MP-RW-LSH: An Efficient Multi-Probe LSH Solution to ANNS-L_1

Huayi Wang[†], Jingfan Meng[†] , Long Gong[††] , Jun Xu[†] , Mitsunori Ogihara[‡]
[†]Georgia Institute of Technology, USA [††]Facebook, USA [‡]University of Miami, USA
[†]{huayiwang,jmeng40}@gatech.edu, jx@cc.gatech.edu, [††]lgong30@fb.com, [‡]ogihara@cs.miami.edu

ABSTRACT

Approximate Nearest Neighbor Search (ANNS) is a fundamental algorithmic problem, with numerous applications in many areas of computer science. Locality-Sensitive Hashing (LSH) is one of the most popular solution approaches for ANNS. A common shortcoming of many LSH schemes is that since they probe only a single bucket in a hash table, they need to use a large number of hash tables to achieve a high query accuracy. For ANNS-L_2, a multi-probe scheme was proposed to overcome this drawback by strategically probing multiple buckets in a hash table. In this work, we propose MP-RW-LSH, the first and so far only multi-probe LSH solution to ANNS in L_1 distance, and show that it achieves a better tradeoff between scalability and query efficiency than all existing LSH-based solutions. We also explain why a state-of-the-art ANNS-L_1 solution called Cauchy projection LSH (CP-LSH) is fundamentally not suitable for multi-probe extension. Finally, as a use case, we construct, using MP-RW-LSH as the underlying "ANNS-L_1 engine", a new ANNS-E (E for edit distance) solution that beats the state of the art.

PVLDB Reference Format:
Huayi Wang, Jingfan Meng, Long Gong, Jun Xu, Mitsunori Ogihara.
MP-RW-LSH: An Efficient Multi-Probe LSH Solution to ANNS-L_1. PVLDB, 14(13): 3267-3280, 2021.
doi:10.14778/3484224.3484226

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Why1221/MP-RW-LSH.

1 INTRODUCTION

Approximate Nearest Neighbor Search (ANNS) is a fundamental algorithmic problem, with numerous applications in many areas of computer science, including informational retrieval [35], recommendations [40], near-duplication detections [47], etc. In ANNS, given a query point (vector) $\vec{q}$, we search in a massive dataset $\mathcal{D}$, that lies in a high-dimensional space, for one or more points in $\mathcal{D}$ that are *among the closest* to $\vec{q}$ according to some distance metric. Throughout this paper, we accent every vector with a right arrow, like in the case of $\vec{q}$.

The ANNS literature is mostly focused on ANNS in the Euclidean (L_2) distance, or ANNS-L_2 for short. In this work, we focus instead on ANNS-L_1, ANNS in the Manhattan (L_1) distance, which is much less thoroughly studied in comparison. ANNS-L_1 is nonetheless an extremely important problem for four reasons. First, it arises in almost all application domains of ANNS-L_2. Second, set and multiset similarity search and join [45], an increasingly important family of ANNS problems that arise in database application domains such as data cleaning [14], social network data mining [48], finding similar sequences [58], and information retrieval [52], can be reduced to ANNS-L_1 [23]; any breakthrough on the latter can lead to much better solutions to the former. Third, ANNS in the edit distance (ANNS-E), an important yet extremely challenging problem [58], can be reduced to and solved as an ANNS-L_1 problem [38]. Finally, the same can be said about ANNS in earth mover's distance (EMD). ANNS-EMD is an extremely important problem [54] that arises in application domains such as duplicated document detection [32], image retrieval [44], and detection of similar distributions [54]. So far, no scalable LSH solution exists for ANNS-EMD. A scalable LSH solution for ANNS-L_1 can lead to one for ANNS-EMD for the following reason. The L_1 distance is known to be the "closest relative" to the EMD: In one-dimension, the EMD between two probability density functions (pdf's) is equal to the L_1 distance between the corresponding cumulative density functions (cdf's), and in two or higher dimensions, the best (in terms of distortion factor) metric embedding result on the former is to the L_1 distance [9, 28].

Proceedings of the VLDB Endowment, Vol. 14, No. 13 ISSN 2150-8097.
doi:10.14778/3484224.3484226

1.1 LSH and Multi-Probe LSH

One of the most popular ANNS solution approaches is Locality-Sensitive Hashing (LSH) [27]. The key intellectual component of an LSH scheme is its hash function family $\mathcal{H}$. Any function h sampled uniformly at random from $\mathcal{H}$ has the following nice collision property: It maps two distinct points in $\mathcal{D}$ to the same hash value with probability p_1 if they are close to each other (say no more than distance r_1 apart) and with probability $p_2 < p_1$ if they are far apart (say more than $r_2 > r_1$ apart), respectively. Such an LSH scheme can achieve a query time complexity of roughly $O(n^\rho)$, where $\rho \triangleq \log p_1 / \log p_2$ is called the *quality* of the LSH family, and n is the number of points in $\mathcal{D}$. However, an LSH scheme requires the maintenance and search of a large number ($O(n^\rho)$ in theory [27] and tens to hundreds in practice [13]) of hash tables, for the reason explained next.

Whenever possible, in the rest of this paper, we focus on only one of these hash tables and explain how it is probed for the nearest neighbors of $\vec{q}$. In this hash table, an LSH scheme probes only a single bucket that has the highest success probability (of containing a nearest neighbor of $\vec{q}$): $\overrightarrow{h(q)}$, the bucket that $\vec{q}$ is hashed to by an LSH function vector $\overrightarrow{h(\cdot)} \triangleq \langle h_1(\cdot), \cdots, h_M(\cdot)\rangle$. We refer to $\overrightarrow{h(q)}$ as the *epicenter bucket* in the sequel. Unfortunately, the success probability of the epicenter bucket is still quite low for the following reason. In order for a nearest neighbor $\vec{s}$ to be (successfully) found in the epicenter bucket, each of its M hash values $h_i(\vec{s})$, $i = 1, 2, \cdots, M$, has to agree with the corresponding hash value of $\vec{q}$. Hence this probability decays exponentially with M, and M can be as large as 20 in some LSH schemes. As a result, a large number of hash tables have to be used to boost this probability.

Multi-probe [36] was proposed for boosting this success probability when the Gaussian-projection LSH scheme (GP-LSH) [18] for ANNS-L_2 is used as the baseline LSH. The idea of multi-probe is that, the algorithm probes not only the epicenter bucket, but also $T > 0$ other nearby buckets whose success probabilities are among the $T + 1$ highest. This way, the total success probability can be significantly increased, and the number of hash tables used for reaching a target success probability can be significantly reduced.

Due to its spectacular efficacy, multi-probe GP-LSH (MP-GP-LSH) has since been deployed in various systems including smartphone applications [43], audio content retrieval [56], automatic product suggestions [30], etc. We will explain in § 2.2 that the efficacy of MP-GP-LSH stems entirely from the following property of GP-LSH: The success probability of a bucket decreases roughly at the "Gaussian pdf rate" $O(e^{-c{d_2}^2})$, where d_2 is the bucket's L_2 distance from the epicenter (to be defined in § 2.2), and $c > 0$ is a constant. It appears hard to apply multi-probe to LSH schemes lacking this property. Currently, besides those for ANNS-L_2 [33, 36], multi-probe LSH solutions exist only for ANNS in the Chi-squared distance [24] and in the angular distance [10, 33], and in both cases the success probability has this property.

1.2 Our Multi-Probe LSH Solution

In this work, we propose multi-probe random-walk LSH (MP-RW-LSH), the first and so far only multi-probe LSH solution for ANNS-L_1. Our solution significantly outperforms Cauchy projection LSH (CP-LSH) [18], the state-of-the-art LSH scheme for ANNS-L_1. Our solution however is not a multi-probe extension of CP-LSH. In fact, we discover that CP-LSH is fundamentally not suitable for the multi-probe extension for the following reason: The total success probability of the top-$(T + 1)$ buckets remains quite low even when T is very large thanks to the heavy-tail nature [42] of its underlying Cauchy distribution.

