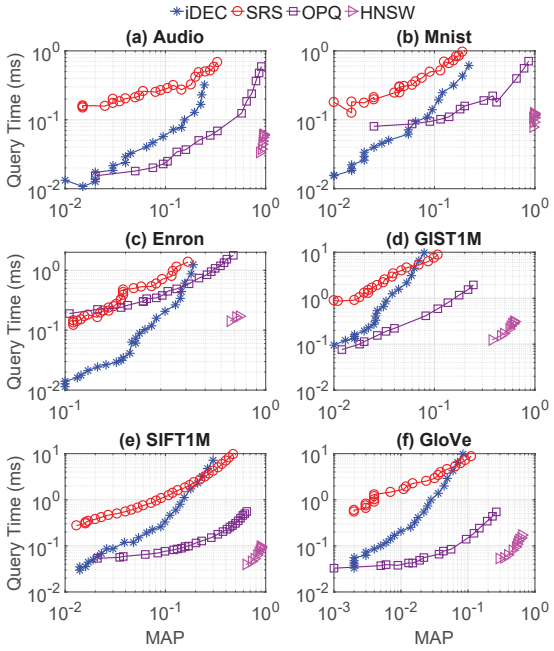

Figure 3: Query time vs. MAP.

the rectangles in the R-tree is only 16 bits long, since each $\psi_i(\cdot)$ maps a d-dimensional data point in $\mathcal{D}$ to an integer between $-d$ to d in value, whereas in SRS each coordinate is 32 bits long since it is a floating-point number.

We are not able to compare their indexing times fairly due to the difference in their implementations. During the index construction, QALSH and C2LSH store all raw data in main memory to maximize the efficiency, whereas iDEC and SRS store only a tiny fraction of them in main memory to minimize main memory usage. Despite this significant disadvantage, the indexing times of iDEC and SRS are still comparable to those of QALSH and C2LSH thanks to their much smaller index sizes. For example, on SIFT1M, it takes iDEC, SRS, QALSH, and C2LSH 163.2 s, 216.3 s, 48.9 s, and 225.9 s to build their indices respectively.

Table 4: Comparisons of index sizes (in megabytes).

dataset	iDEC	SRS	QALSH	C2LSH
Audio	1.1	2.6	14.3	79.3
Mnist	1.5	3.3	18.3	103.0
Enron	2.1	4.5	25.8	145.3
GIST1M	20.6	46.1	314.8	1,751.9
SIFT1M	20.5	47.0	318.1	1,769.7
GloVe	24.3	56.4	386.5	2,147.5
GIST80M	1,518.4	2,724.0	30,836.1	162,170.1
SIFT1B	23,212.4	37,646.5	480,167.1*	2,472,279.3*

I/O Cost vs. Approximation Ratio. Now we compare the query performance of iDEC with those of SRS, QALSH and C2LSH. Like in most literature on external-memory ANN search, we mainly use the I/O costs as the query efficiency metric. We keep the comparisons with QALSH and C2LSH separate from those with SRS, since QALSH and C2LSH require slightly different experimental settings. In the interest of space, we only present the comparison results with QALSH and C2LSH, since those with SRS are similar to the corresponding in-memory comparison results.

Table 5 compares the I/O costs of iDEC with those of QALSH on the left side and those of C2LSH on the right side, on all the 8 datasets except SIFT1B, on which index constructions for QALSH and C2LSH cannot be successfully completed as explained earlier. In each comparison, we tune the value of t in iDEC to match the approximation ratio achieved by the corresponding benchmark algorithm (QALSH or C2LSH). Table 5 shows that the I/O costs of iDEC are 1.1 to 32.0 and 1.8 to 52.4 times smaller than those of QALSH and C2LSH, respectively.

Due to their extremely long query times on GIST80M, QALSH and C2LSH processed only 3,543 and 1,803 queries respectively, each in around a week. We use the average

over these 3,543 and 1,803 queries for QALSH and C2LSH, respectively. For fair comparisons, the two iDEC results on GIST80M reported in Table 5 are also averaged over the same 3,543 and 1,803 queries respectively.