We propose a new LSH scheme for ANNS-L_1 that is much better suited for multi-probe. We call it random-walk LSH (RW-LSH), because any raw hash value function (defined later) f in it has the following property: Given any two nonnegative integer data points $\vec{s}$ and $\vec{t}$, $f(\vec{s}) - f(\vec{t})$, the difference between their raw hash values, has the same probability distribution as that of a d_1-step random walk, where $d_1 = \|\vec{s} - \vec{t}\|_1$ is their L_1 distance. Hence, when d_1 is large, this difference converges to a zero-mean Gaussian distribution with variance d_1. As a result, given a query point $\vec{q}$, the success probability of a bucket decays in the same aforementioned Gaussian pdf manner as in GP-LSH. Hence RW-LSH can be extended to MP-RW-LSH in almost the same way as GP-LSH (to MP-GP-LSH). We will show that MP-RW-LSH strikes a much better tradeoff between scalability and query efficiency than all existing LSH-based ANNS-L_1 solutions.

As a use case of MP-RW-LSH, we will show that by replacing, in the state-of-the-art ANNS-E (edit distance) solution called iDEC [23], the "ANNS-L_1 engine" with MP-RW-LSH, the resulting ANNS-E solution achieves much better query efficiency while increasing the index size only slightly.

To summarize, we have made three major contributions in this work. First, MP-RW-LSH is the first multi-probe LSH solution for ANNS-L_1, and it achieves a much better scalability-efficiency tradeoff than all existing LSH solutions. Second, through a thorough analysis, we explain why CP-LSH, the state-of-the-art LSH solution for ANNS-L_1, is fundamentally unsuitable for multi-probe extension. To the best of our knowledge, this is the first such unsuitability study. Third, using MP-RW-LSH as the underlying "ANNS-L_1 engine", we construct a new ANNS-E solution that beats the state of the art.

2 PRELIMINARIES

2.1 Locality-Sensitive Hashing

In an LSH scheme, typically a vector of $M > 1$ LSH functions $\vec{h} = \langle h_1, h_2, \cdots, h_M \rangle$ are used to map each point $\vec{s}$ in $\mathcal{D}$ to an M-dimensional vector of hash values $\overrightarrow{h(s)} = \langle h_1(\vec{s}), h_2(\vec{s}), \cdots, h_M(\vec{s}) \rangle$. This point $\vec{s}$ is to be stored in a hash bucket indexed by the vector $\overrightarrow{h(s)}$; hence we identify this hash bucket as $\overrightarrow{h(s)}$. Then given a query point $\vec{q}$, the search procedure is to probe all points in the hash bucket $\overrightarrow{h(q)}$ in the hope that some nearest neighbors of $\vec{q}$ are mapped to the same hash vector (bucket).

We now describe what such an LSH function h_i (a scalar in the vector $\vec{h}$ defined above) is in the three aforementioned LSH schemes respectively: GP-LSH, CP-LSH, and RW-LSH. In all three LSH schemes, h_i takes the same following form: $h_i(\vec{s}) = \lfloor (f_i(\vec{s}) + b_i)/W \rfloor$, where $W > 0$ is a constant and b_i is a random variable (fixed after generation) uniformly distributed in $[0, W]$. Here $f_i(\vec{s})$ is called the raw hash value of $\vec{s}$. For an m-dimensional point $\vec{s} = \langle s_1, s_2, \cdots, s_m \rangle$, the raw hash value function $f_i(\cdot)$ takes the same form in GP-LSH and CP-LSH: $f_i(\vec{s}) = \vec{s} \cdot \vec{\eta}$, where "$\cdot$" is the inner product (which is mathematically a *projection*). GP-LSH and CP-LSH differ only in the choice of $\vec{\eta}$. In GP-LSH, $\vec{\eta}$ is an m-dimensional i.i.d. standard Gaussian random vector (fixed after generation), so its f_i is called a Gaussian projection. In CP-LSH, $\vec{\eta}$ is an m-dimensional i.i.d. standard Cauchy random vector, so its f_i is called a Cauchy projection. In RW-LSH, f_i mimics the one-dimensional random walk, which will be described in § 3.1.

Each bucket $\vec{\beta}$ corresponds to an M-dimensional cube with width W in each dimension. Any point $\vec{s}$ whose shifted (by $\vec{b}$) raw hash value vector $\overrightarrow{f(q)} + \vec{b} \triangleq \langle f_1(\vec{s}) + b_1, f_2(\vec{s}) + b_2, \cdots, f_M(\vec{s}) + b_M \rangle$ falls into this cube belongs to $\vec{\beta}$. Given a query point $\vec{q}$, we refer to its shifted raw hash value vector $\overrightarrow{f(q)} + \vec{b}$ as the *epicenter* and its hash bucket $\overrightarrow{h(q)}$ as the *epicenter bucket* in the sequel. We can represent any other bucket, say $\vec{\beta}$ (an M-dimensional vector), by $\vec{\beta} - \overrightarrow{h(q)}$, its offset from $\overrightarrow{h(q)}$. This offset, denoted as $\vec{\delta} = \langle \delta_1, \delta_2, \cdots, \delta_M \rangle$, is called the perturbation vector [36] of the bucket $\vec{\beta}$.

We denote as $\overrightarrow{x^q(1)}$ the vector $\langle x_1^{\vec{q}}(1), x_2^{\vec{q}}(1), \cdots, x_M^{\vec{q}}(1) \rangle$, where $x_i^{\vec{q}}(1) = W - \{(f_i(\vec{q}) + b_i)/W\}\, W$, for $i = 1, 2, \cdots, M$; here $\{y\}$ denotes the fractional part of y. We call $\overrightarrow{x^q(1)}$ the *coordinates vector* of the query point $\vec{q}$ since the length of each $x_i^{\vec{q}}(1)$ is the distance from the epicenter $\overrightarrow{f(q)} + \vec{b}$ to the "right" face of of the epicenter cube (bucket) that is perpendicular to the i^{th} axis (dimension).

To simplify the notation, we drop the superscript $\vec{q}$ from $\overrightarrow{x^q(1)}$ and $x_i^{\vec{q}}(1)$ in the sequel, and simply write them as $\overrightarrow{x(1)}$ and $x_i(1)$ respectively, with the understanding that they depend on $\vec{q}$. Since each b_i is uniformly distributed in $[0, W]$, each x_i is uniformly distributed in $[0, W]$ also. Hence, $\overrightarrow{x(1)}$ is a random vector that is uniformly distributed in the M-dimensional cube $[0, W]^M$.

Let p_1 and p_2 be as defined in § 1.1. Given a query point $\vec{q}$, the number of *spurious points* in $\mathcal{D}$ (say containing n points), is equal to $p_2^M n$ in expectation, where a spurious point is one that is mapped by the M LSH functions to the same vector as, but is not actually close to, the query point $\vec{q}$. Since this number, which contributes to the time cost of probing each bucket, needs to be kept low at $O(1)$, we need $M = \log_{1/p_2} n + O(1)$ LSH functions. However, in this case the probability with which any good point (one that is close to $\vec{q}$) is hashed to the epicenter bucket, is only $p_1^M = O(n^{-\rho})$, where $\rho = \log p_1 / \log p_2$ is the quality of the LSH family as defined above. Hence roughly $O(n^\rho)$ hash tables have to be used to guarantee that any good point has a probability at least $1 - e^{-1}$ to be found in at least one hash table, Therefore, the query time complexity of such an LSH scheme is also $O(n^\rho)$.

2.2 Multi-Probe LSH

In this section, we describe MP-GP-LSH, the original multi-probe LSH scheme for ANNS-L_2 [36] that uses the Gaussian projection LSH (GP-LSH) as its baseline. Again, we fix a query point $\vec{q}$ and one hash table, and focus on probing for a nearest neighbor of $\vec{q}$, which we denote as $\vec{s}$, in this hash table. As explained earlier, the idea of multi-probe is to probe the top-(T+1) buckets, which include the epicenter bucket $\overrightarrow{h(q)} = \langle h_1(\vec{q}), h_2(\vec{q}), \cdots, h_M(\vec{q})\rangle$ and T other buckets that have the highest success probabilities of containing $\vec{s}$.