Table 5: Comparisons of I/O costs of iDEC with those of QALSH and C2LSH for achieving the same query quality (approximation ratio).

dataset	ratio	QALSH	iDEC	ratio	C2LSH	iDEC
Audio	1.08	1,804	151	1.03	4,052	289
Mnist	1.10	2,512	505	1.04	5,715	1,098
Enron	1.09	2,646	2,341	1.02	6,428	3,537
GIST1M	1.09	40,106	9,127	1.04	87,728	15,930
SIFT1M	1.07	33,408	3,347	1.01	73,127	4,439
GloVe	1.08	52,345	1,636	1.07	111,058	2,120
GIST80M	1.14	3,253,014	143,008	1.04	7,264,200	332,301

Although higher I/O cost generally predicts longer query time, since most external-memory ANN algorithms are I/O-intensive, we have measured their query times just to confirm that. In the interest of space, we summarize the results as follows. We have found that the query time results (for iDEC and SRS) are roughly consistent with the I/O cost results except for a few approximation ratio values on some datasets (*e.g.,* GloVe) where the query time of iDEC is slightly longer. We have also found that iDEC outperforms QALSH and C2LSH on all medium and large datasets. Furthermore, the larger the dataset is, the higher the advantage iDEC has over QALSH and C2LSH. More specifically, the query times of iDEC are at least 1.2 and 1.4 times shorter than those of QALSH and C2LSH respectively on the three medium datasets, but 12.8 and 25.2 times shorter on the much larger GIST80M dataset, respectively. This asymptotic behavior (that query time advantage grows with dataset size n) is expected for the following reason. Query time is roughly equal to I/O time and CPU time adding up. On one hand, it is clear from Table 5 that iDEC should use much less I/O time than QALSH and C2LSH. On the other hand, the CPU time complexity of iDEC is $O(n)$ as described in §4.5, whereas those of QALSH and C2LSH are both $O(n \log n)$ [32, 40], so iDEC's advantage on CPU time grows with n.

7.4 Performance Evaluation for ANN-E

7.4.1 Benchmark Algorithms and Datasets

We compare the performances of iDEC, including multiset-iDEC and context-iDEC, with that of CGK-LSH [76], the state-of-the-art solution to ANN-E. For both multiset-iDEC and context-iDEC, we do not use raw multiset "features" for indexing, since their dimensions are too high for the k-d tree. Instead we reduce their dimensions by applying $\xi_i(\cdot)$'s, the ToW-for-L_1 sketching functions, to them (described in §6).

CGK-LSH is a combination of the CGK-embedding [18], which maps strings measured in edit space to longer strings measured in Hamming space via padding, and the bit-sampling LSH [41]. As shown in [18], the distortion of the CGK-embedding is quite large especially when the edit distance between two input strings is large, so multiple CGK-embeddings are used in CGK-LSH to reduce the distortion.

We use two string datasets: GEN50KS [75] and UNIREF [75]. Table 6 summarizes the number of strings (excluding those in the query workload), the alphabet size $|\Sigma|$, the minimum, average and maximum lengths of the strings, the type (DNA or protein sequences), and the average (over 1,000 queries) query time t_{BF} that a brute-force linear scan takes on the 2 datasets.

Table 6: Datasets and t_{BF} (in milliseconds).

dataset	n	$\|\Sigma\|$	length			type	t_{BF}
			Min	Avg	Max		
GEN50KS	49K	4	4,844	5,000	5,109	DNA	77,593.6
UNIREF	399K	25	200	445	35,213	protein	9,024.2

7.4.2 Experimental Results

Table 7 and Table 8 show the experimental results, each of which is averaged over 1,000 queries, of the CGK-LSH, multiset-iDEC, and context-iDEC, on GEN50KS and UNIREF. We have explored various parameter settings for CGK-LSH, multiset-iDEC, and context-iDEC. For each dataset, the table presents the index size, the indexing time, and the query time for achieving each of the two target recall values. The two target recall values for iDEC are reached by tuning only the value of t. Hence both the index size and the indexing time remain the same for both multiset-iDEC and context-iDEC, for both recall values.