In a multi-probe LSH scheme, the top-(T+1) buckets need to be first identified and then probed in the decreasing order of their success probabilities; we call this ordered list the *optimal probing sequence* and denote it as $S_T(\overrightarrow{x(1)})$, where $\overrightarrow{x(1)}$ is the aforementioned coordinates vector. This succinct notation is justified since $S_T(\overrightarrow{x(1)})$ is determined by $\overrightarrow{x(1)}$. However, given an arbitrary $\overrightarrow{x(1)}$, to efficiently compute $S_T(\overrightarrow{x(1)})$ is nontrivial.

Three refinements were proposed in [36] for more efficiently computing the optimal or a near-optimal probing sequence. *The first refinement* is to explore the "neighborhood" of $\overrightarrow{h(q)}$, for the *next bucket* (that has the next highest success probability) in the ordered list $S_T(\overrightarrow{x(1)})$ in an "M-dimensional spiral fashion" using a heap data structure. As shown in [36], this "spiral search" algorithm guarantees to find the correct $S_T(\overrightarrow{x(1)})$, by traversing, and computing the success probabilities of, at most $O(T)$ buckets. However, even to compute $O(T)$ such success probabilities can be quite time-consuming because each success probability is the product of M different probability values (like that will be shown in (4)).

The second refinement significantly reduces the computation cost of each success probability. To explain the second refinement and later our MP-RW-LSH solution, we need to introduce some notations. Recall that in the coordinates vector $\overrightarrow{x(1)}$, each scalar $x_i(1)$ is the distance from the epicenter to the "right" face of the epicenter bucket that is perpendicular to the i^{th} axis (dimension). We define each $x_i(-1)$ as $W - x_i(1)$, which is the distance from the epicenter to the corresponding "left" face, and define each $x_i(0)$ as 0. We denote as $\overrightarrow{x(\delta)}$ the vector $\langle x_1(\delta_1), x_2(\delta_2), \cdots, x_M(\delta_M)\rangle$. We call it a *distance vector* since its length is the distance from the epicenter to the bucket with perturbation $\vec{\delta}$. Note that the notation $\overrightarrow{x(\delta)}$ is "backward compatible" with the notation $\overrightarrow{x(1)}$, since when $\vec{\delta} = \langle 1, 1, \cdots, 1\rangle$, the former vector is the same as the latter vector.

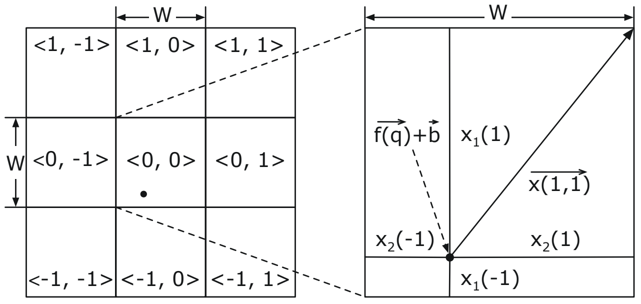


Figure 1: A toy example on multi-probe

We illustrate these notations using a two-dimensional toy example shown in Figure 1. In this example, $M = 2$ and the "neighborhood" of $\overrightarrow{h(q)}$ contains 8 equal-sized buckets. Each bucket is geometrically a $W \times W$ rectangle (here $W = 10$) and is represented by its perturbation vector. For example, the bucket in the center with perturbation vector $\langle 0, 0\rangle$ is the epicenter bucket. In this example, the distances between the epicenter and the four "faces" of the epicenter bucket are $x_1(1)$, $x_2(1)$, $x_1(-1)$, and $x_2(-1)$ respectively, and the distance vector is $\overrightarrow{x(1,1)}$.

We now fix another arbitrary point $\vec{s}$ that is a nearest neighbor of the query point $\vec{q}$. Suppose the L_2 distance between $\vec{s}$ and $\vec{q}$ is d_2 (the subscript of which refers to L_2 distance). The second refinement is based on the following fact established in [36]: The probability for $\vec{s}$ to land in a bucket with perturbation $\vec{\delta}$ is roughly proportional to $e^{-\|\overrightarrow{x(\delta)}\|_2^2/(2d_2^2)}$, where the distance vector $\overrightarrow{x(\delta)}$ was defined above and $\|\overrightarrow{x(\delta)}\|_2^2 = \sum_{i=1}^M x_i^2(\delta_i)$. This approximation formula $e^{-\|\overrightarrow{x(\delta)}\|_2^2/(2d_2^2)}$ implies that the (approximate) success probability of (finding $\vec{s}$ in) a bucket decreases when its squared distance from the epicenter increases. Hence a near-optimal probing sequence $S_T(\overrightarrow{x(1)})$ can instead be obtained by sorting these squared distances (using the aforementioned heap) in the increasing order, which is much cheaper than computing and sorting the corresponding success probabilities.

The third refinement is to precompute a *unique* (for any M) universal (for all future queries) template from which a near-optimal probing sequence for any given query can be instantiated. Given a query $\vec{q}$, the universal template and the order among the $2M$ distances $x_i(-1)$ and $x_i(1)$ for $i = 1, 2, \cdots, M$ uniquely determine the probing sequence. Since the third refinement involves the sorting of only $2M$ numbers (to determine this order) for each query, it has a smaller time complexity than the second refinement, which involves $O(T)$ heap operations. We omit the detailed description of this template and its instantiation here, since it has been implemented and used many times by the database community.

3 RW-LSH AND MP-RW-LSH

In this section, we first describe random-walk LSH (RW-LSH), a new LSH scheme for ANNS-L_1. Then we describe MP-RW-LSH, the multi-probe enhancement of RW-LSH. Throughout this section, we focus on the operations in a single hash table.

3.1 The RW-LSH Scheme

To describe RW-LSH, we need to define what a random walk is. Let $\tau^{(1)}, \tau^{(2)}, \cdots$ be a sequence of i.i.d. random variables. Each $\tau^{(i)}$ is a *single-step random walk* that takes value 1 or -1 with equal probability 1/2, and its value (realization), once generated, is fixed. The resulting (deterministic) sequence of values, denoted simply as τ, is called a random walk. With a slight abuse of notation, we denote as $\tau(t)$ the position after t steps along the random walk τ starting at the origin; that is, $\tau(t) \triangleq \tau^{(1)} + \cdots + \tau^{(t)}$.

It suffices to define a single raw hash value function f, since as explained earlier an RW-LSH function h is derived from f in the same way as in GP-LSH and CP-LSH: $h(\cdot) = \lfloor (f(\cdot) + b)/W \rfloor$. Suppose the dimension of the dataset $\mathcal{D}$ is m. Then f is a random walk projection parameterized by a vector of m mutually independent random walks $\vec{\tau} = \langle \tau_1, \cdots, \tau_m \rangle$; for the moment, we denote it as $f_{\vec{\tau}}$ to emphasize its dependence on $\vec{\tau}$. Then given a data point $\vec{s} = \langle s_1, s_2, \cdots, s_m \rangle$, $f_{\vec{\tau}}(\vec{s})$ is defined as $\sum_{i=1}^{m} \tau_i(s_i)$. We require that each s_i, $i = 1, 2, \cdots, m$, be a nonnegative even integer. This requirement is imposed on all data points in $\mathcal{D}$ and all query points. We will explain shortly why this assumption is not overly restrictive for real-world applications.

Let $\vec{q} = \langle q_1, q_2, \cdots, q_m \rangle$ be the query point. We denote as d_1 the value of the L_1 distance between $\vec{s}$ and $\vec{q}$, that is, $d_1 = \sum_{i=1}^{m} |s_i - q_i|$; d_1 is a nonnegative even integer since every s_i and q_i is. Then $f_{\vec{\tau}}(\vec{s}) - f_{\vec{\tau}}(\vec{q}) = \sum_{i=1}^{m} (\tau_i(s_i) - \tau_i(q_i))$ is a random walk of $\sum_{i=1}^{m} |s_i - q_i| = d_1$ steps, for two reasons. First, these m random walks $\tau_1, \tau_2, \cdots, \tau_m$ are mutually independent since they are along different sequences. Second, for each i, $i = 1, 2, \cdots, m$, $\tau_i(s_i) - \tau_i(q_i)$ has the same probability distribution as an $|s_i - q_i|$-step random walk along the sequence τ_i. For example, suppose along a dimension i, we have $s_i = 6$ and $q_i = 8$. Then $\tau_i(s_i) - \tau_i(q_i) = \tau_i(6) - \tau_i(8) = -\tau_i^{(7)} - \tau_i^{(8)}$ has the same distribution as a 2-step ($2 = |6 - 8|$) random walk, because $-\tau_i^{(7)}$ and $-\tau_i^{(8)}$ are independent and each has the same probability distribution as a single-step random walk, whose distribution is symmetric about 0.