For the CGK-LSH, there are three parameters to tune: the number of CGK embeddings, the number of hash tables for the embedded strings under each CGK embedding, and the number of sampling bits. The parameter values for achieving the two target recall values are also shown in both tables, each as a 3-tuple after "CGK-LSH". For the multiset-iDEC and the context-iDEC, we set m to 10 and 12 for the results shown in Table 7 and Table 8 respectively. The window size w for the left/right context is 12 in context-iDEC, whereas the gram sizes are different for different datasets. For the results in Table 7, multiset-iDEC uses grams of size $q = 5$, and context-iDEC uses $q = 4$. The q values for the results in Table 8 become 1 and 2 for multiset-iDEC and context-iDEC respectively.

The results show that iDEC is significantly more space- and time-efficient than CGK-LSH. The index sizes using CGK-LSH are much larger (by at least 13 times) than those using iDEC. The query time of iDEC can be up to 3 orders of magnitude shorter than that of CGK-LSH.

The results in Table 7 show that the context-iDEC has shorter query times than the multiset-iDEC on GEN50KS, whereas the results in Table 8 show that they have similar query times on UNIREF. Our interpretation of this observation is as follows. The dataset GEN50KS has a much smaller alphabet but much larger average string length. Hence, block-wise permutations could appear more frequently in GEN50KS than in UNIREF. As we explained in § 6.2, context-iDEC is more robust against such spurious patterns than multiset-iDEC.

The results in the two tables also show that iDEC has a much larger advantage (over CGK-LSH) on GEN50KS (the smaller dataset) than on UNIREF (the larger dataset), which is unusual. The reason for this "anomaly" is that, the distribution of distances between strings in GEN50KS is much more friendly to our iDEC framework than that in

Table 7: Index sizes (in megabytes), indexing times (in seconds) and query times (in milliseconds) on GEN50KS.

recall	algorithm	size	indexing time	query time
0.91	CGK-LSH $(7, 10, 13)$	41.9	11.9	6,618.0
	multiset-iDEC	2.1	11.6	11.9
	context-iDEC	2.0	92.5	5.0
0.95	CGK-LSH $(9, 10, 13)$	53.9	15.3	6,652.1
	multiset-iDEC	2.1	11.6	26.8
	context-iDEC	2.0	92.5	7.0

Table 8: Index sizes (in megabytes), indexing times (in seconds) and query times (in milliseconds) on UNIREF.

recall	algorithm	size	indexing time	query time
0.90	CGK-LSH $(5, 10, 13)$	230.2	8.5	196.1
	multiset-iDEC	17.1	4.5	65.9
	context-iDEC	16.9	242.8	66.7
0.93	CGK-LSH $(7, 10, 13)$	322.3	12.4	355.1
	multiset-iDEC	17.1	4.5	147.9
	context-iDEC	16.9	242.8	146.4

UNIREF: As shown in [76], given a (random) query string α, most of the strings in GEN50KS, except a few nearest neighbors of α, are farther away from α, typically by 10 times or more (according to our measurements), than the nearest neighbor of α, whereas this ratio is much smaller in UNIREF. Consequently, the number of NNs t (see Line 1 of Algorithm 1) searched is much smaller on GEN50KS than on UNIREF to achieve the same quality (recall) in iDEC, resulting in shorter query times. CGK-LSH does not benefit from the more friendly distance distribution in GEN50KS, because the numbers of candidates (strings hashed into the same buckets as the queries) searched by CGK-LSH remain comparable on both datasets, thanks to the aforementioned large distortion of CGK-embedding, and each candidate string takes a longer time to check out in GEN50KS than in UNIREF since the former is on average much longer than the latter.

8. RELATED WORK

In this section, we provide a brief survey of ANN algorithms, besides those we have already described earlier, focusing on those directly related to our work. More related work on ANN search can be found in the following three recent ANN benchmark papers [10, 25, 46].

LSH-Based Algorithms. LSH was first introduced for use in the Hamming space in [41]. A classical LSH-based algorithm needs to use a large number, typically in hundreds to thousands, of hash tables [16], which results in a large index size. To overcome this issue, many techniques, such as LSH forest (LSHF) [12], entropy-based LSH [60] and multi-probe LSH [50], have been proposed to reduce the number of hash tables by allowing multiple buckets to be examined in each table. However, doing so is found (*e.g.*, in [64]) to result in longer query time than the classical LSH-based algorithms sometimes.