We note that a d_1-step random walk, which we denote as Y_{d_1}, is parameterized only by d_1, as it has the following distribution: $\Pr[Y_{d_1} = l]$ is equal to $\binom{d_1}{(d_1+l)/2}(1/2)^{d_1}$ when l is an even integer satisfying $-d_1 \le l \le d_1$, and is equal to 0 otherwise. As a result, the collision probability $\Pr[h(\vec{s}) = h(\vec{q})]$ is a function of only d_1. We denote this probability as $p(d_1)$, since it is the collision probability (when hashed by h) of any two points the L_1 distance between which is d_1.

3.2 Analysis of $p(d_1)$

In this section, we prove Theorem 3.1, which states that $p(d_1)$ monotonically decreases when d_1 increases. This property is important since it qualifies RW-LSH as an LSH function family. Throughout this section, we consider $\vec{q}$ a query point and $\vec{s}$ a point in $\mathcal{D}$.

Theorem 3.1. When W is a positive even integer, the collision probability $p(d_1)$ decreases monotonically when d_1 takes on only nonnegative even integer values (which d_1 indeed does under our assumptions); that is, $p(0) > p(2) > p(4) > \cdots$.

Before we prove Theorem 3.1, we first derive the general formula of $\Pr[h(\vec{s}) - h(\vec{q}) = \delta | x(1)]$, where $\vec{s}$ and $\vec{q}$ are two points that are d_1 apart in L_1 distance as assumed above, δ is an arbitrary integer, and $x(1) = W - \{(f(\vec{q}) + b)/W\}\,W$. When h is a certain h_i (the LSH function in the i^{th} dimension), this $x(1)$ is precisely $x_i(1)$ as defined in §2.2. It is not hard to verify that

$$\Pr[h(\vec{s}) - h(\vec{q}) = \delta | x(1)] = \sum_{l=(\delta-1)W+\lceil x(1) \rceil}^{\delta W + \lfloor x(1) \rfloor} \Pr[Y_{d_1} = l]. \quad (1)$$

We now derive $p(d_1)$ from the general formula. When $\delta = 0$, we have $\Pr[h(\vec{s}) = h(\vec{q}) | x(1)] = \sum_{l=-W+\lceil x(1) \rceil}^{\lfloor x(1) \rfloor} \Pr[Y_{d_1} = l]$. Since $x(1)$ is uniformly distributed in $[0, W]$ (since b is uniformly distributed in $[0, W]$), we have

$$\begin{aligned} p(d_1) = \Pr[h(\vec{s}) = h(\vec{q})] &= E[\Pr[h(\vec{s}) = h(\vec{q}) | x(1)]] \\ &= \int_0^W \frac{1}{W} \sum_{l=-W+\lceil x(1) \rceil}^{\lfloor x(1) \rfloor} \Pr[Y_{d_1} = l]\, dx(1) \\ &= \sum_{l=-W}^{W} \left(1 - \frac{|l|}{W}\right) \Pr[Y_{d_1} = l]. \quad (2) \end{aligned}$$

In the following proof of Theorem 3.1, we drop the subscript 1 from d_1 in both places they appear in: $p(d_1)$ and Y_{d_1}.

Proof. It suffices to prove that, for any nonnegative even integer d, we have $p(d) > p(d+2)$. We have $p(d) = \sum_{l=-W}^{W} \left(1 - \frac{|l|}{W}\right) \Pr[Y_d = l] = \sum_{\ell=0}^{W} \left(1 - \frac{\ell}{W}\right) \Pr[|Y_d| = \ell] = \sum_{\ell=0}^{W-1} \sum_{t=0}^{W-\ell-1} \frac{1}{W} \Pr[|Y_d| = \ell] = \sum_{t=0}^{W-1} \sum_{\ell=0}^{W-t-1} \frac{1}{W} \Pr[|Y_d| = \ell] = \frac{1}{W} \sum_{t=0}^{W-1} \Pr[|Y_d| \le W - t - 1] = \frac{1}{W} \sum_{t=0}^{W-1} \Pr[|Y_d| \le t]$.

Replacing d with $d+2$ in the above equation, we obtain that $p(d+2) = \frac{1}{W} \sum_{t=0}^{W-1} \Pr[|Y_{d+2}| \le t]$. To prove $p(d) > p(d+2)$, we use the following *stochastic ordering* (see Definition 3.1 below) result established in [25]: $|Y_z| \le_{st} |Y_{z+2}|$ for any nonnegative integer z. By Definition 3.1, we have $\Pr[|Y_d| \le t] \ge \Pr[|Y_{d+2}| \le t]$ for $t = 1, 2, \cdots, W$. When $t = 0$, we have $\Pr[|Y_d| \le t] > \Pr[|Y_{d+2}| \le t]$ since $\Pr[|Y_d| = 0] - \Pr[|Y_{d+2}| = 0] = \frac{1}{2^{d+1}(d+2)} \binom{d+2}{(d+2)/2} > 0$. Hence $p(d) = \frac{1}{W} \sum_{t=0}^{W-1} \Pr[|Y_d| \le t] > \frac{1}{W} \sum_{t=0}^{W-1} \Pr[|Y_{d+2}| \le t] = p(d+2)$. □

Definition 3.1. Random variable X is said to be stochastically less than or equal to random variable Y, denoted as $X \le_{st} Y$, if and only if $\Pr[X \le t] \ge \Pr[Y \le t]$ for $-\infty < t < \infty$.

Now we connect $p(d_1)$ with the success probability for the baseline RW-LSH scheme (without multi-probe) to find $\vec{s}$ (given a query point $\vec{q}$) in the epicenter bucket in the hash table, which we denote as $\mathcal{P}_0(d_1)$. Using (2) and the fact that $h_1(\cdot)$, $h_2(\cdot)$, $\cdots$, $h_M(\cdot)$ are independent, we obtain

$$\mathcal{P}_0(d_1) = \prod_{i=1}^{M} \Pr[h_i(\vec{s}) - h_i(\vec{q}) = 0] = p^M(d_1). \tag{3}$$

3.3 Discussions on RW-LSH

Recall that we restrict the domain of each coordinate value of each data or query point (vector) to nonnegative even integers, the RW-LSH scheme can be extended to work without this restriction as follows. First, for each dimension i, we can increment (shift) the i^{th} coordinate of every (data or query) vector by a large enough positive constant a_i so that these i^{th} coordinates all become nonnegative. Second, we can multiply (scale) every vector by a large enough integer number C and then round each resulting scalar to the nearest even integer. It is clear that both the shift and the scaling operations preserve the ranked order among the L_1 distance values. Although rounding can cause changes to this ranked order, the percentage of such changes can be made extremely small, by increasing the C value, so that with overwhelming probability, an ANNS query over the original dataset has the same correct answer as that over the rounded scaled shifted dataset.

We now discuss an implementation issue of RW-LSH. As a common practice, each random walk sequence τ_i (for implementing a function $f_{\vec{\tau}}$) is implemented as a pseudorandom bit sequence wherein bit 0 is interpreted as -1. It certainly does not make sense to regenerate these m pseudorandom sequences when computing $f_{\vec{\tau}}(\vec{q})$ for each query point $\vec{q}$. Our baseline solution is to precompute and store each $\tau_i(t)$ for $t = 2, 4, 6, \cdots, U_i$ where U_i is the maximum possible (even) value for the i^{th} coordinate of a data point. Let the *universe* U be the maximum value among $U_1, U_2, \cdots, U_m$. For each hash table (with M hash functions), we need a maximum of mUM bytes for storing the precomputed table (one table entry costs 2 bytes for each even t value). For large datasets, this storage cost is small (typically more than an order of magnitude smaller) relative to the size of each hash table, since this cost is fixed in the sense it is independent of the size (number of points n) in $\mathcal{D}$. However, for some smaller datasets where mUM is much larger than n, this storage cost can significantly increase the total index size. For example, for the Enron dataset to be described in § 6, this storage cost increases the total index size by 709 times, from 75.3 MB to 52.2 GB. For such datasets, we employ a "long-jump" technique, to be described next, that can significantly reduce this storage cost while increasing the query time only slightly.

To simplify notation, we drop the subscript i from τ_i in the sequel, with the understanding that each τ_i is an instance of τ. Our solution, called "long-jump" (LJ), is to precompute and store $\tau(t)$ only for t values that are multiples of a large (relative to the jump size 2 in the baseline solution) integer jump size J, such as 64. Suppose J is 64, which would reduce this storage cost by 32 times. Then, for any $t > 0$, the value of $\tau(t)$ can be computed as follows. We write $t = 64l + r$, where $l = \lfloor t/64 \rfloor$ is the quotient (of dividing t by 64) and r is the remainder. We only need to compute an r-step random walk $\tau(t) - \tau(64l) = \tau^{(64l+1)} + \cdots + \tau^{(64l+r)}$ in this case, since $\tau(64l)$ is precomputed and stored. Using a pseudorandom sequence (bitmap) implementation as just described, computing this r-step random walk boils down to counting the number of $1's$ in a 64-bit (in this case) bitmap, which can be accomplished in one CPU cycle using a built-in instruction called `__builtin_popcount` (in the GCC compiler) that is supported on Intel and AMD processors. This popcount-based implementation can easily scale to very large J values. For example, even when $J = 2,048$ (which reduces the storage cost by 1,024 times), no more than 32 popcount operations are involved.

3.4 Multi-Probe Extension

From this point on, we drop the subscript $\vec{\tau}$ from $f_{\vec{\tau}}$. The multi-probe extension of RW-LSH (to MP-RW-LSH) is straightforward: It is identical to that of GP-LSH. This "porting" is possible for the following reasons: Recall that in both RW-LSH and GP-LSH, an LSH function h is defined as $h(\cdot) = \lfloor (f(\cdot) + b)/W \rfloor$. They differ only in (the choice of) the raw hash value function $f(\cdot)$. Recall that the following property of a Gaussian projection $f(\cdot)$ is a sufficient condition for all three refinements (for computing $S_T(\overrightarrow{x(1)})$) to work for GP-LSH: For any two points $\vec{s}$ and $\vec{q}$, $f(\vec{s}) - f(\vec{q})$ has a zero-mean Gaussian distribution (with variance $d_2^2 = \|\vec{s} - \vec{q}\|_2^2$). However, this zero-mean Gaussian distribution (with variance $d_1 = \|\vec{s} - \vec{q}\|_1$) property continues to hold approximately when $f(\cdot)$ is instead a random walk projection, especially when d_1 is not tiny ($d_1 \geq 20$).

As just explained, our MP-RW-LSH solution "inherits" all three refinements. We have measured, on a dataset containing 1 million points, the amounts of time needed for MP-RW-LSH to compute the exact or an approximate $S_T(\overrightarrow{x(1)})$ (which is a part of the query time) using the three refinements respectively. We have found that the second refinement is 2 to 3 orders of magnitude faster than the first refinement, and the third refinement is 2 to 3 times faster than the second refinement. We have also found that the approximate $S_T(\overrightarrow{x(1)})$ computed using the third refinement is very close to the exact $S_T(\overrightarrow{x(1)})$ computed using the first refinement, and as a result leads to a negligible loss of query accuracy, as will be reported at the end of § 4. For these reasons, we conclude that the third refinement achieves the best tradeoff between query time and query accuracy, and hence use it in our implementation and evaluation of MP-RW-LSH. However, the exact $S_T(\overrightarrow{x(1)})$ is used in all success probability analyses of MP-RW-LSH, to be shown next.

3.5 Analysis of $\mathcal{P}_T(d_1)$

In this section, we derive the success probability $\mathcal{P}_T(d_1)$ of finding a point $\vec{s} \in \mathcal{D}$, whose L_1 distance from the query point $\vec{q}$ is d_1, in the top-(T+1) buckets using MP-RW-LSH. Note that $\mathcal{P}_0(d_1)$, derived in (3), is a special case of $\mathcal{P}_T(d_1)$, where $T = 0$. We emphasize that the following derivations are precise. In particular, here we do not use any Gaussian approximation of random walk that was used in the previous section to clearly explain MP-RW-LSH.

Recall that $\overrightarrow{x(1)} \triangleq \langle x_1(1), \cdots, x_M(1) \rangle$ is the coordinates vector (defined in § 2.1) of the epicenter in the epicenter bucket and that $\overrightarrow{x(1)}$ is uniformly distributed in the cube $[0, W]^M$. Recall that, given a realization of $\overrightarrow{x(1)}$ (still denoted as $\overrightarrow{x(1)}$ here with a slight abuse of notation), the optimal probing sequence $S_T(\overrightarrow{x(1)})$, comprised of the top-($T$+1) buckets, is fully determined. Let each bucket in $S_T(\overrightarrow{x(1)})$ be referenced by its perturbation vector $\vec{\delta}$. Like before, we assume that M independent hash functions $h_1(\cdot)$, $h_2(\cdot)$, $\cdots$, $h_M(\cdot)$, which we denote as $\overrightarrow{h(\cdot)}$, are used for the hash table. The conditional (upon $\overrightarrow{x(1)}$) success probability of finding $\vec{s}$ in the bucket with perturbation vector $\vec{\delta}$ can be calculated as follows:

$$\Pr[\overrightarrow{h(s)} - \overrightarrow{h(q)} = \vec{\delta}|\overrightarrow{x(1)}] = \prod_{i=1}^{M} \Pr[h_i(\vec{s}) - h_i(\vec{q}) = \delta_i | x_i(1)] \quad (4)$$

where each $\Pr[h_i(\vec{s}) - h_i(\vec{q}) = \delta_i|x_i(1)]$ is computed using (1).

The conditional probability (upon $\overrightarrow{x(1)}$) for $\vec{s}$ to be hashed to one of the top-(T+1) buckets is $\sum_{\vec{\delta}\in S_T(\overrightarrow{x(1)})} \Pr[\overrightarrow{h(s)} - \overrightarrow{h(q)} = \vec{\delta}|\overrightarrow{x(1)}]$, in which each summand is calculated using (4). We denote this conditional probability as $\mathcal{P}_T(d_1, \overrightarrow{x(1)})$. The unconditional success probability $\mathcal{P}_T(d_1)$ is the average of the conditional probability $\mathcal{P}_T(d_1, \overrightarrow{x(1)})$ over the uniform distribution $W^{-M} d^M \overrightarrow{x(1)}$, where $d^M \overrightarrow{x(1)} \triangleq dx_1(1)dx_2(1)\cdots dx_M(1)$. That is:

$$\mathcal{P}_T(d_1) = \int_{\overrightarrow{x(1)}\in[0,W]^M} \mathcal{P}_T(d_1, \overrightarrow{x(1)}) \cdot W^{-M}\, d^M\overrightarrow{x(1)} \quad (5)$$

Finally, we relate this quantity $\mathcal{P}_T(d_1)$ to the query accuracy. Suppose L hash tables are used in a MP-RW-LSH. Then $1 - (1 - \mathcal{P}_T(d_1))^L$ is the probability that $\vec{s}$ appears in at least one of the L lists of top-(T + 1) buckets. If $\vec{s}$ is indeed the nearest neighbor of $\vec{q}$, then this probability is precisely the expected recall value of the one nearest neighbor (1-NN) query given $\vec{q}$ as the query point. Note this probability is a constant unaffected by how other data points in $\mathcal{D}$ are distributed.

4 CAN CP-LSH BE MULTI-PROBED ALSO?

As mentioned earlier, Cauchy-projection LSH (CP-LSH) is the state-of-the-art baseline LSH scheme for ANNS-L_1. In this section, we will show that CP-LSH is actually a slightly better baseline LSH scheme than RW-LSH. We will also show, however, that CP-LSH is fundamentally unsuitable for multi-probe, and as a result, the multi-probe CP-LSH (MP-CP-LSH) would perform far worse than the multi-probe RW-LSH (MP-RW-LSH).