The LSH-based techniques described above are designed for in-memory operations. Many recent works have adapted LSH for external-memory operations. Examples of them include LSB-forest [69], SK-LSH [49] (a variant of LSB-forest), C2LSH [32], QALSH [40], and I-LSH [48] (a variant of QALSH). These works usually use disk-resident data structures, such as B^+ trees [22, 26], to store each hash table. They all have a large index size. In addition, to achieve a relatively high level of query quality, most of them require a large number of disk I/Os that result in long query time. SRS [68] tackles these issues using a single low-dimensional (say 6-dimensional) R-tree so that the index size and the number of I/Os are reduced significantly.

Graph-Based Algorithms. Their basic idea is for each point α in the database to maintain a list of other points in the database that are close to α (*i.e.,* neighbors of α), and the query processing is performed via searching this neighbor-relationship graph. To achieve the same level of query quality (say recall), graph-based algorithms, such as HNSW [51], NSG [31], FANNG [36], SSG [30] and DiskANN [42], are in general much faster than LSH-based solutions. However, in general their index construction times are much longer.

Quantization-Based Algorithms. Quantization [34] is another commonly used technique. Its basic idea is to quantize the data in $\mathcal{D}$ to a small (relative to n, the number of points in $\mathcal{D}$) set of designated points and the search is performed over those points in $\mathcal{D}$ that are quantized to the same designated point as the query point, and to the nearby designated points. The state-of-the-art quantization-based algorithms include OPQ [33], Inverted Multi-Index (IMI) [11], and Faiss [43] (a GPU-based variant of OPQ).

Other ANN Algorithms. Since there are many of them, we describe only a few representatives here. Tree-based data structures have been widely used for both exact NN and ANN. Representative tree-based ANN algorithms include FLANN [54] and Annoy [3]. A recent work for external-memory queries, called HD-index [9], is based on space-filling curves [65]. Its basic idea is to map each point β in $\mathcal{D}$ to a so-called ordering value that corresponds roughly to the along-the-curve distance between β and the origin of the space-filling curve, so that points close in $\mathcal{D}$ are mapped to similar ordering values. The (one-dimensional) ordering values of all points in $\mathcal{D}$ are then indexed using a reference distance B^+ tree. Multiple trees are generally required to achieve a good efficiency-quality tradeoff. As shown in [9], the index size of HD-index is around 2 to 3 times larger than that of SRS (for external-memory). In comparison, the index size of iDEC is 2 times smaller than that of SRS.

9. CONCLUSION

In this paper, we propose *indexable distance estimating codes (iDEC)*, a new solution framework to the approximate nearest neighbor (ANN) search problem that fundamentally extends and improves the LSH framework. We also discover subtle deep connections between EEC, LSH, and iDEC. We show that our iDEC-based solutions for ANN-H and ANN-E outperform the respective state-of-the-art LSH-based solutions in both index sizes and query time complexities. We also show that our iDEC-based in-memory ANN-H solution significantly outperforms the respective state-of-the-art solutions in peak memory usage and indexing time, which allows it to easily scale to massive datasets.

Acknowledgment. This work is supported in part by US NSF through award CNS-1909048 and by a research grant from Keysight Technologies, Inc.

10. REFERENCES

[1] http://man7.org/linux/man-pages/man2/getrusage.2.html.

[2] https://www.mathworks.com/help/matlab/ref/profile.html.

[3] Annoy: Approximate nearest neighbors in C++/Python optimized for memory usage and loading/saving to disk. https://github.com/spotify/annoy.

[4] Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[5] Enron email dataset. http://www.cs.cmu.edu/~enron/.

[6] D. Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, June 2003.

[7] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, Feb. 1999.

[8] A. Andoni, P. Indyk, and I. P. Razenshteyn. Approximate nearest neighbor search in high dimensions. *CoRR*, abs/1806.09823, 2018.

[9] A. Arora, S. Sinha, P. Kumar, and A. Bhattacharya. HD-Index: Pushing the scalability-accuracy boundary for approximate KNN search in high-dimensional spaces. *PVLDB*, 11(8):906–919, 2018.