Recall that the quality ρ of an LSH function family measures and determines the efficacy of the corresponding (baseline) LSH scheme, since both the query time complexity of the LSH scheme and its space complexity in terms of the number of hash tables are $O(n^\rho)$. We have found that the quality value ρ of RW-LSH is slightly larger (worse) than that of CP-LSH. For example, when $r_1 = 6$ (near radius) and $r_2 = 12$ (far radius), the best attainable quality value of RW-LSH, reached when the bucket width W is set to 8, is $p_1 = p(6) = 0.7656$, $p_2 = p(12) = 0.6633$ and $\rho = \log p_1/\log p_2 = 0.6506$. For the same far and near radii, the best attainable quality value of CP-LSH, reached when $W = 20$, is $p_1 = 0.5763$, $p_2 = 0.4021$, and $\rho = \log p_1/\log p_2 = 0.6050$. Here the p_1 and the p_2 values of RW-LSH are calculated using (2), the formula of $p(d_1)$ derived in §3.2. Those of CP-LSH are similarly derived and calculated using formula from [18].

Because of the difference in ρ, RW-LSH is slightly less efficient, in terms of both memory space (in number of hash tables) and query time (both are $O(n^\rho)$ as just explained), than Cauchy projection LSH (CP-LSH). We will show that, with multi-probing, MP-RW-LSH can successfully reduce the number of hash tables to almost a constant (typically between 6 and 8), so this quality ρ no longer affects its space complexity. However, the time complexity of MP-RW-LSH remains $O(n^\rho)$, since $O(n^\rho)$ buckets still have to be probed except that these buckets are now spread over 6 to 8 (instead of $O(n^\rho)$) hash tables. This, combined with a slightly larger ρ value for RW-LSH, explains why the query time of MP-RW-LSH is slightly higher that of CP-LSH shown in §6.3.

Table 1: $\mathcal{P}_T(d_1)$ w/ optimal probing sequences.

d_1	MP-RW-LSH			MP-CP-LSH		
	T=30	T=60	T=100	T=30	T=60	T=100
6	0.50	0.63	0.72	0.0405	0.0568	0.0716
8	0.36	0.48	0.57	0.0137	0.0203	0.0268
12	0.19	0.27	0.34	0.0018	0.0030	0.0043
16	0.10	0.15	0.20	0.0003	0.0005	0.0008

We now explain why, despite that RW-LSH has a worse quality ρ than CP-LSH, RW-LSH is much better suited for multi-probe extension than CP-LSH. We do so by comparing $\mathcal{P}_T(d_1)$ of their respective multi-probe extensions MP-RW-LSH and MP-CP-LSH. Like before, here d_1 is the L_1 distance between a query point $\vec{q}$ and a point $\vec{s}$ (in $\mathcal{D}$) that is a nearest neighbor of $\vec{q}$. Recall from §3.5 that $\mathcal{P}_T(d_1)$ is the success probability of finding $\vec{s}$ in the top-(T + 1) buckets along the optimal probing sequence. For MP-RW-LSH, (5) is the formula for calculating $\mathcal{P}_T(d_1)$, and for MP-CP-LSH, a similar formula can be derived. We compare $\mathcal{P}_T(d_1)$ values under MP-RW-LSH and MP-CP-LSH. For a fair comparison, M is set to a typical value of 10 in both baselines RW-LSH and CP-LSH; and like in the quality (ρ) comparison example above, W is set to 8 in RW-LSH and set to 20 in CP-LSH to achieve a respective optimal or near-optimal ρ value for $r_1 = 6$ (near radius) and $r_2 = 12$ (far radius).

The comparison results are shown in Table 1. For both algorithms, we calculate and demonstrate in Table 1 the $\mathcal{P}_T(d_1)$ values for the following 12 value combinations of d_1 and T: $d_1 = 6, 8, 12, 16$ and $T = 30, 60, 100$. Table 1 shows, for the same T and d_1, the $\mathcal{P}_T(d_1)$ values under MP-CP-LSH are one to two orders of magnitude smaller than those under MP-RW-LSH; this "top-light" behavior of MP-CP-LSH is expected since the Cauchy distribution underlying CP-LSH is heavy-tailed [42]. As a result, MP-CP-LSH would need a much larger number of hash tables to achieve the same query accuracy (success probability) as MP-RW-LSH. For example, when $T = 100$ and $d_1 = 8$, MP-RW-LSH needs to use only 6 hash tables to achieve a success probability of $1-(1-0.57)^6 = 0.99$, whereas MP-CP-LSH needs to use 186 hash tables to do the same. Hence we conclude that whereas multi-probe significantly reduces the number of hash tables and correspondingly the index size for RW-LSH, it offers no or little such improvement for CP-LSH.

Finally, we explain another reason why CP-LSH is not suitable for multi-probe extension. The second and the third refinements (for computing $S_T(\overrightarrow{x(1)})$) are inapplicable to CP-LSH, since an approximation formula that can simplify the success probability computation and comparison, such as $e^{-\|\overrightarrow{x(\delta)}\|_2^2/(2d_2^2)}$ for GP-LSH (and

RW-LSH), does not appear to exist for CP-LSH. Hence, MP-CP-LSH can use only the first refinement which, as just explained, is roughly three orders of magnitude slower than the third refinement. As a result, the query time of MP-CP-LSH, of which the $S_T(\overrightarrow{x(1)})$ computation time is a small but nontrivial part, would be a few times longer than that of MP-RW-LSH.

As explained at the end of §3.4, in our evaluations next, for MP-RW-LSH, we compute an approximate $S_T(\overrightarrow{x(1)})$ using the template-based algorithm (the third refinement described in §2.2), since it reduces the computation time of $S_T(\overrightarrow{x(1)})$ by roughly three orders of magnitude, as just explained. Our simulations show that this approximation reduces the success probability values (attained when using the exact $S_T(\overrightarrow{x(1)})$) shown in Table 1 by only 5% to 10%, and hence sacrifices the query accuracy only slightly.

5 ANNS-E AS A USE CASE FOR MP-RW-LSH

In this section, we describe a use case of our MP-RW-LSH: ANNS in the edit distance (ANNS-E), where the edit distance between two strings x and y is defined as the minimum number of symbol insertions, deletions, and replacements that are needed to change x to y. This use case also serves as another motivation for ANNS-L_1 as explained earlier. ANNS-E is more challenging than most other ANNS problems for the following reason: Whereas computing almost any other distance between two m-dimensional points has a time complexity of $O(m)$, computing the edit distance between two $O(m)$-symbol-long strings has a high time complexity of $O(m^2)$ using the textbook dynamic programming algorithm for the longest common substring [15]. For this reason, an ANNS-E solution can afford to perform this computation for only a very *short list* of candidates, unless m is small. However, to achieve a high query accuracy, this short list must be of high-quality in the sense it includes the vast majority of the true nearest neighbors in edit distance. To generate a short yet high-quality list is a challenge that any efficient ANNS-E solution has to address.

5.1 iDEC: the State of the Art

The state-of-the-art ANNS-E solution is proposed in [23] and based on a framework called indexable distance estimating codes (iDEC). It first converts an ANNS-E problem into an ANNS-L_1 problem, and then solves the resulting ANNS-L_1 problem. A key innovation in the iDEC-based ANNS-E solution is the introduction of two excellent features: *multiset* and *context*. Each feature, denoted as $\vec{\mu}(\cdot)$, maps a string y to a feature vector $\vec{\mu}(y)$ that is of a constant dimension. For example, the multiset feature maps y to a multiset of q-grams [41] (also called q-shingles in the literature); the resulting feature vector $\vec{\mu}(y)$ is the multiplicity vector of the resulting multiset, and its dimension is the number of possible q-gram values, which is clearly a constant. Both features are excellent for the ANNS-E purpose in that each maps two strings that are close in edit distance to two feature vectors that are close in L_1 distance, as shown in [23]. Hence both features convert an ANNS-E (over $\mathcal{D}$) problem to an ANNS-L_1 (over $\vec{\mu}(\mathcal{D})$) problem. This ANNS-L_1 (over $\vec{\mu}(\mathcal{D})$) problem remains challenging, since the dimension of $\vec{\mu}(\mathcal{D})$ can still be quite large (say tens to hundreds).