[10] M. Aumüller, E. Bernhardsson, and A. Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *Similarity Search and Applications*, pages 34–49, Cham, 2017.

[11] A. Babenko and V. Lempitsky. The inverted multi-index. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3069–3076, June 2012.

[12] M. Bawa, T. Condie, and P. Ganesan. LSH Forest: Self-tuning indexes for similarity search. In *Proceedings of the International Conference on World Wide Web*, page 651–660, 2005.

[13] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Sept. 1975.

[14] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the International Conference on Machine Learning*, pages 97–104, 2006.

[15] J. L. Blanco and P. K. Rai. nanoflann: a C++11 header-only library for nearest neighbor (NN) search with KD-trees. https://github.com/jlblancoc/nanoflann, 2014.

[16] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[17] R. Cai, C. Zhang, L. Zhang, and W.-Y. Ma. Scalable music recommendation by search. In *Proceedings of the ACM International Conference on Multimedia*, pages 1065–1074, 2007.

[18] D. Chakraborty, E. Goldenberg, and M. Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 712–725, 2016.

[19] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 380–388, 2002.

[20] B. Chen, Z. Zhou, Y. Zhao, and H. Yu. Efficient error estimating coding: Feasibility and applications. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 3–14, New Delhi, India, Aug. 2010.

[21] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the Annual Symposium on Computational Geometry*, pages 160–164, 1994.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[23] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Annual Symposium on Computational Geometry*, pages 253–262, 2004.

[24] DBWangGroupUNSW. SRS - fast approximate nearest neighbor search in high dimensional euclidean space with a tiny index. https://github.com/DBWangGroupUNSW/SRS.

[25] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the Lernaean Hydra: experimental evaluation of data series approximate similarity search. *PVLDB*, 13(3):403–420, 2020.

[26] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010.

[27] J. Feigenbaum, S. Kannan, M. J. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams. *SIAM Journal on Computing*, 32(1):131–151, Jan. 2003.

[28] R. Fergus, A. Torralba, and W. T. Freeman. Tiny images dataset. http://horatio.cs.nyu.edu/mit/tiny/data/index.html.

[29] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, Sept. 1977.

[30] C. Fu, C. Wang, and D. Cai. Satellite system graph: Towards the efficiency up-boundary of graph-based approximate nearest neighbor search. *CoRR*, abs/1907.06146, 2019.

[31] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *PVLDB*, 12(5):461–474, 2019.

[32] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 541–552, Scottsdale, Arizona, USA, May 2012. Source code: https://github.com/fengjl18/C2LSH-Code.

[33] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer*

Vision and Pattern Recognition, pages 2946–2953, June 2013.

[34] R. M. Gray and D. L. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383, Oct 1998.

[35] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, Massachusetts, June 1984.

[36] B. Harwood and T. Drummond. FANNG: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June 2016.

[37] N. Hua, A. Lall, B. Li, and J. Xu. A simpler and better design of error estimating coding. In *Proceedings of the IEEE International Conference on Computer Communications*, pages 235–243, Orlando, FL, USA, Mar. 2012.

[38] N. Hua, A. Lall, B. Li, and J. Xu. Towards optimal error-estimating codes through the lens of fisher information analysis. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 125–136, London, England, UK, June 2012.

[39] J. Huang, S. Yang, A. Lall, J. Romberg, J. Xu, and C. Lin. Error estimating codes for insertion and deletion channels. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems*, pages 381–393, Austin, Texas, USA, June 2014.

[40] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *PVLDB*, 9(1):1–12, 2015. Source code: https://github.com/DBWangGroupUNSW/nns_benchmark/tree/master/algorithms/QALSH.

[41] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 604–613, Dallas, Texas, USA, May 1998.

[42] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi. DiskANN: Fast accurate billion-point nearest neighbor search on a single node. In *Advances in Neural Information Processing Systems 32*, pages 13771–13781. Curran Associates, Inc., 2019.

[43] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, pages 1–1, 2019.

[44] W. Kong, W.-J. Li, and M. Guo. Manhattan hashing for large-scale image retrieval. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 45–54, Portland, Oregon, USA, Aug. 2012.

[45] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[46] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2019.