For both features, the resulting ANNS-L_1 (over $\vec{\mu}(\mathcal{D})$) problem is solved using iDEC-ToW4L1. Here ToW4L1, which stands for Tug-of-War for L_1, is a variant of the Tug-of-War sketch (originally proposed in [7] for estimating the L_2 norm of a data stream [19]) for estimating the L_1 distance of two multisets [19]. A ToW4L1 function, denoted as $\vec{\xi}(\cdot)$, maps each $\vec{\mu}(y) \in \vec{\mu}(\mathcal{D})$, which has a higher dimension, to a low-dimensional (say between 6 and 12) iDEC vector $\overrightarrow{\xi(\mu(y))} = \langle \xi_1(\overrightarrow{\mu(y)}), \xi_2(\overrightarrow{\mu(y)}), \cdots, \xi_M(\overrightarrow{\mu(y)})\rangle$. As shown in [23], the ToW4L1 function $\vec{\xi}(\cdot)$ maps any two feature vectors that are close in L_1 distance to two iDEC vectors that are close in L_1 distance.

The iDEC-based solution contains two variants that use the multiset and the context features respectively. In both variants, the iDEC-based solution works as follows. At the indexing stage, iDEC maps the set of feature vectors $\vec{\mu}(\mathcal{D})$ into a low-dimensional point set $\overrightarrow{\xi(\mu(\mathcal{D}))} \triangleq \{\overrightarrow{\xi(\mu(s))} \mid s \in \mathcal{D}\}$ using the aforementioned ToW4L1 function $\vec{\xi}(\cdot)$. Then given a query string x, iDEC computes an ANNS-E of x in two steps. First, iDEC searches in the "projection image" $\overrightarrow{\xi(\mu(\mathcal{D}))}$ for t exact nearest neighbors (t-NN) of $\overrightarrow{\xi(\mu(x))}$ over L_1 distance. This t-NN search can be computed very efficiently by organizing $\overrightarrow{\xi(\mu(\mathcal{D}))}$, a low-dimensional point set, as a k-d tree. Second, the exact edit distances between these t nearest neighbors and the query string x are computed and compared, to arrive at the final ANNS-E query result.

5.2 Our ANNS-E Solution

Our ANNS-E solution also contains two variants that use the multiset and the context features respectively. Each variant simply replaces iDEC-ToW4L1 with MP-RW-LSH in solving the ANNS-L_1 problem as follows. At the indexing stage, MP-RW-LSH takes as input the set of feature vectors $\vec{\mu}(\mathcal{D})$, and organizes the RW-LSH hash value vectors into L hash tables, as described in §3.1. Then given a query string x, MP-RW-LSH computes an ANNS-E of x in three steps. First, MP-RW-LSH "multi-probes" the L hash tables for the ANNS candidates (in $\vec{\mu}(\mathcal{D})$) of $\vec{\mu}(x)$ (the feature vector of x) in L_1 distance. Second, the exact L_1 distances between these candidates and x are then calculated and compared for generating a short list of $\mathcal{K}$ (typically less than 100) finalists. The purpose of this step is to filter out the vast majority of low-quality ANNS-E candidates so that the list of (surviving) finalists is both short and of high-quality, which is critical for achieving high query efficacy as explained at the beginning of §5. For this step, our MP-RW-LSH solution needs to keep in memory a copy of the feature vectors set $\vec{\mu}(\mathcal{D})$. Third, the exact edit distances between only these $\mathcal{K}$ finalists and the query string x are computed and compared, to arrive at the final ANNS-E query result.

Unlike our solution, the iDEC-based solution does not have the second step (of L_1-distance-based filtering using $\vec{\mu}(\mathcal{D})$) since, to make the index size competitively small, it does not keep a copy of $\vec{\mu}(\mathcal{D})$ in memory. As a result, the iDEC-based solution has to check a much longer list of candidates (typically in thousands), whose (low-dimensional) iDEC vectors are close to $\overrightarrow{\xi(\mu(x))}$ (the iDEC vector of the query x) in L_1 distance, for their closenesses to x in edit distance, which is computationally very expensive as explained earlier. Augmenting the iDEC-based solution with the second step significantly improves its query time, making it a much worthier competitor for our solution; we refer to the augmented solution as A-iDEC. Hence, in evaluating our ANNS-E solution in §7, we compare it with A-iDEC instead of with iDEC. We do so also for the following reason. Since A-iDEC and our solution differ only in the first step, which performs ANNS-L_1 queries (to generate the ANNS-E candidates), comparing A-iDEC and our solution boils down to comparing their respective ANNS-L_1 query efficacies. The latter comparison is precisely the objective of our ANNS-E evaluations, since our ANNS-E solution is proposed here only as a use case of our ANNS-L_1 solution, and should be evaluated as such.

6 ANNS-L_1 PERFORMANCE EVALUATION

In this section, we evaluate the ANNS-L_1 query performance of MP-RW-LSH against those of the following four LSH schemes: CP-LSH, RW-LSH (its baseline LSH without multi-probe), SRS [49] and QALSH [26]. All five algorithms except QALSH are implemented and optimized for in-memory operations, and are hence evaluated as such. Since QALSH was originally implemented and optimized for external-memory operations, to fairly compare QALSH with others without modifying its code, we run QALSH on a Ubuntu RAM disk so that its disk I/O's become memory reads/writes. CP-LSH (in terms of query efficiency), SRS (in terms of scalability) and QALSH are three state-of-the-art LSH solutions for ANNS-L_1. Our evaluations show conclusively that although its baseline RW-LSH is "mediocre" compared to CP-LSH and SRS (but RW-LSH still outperforms QALSH), MP-RW-LSH achieves a much better tradeoff between the query efficiency and scalability than CP-LSH, SRS, and QALSH. In §6.6, we have also evaluated the query performance of MP-RW-LSH against a non-LSH-based algorithm called FLANN [37].

6.1 Experiment Settings

Evaluation Datasets. We use eight widely used publicly available datasets of diverse dimensions, sizes (number of points), and types. The SIFT50M dataset contains 50 million points sampled uniformly at random from the 1 billion points contained in SIFT1B [3]. We cannot use SIFT1B instead since the resulting index structures of CP-LSH, RW-LSH and QALSH would not fit into the main memory (Those of MP-RW-LSH and SRS can). We normalize (scale and round as described in § 3.3) the coordinates of all data points to nonnegative even integers in all eight datasets. For each of the eight nominalized datasets, Table 2 shows its size n, its dimension m, the number of queries n_q processed on it, its universe U (defined in §3.3) and its type. We drop the word "normalized" in the sequel with the understanding that all datasets we refer to by names have been normalized.

Performance Metrics. We evaluate the performances of these five algorithms in three aspects: scalability, query efficiency, and query accuracy. To measure scalability (how well an algorithm can scale to very large datasets), we use the *index size* (excluding the size of the original dataset). For each query, each algorithm being evaluated needs to find $k = 50$ nearest neighbors in L_1 distance; using any other value of k ranging from 1 to 100 (commonly used in the ANNS literature [20, 26, 36, 59]) results in similar query accuracies of all algorithms except QALSH, whose query accuracy decreases more rapidly than all other algorithms when k increases [59]. To measure query efficiency, we use *query time*. To measure query accuracy, we use *recall*, which we define carefully next such that it can correctly accommodate the tie situation (of two distinct points having the same distance to the query point). Given a query point $\vec{q}$, we define $D(R) = \{||\vec{q}, \vec{o}_1||_1, ||\vec{q}, \vec{o}_2||_1, \cdots, ||\vec{q}, \vec{o}_k||_1\}$, where $\vec{o}_1, \vec{o}_2, \cdots, \vec{o}_k$ are the k (ANN) points returned by the algorithm that are sorted in the increasing order (with ties broken arbitrarily) of their L_1 distances to $\vec{q}$. Since there can be ties, $D(R)$ is in general a mutliset. Similarly, we define $D(R^*) = \{||\vec{q}, \vec{o}^*_1||_1, ||\vec{q}, \vec{o}^*_2||_1, \cdots, ||\vec{q}, \vec{o}^*_k||_1\}$, where $\vec{o}^*_1, \vec{o}^*_2, \cdots, \vec{o}^*_k$ are the true k nearest points to $\vec{q}$. The recall value is computed as $|D(R) \cap D(R^*)|/k$. Each query time or recall value presented in this section is the average over n_q queries, where n_q for each dataset is shown in Table 2.