[47] K. Lin, H. Yang, J. Hsiao, and C. Chen. Deep learning of binary hash codes for fast image retrieval. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 27–35, Boston, MA, USA, June 2015.

[48] W. Liu, H. Wang, Y. Zhang, W. Wang, and L. Qin. I-LSH: I/O efficient c-approximate nearest neighbor search in high-dimensional space. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1670–1673, April 2019.

[49] Y. Liu, J. Cui, Z. Huang, H. Li, and H. T. Shen. SK-LSH: An efficient index structure for approximate nearest neighbor search. *PVLDB*, 7(9):745–756, 2014.

[50] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In *Proceedings of the International Conference on Very Large Data Bases*, pages 950–961, Vienna, Austria, Sept. 2007.

[51] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020. Source code: https://github.com/nmslib/hnswlib.

[52] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the International Conference on World Wide Web*, pages 141–150, 2007.

[53] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Springer Publishing Company, Incorporated, 2005.

[54] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, Nov 2014.

[55] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[56] T. T. Nguyen, P.-M. Hui, F. M. Harper, L. Terveen, and J. A. Konstan. Exploring the filter bubble: The effect of using recommender systems on content diversity. In *Proceedings of the International Conference on World Wide Web*, pages 677–686, Seoul, Korea, Apr. 2014.

[57] 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*, pages 3108–3115, Providence, RI, USA, June 2012.

[58] R. O'Donnell, Y. Wu, and Y. Zhou. Optimal lower bounds for locality-sensitive hashing (except when q is tiny). *ACM Transactions on Computation Theory*, 6(1):5:1–5:13, Mar. 2014.

[59] S. M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989.

[60] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithm*, page 1186–1195. Society for Industrial and Applied Mathematics, 2006.

[61] J. Pennington, R. Socher, and C. D. Manning. GloVe: Global vectors for word representation. https://nlp.stanford.edu/projects/glove/.

[62] L. Qi, X. Zhang, W. Dou, and Q. Ni. A distributed locality-sensitive hashing-based approach for cloud service recommendation from multi-source data. *IEEE Journal on Selected Areas in Communications*, 35(11):2616–2624, Nov 2017.

[63] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[64] K. Rong, C. E. Yoon, K. J. Bergen, H. Elezabi, P. Bailis, P. Levis, and G. C. Beroza. Locality-sensitive hashing for earthquake detection: A case study of scaling data-driven science. *PVLDB*, 11(11):1674–1687, 2018.

[65] H. Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[66] S. Sood and D. Loguinov. Probabilistic near-duplicate detection using simhash. In *Proceedings of the ACM International Conference on Information and Knowledge Management*, pages 1117–1126, Glasgow, Scotland, UK, Oct. 2011.

[67] M. Šošić and M. Šikić. Edlib: A C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

[68] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: Solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *PVLDB*, 8(1):1–12, 2014.

[69] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems*, 35(3), July 2010.

[70] S. S. Vempala. *The Random Projection Method*, volume 65. American Mathematical Soc., 2005.

[71] Z. Wang, W. Dong, W. Josephson, Q. Lv, M. Charikar, and K. Li. Sizing sketches: A rank-based analysis for similarity search. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 157–168. Association for Computing Machinery, 2007. Dataset is available at http://www.cs.princeton.edu/cass/audio.tar.gz.

[72] L. Yann, C. Corinna, and J. B. Christopher. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[73] A. C.-C. Yao. Some complexity questions related to distributive computing (preliminary report). pages 209–213, Atlanta, Georgia, USA, Apr. 1979.

[74] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 311–321, USA, 1993. Society for Industrial and Applied Mathematics.

[75] H. Zhang. String datasets. https://iu.box.com/s/x7hg7uxj7xmmcdvc62k7iux9txtt9doi.

[76] H. Zhang and Q. Zhang. EmbedJoin: Efficient edit similarity joins via embeddings. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 585–594, Halifax, NS, Canada, Aug. 2017.

[77] Z. Zhang and P. Kumar. mEEC: A novel error estimation code with multi-dimensional feature. In *Proceedings of the IEEE International Conference on Computer Communications*, pages 1–9, Atlanta, GA, USA, May 2017.