Table 2: Datasets summary.

	Dataset	n	m	n_q	U	Type
Small	Audio [2]	53.3K	192	200	200K	Audio
	MNIST [55]	69.0K	784	200	2K	Image
	Enron [4]	95.0K	1,369	200	505K	Text
	Trevi [53]	99.9K	4,096	200	510	Image
Medium	GIST [3]	1.0M	960	1K	3K	Image
	Glove [39]	1.2M	100	200	25K	Text
Large	Deep10M [11]	10.0M	96	10K	3K	Image
	SIFT50M [3]	50.0M	128	10K	510	Image

Implementation Details. We implement RW-LSH functions, CP-LSH functions, and the multi-probe framework with the third refinement (template-generated probing sequence) in C++. For indexing and querying in LSH, we use an efficient open-source C++ LSH implementation called FALCONN [5]. For SRS and QALSH, we use the C++ source code provided by their authors. We compile all C++ source code using g++ 7.5 with -O3. All experiments are done on a workstation running Ubuntu 18.04 with Intel(R) Core(TM) i7-9800X 3.8 GHz CPU, 128 GB DRAM and 4 TB hard disk drive (HDD).

6.2 Benchmark Algorithms

We first briefly describe SRS [49], the only benchmark algorithm that has not been introduced before. The SRS framework is conceptually the same as the iDEC framework described in §5.1. Like that of iDEC, the idea of SRS is to first map each point $\vec{s} \in \mathcal{D}$ to an M-dimensional vector $\overrightarrow{f(s)} = \langle f_1(\vec{s}), f_2(\vec{s}), \cdots, f_M(\vec{s})\rangle$ and then perform the aforementioned t-NN search on $\vec{f}(\mathcal{D})$, which is organized as a cover tree (whereas iDEC uses a k-d tree). Intuitively, this algorithm works because the Cauchy projection vector $\overrightarrow{f(\cdot)}$ is statistically distance-preserving in the sense if the point $\vec{s}$ is among the closest points to $\vec{q}$ in L_1 distance, then $\overrightarrow{f(s)}$ is with high probability among the closest to $\overrightarrow{f(q)}$ in L_1 distance.

Now for each algorithm, we describe how we tune its parameters for the best query performance. In RW-LSH, MP-RW-LSH, and CP-LSH, we have three parameters to tune: M (the dimension of an LSH function vector), W (the bucket "width"), and L (the number of hash tables). In SRS, we have two parameters to tune: M and t (defined above in "t-NN search"). There is no L in SRS, since it uses a cover tree instead of hash tables as the index structure. In QALSH, we have one parameter to tune: the approximation ratio.

RW-LSH and MP-RW-LSH. For RW-LSH, we find that the following value combinations of (M, W) strike the best tradeoffs between query accuracy and query efficiency for the eight datasets listed in Table 2 from top to bottom respectively: $(12, 3144)$, $(12, 930)$, $(9, 122)$, $(16, 1728)$, $(8, 452)$, $(16, 1104)$, $(17, 424)$, $(14, 224)$. The same value combinations are used for MP-RW-LSH. MP-RW-LSH has an additional parameter to tune: T (number of additional buckets to be probed in each hash table). We find that $T = 100$ strikes near-optimal tradeoffs between query time and query accuracy for all eight datasets. For both RW-LSH and MP-RW-LSH, we adjust L to achieve a recall value larger than 0.9 for each dataset.

Recall that MP-RW-LSH can use the long-jump (LJ) technique described in §3.3 to reduce the total size of precomputed tables without noticeably increasing the query time. To show the effect of this reduction, the index sizes of MP-RW-LSH and RW-LSH are broken down into two parts: (1) the total size of the L hash tables, which we denote as v_1; and (2) the total size of the precomputed tables, which we denote as v_2. Each index size entry of MP-RW-LSH and RW-LSH in Table 3 is written as "v_1 ($+v_2$)". For MP-RW-LSH, we enable the long-jump technique on all small and medium datasets because, without this reduction, their v_2 can be much larger than v_1. We set the jump size J to be 64 on all small and medium datasets except Audio and Enron, and we set jump steps to 512 for Audio and to 8192 for Enron respectively. We do not enable LJ on the two large datasets Deep10M and SIFT50M, since their v_2 values without long-jump are already much smaller than their v_1 values. For RW-LSH, we do not enable LJ, and as a result its v_2 values are much larger than those of MP-RW-LSH on some datasets. In the following, we ignore the v_2 values (in parentheses) of MP-RW-LSH and RW-LSH in index size comparisons, since v_2 is less than 20% of v_1 in MP-RW-LSH on all eight datasets and it will become clear that including v_2 would not change the scalability narrative in the index size comparison between MP-RW-LSH and any other algorithm.

CP-LSH. We use the following near-optimal parameter settings for the eight datasets in the same order as above: $(M, W) = (7, 4401336)$, $(8, 384416)$, $(9, 22000)$, $(7, 561426)$, $(6, 153732)$, $(8, 303312)$, $(6, 34014)$, $(8, 17336)$. For each dataset, we adjust L to achieve a similar query accuracy as achieved by RW-LSH and MP-RW-LSH.

SRS. It was suggested by authors of SRS that M should range from 6 to 10 [49]. For all eight datasets, we find that $M = 10$ strikes roughly the best tradeoffs between query accuracy and query efficiency. As suggested by authors of SRS [49], we adjust parameter t to reach the same level of query accuracy as achieved by the other three algorithms for each dataset.

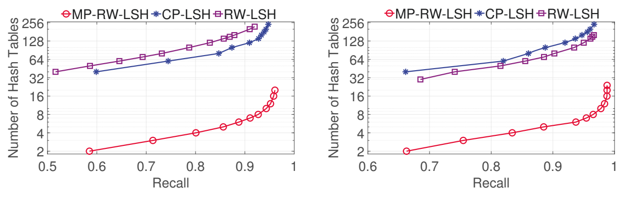
QALSH. QALSH [26] is an LSH-based ANNS-L_1 algorithm optimized for external-memory operations. We set the approximation ratio to 1.4 except in SIFT50M (where we set approximation ratio to 1.8 in order to fit the index structure into the RAM disk), and use e^{-1} for the error probability and $100/n$ for the false positive percentage as suggested by authors of QALSH [26]. For each dataset, we set the page size in QALSH in such a way that at least one object (point) can fit in one page. As a result, the page size is 4 KB for all eight datasets except Enron and Trevi, where the page sizes are 16 KB and 64 KB respectively.

6.3 Comparison with CP-LSH and RW-LSH

In this section, we compare MP-RW-LSH with CP-LSH and RW-LSH in terms of scalability and query efficiency. They all use hash tables and each hash table has the same size for the same dataset. Hence, their index sizes are proportional to the number of hash tables they use. In Table 3, we report the query times and the index sizes needed by all three algorithms for achieving similar query accuracies (if possible) on each dataset.

Scalability. Table 3 clearly shows that MP-RW-LSH has much better scalability than both CP-LSH and RW-LSH. On all eight datasets, the index sizes of, and equivalently the numbers of hash tables used by, MP-RW-LSH are 14.8 to 53.3 and 15.0 to 27.5 times smaller than those of CP-LSH and RW-LSH, respectively. Figure 2 shows the tradeoffs between recall values achieved and the numbers of hash tables used by these three algorithms on two medium datasets GIST and Glove. Figure 2a shows that for achieving the same recall value, CP-LSH and RW-LSH need to use roughly 18.2 to 20.1 and roughly 24.8 to 27.5 times more hash tables than MP-RW-LSH on GIST, respectively. Figure 2b shows that for achieving the same recall value, CP-LSH and RW-LSH need to use roughly 20.1 to 29.2 and roughly 13.9 to 19.4 times more hash tables than MP-RW-LSH on Glove, respectively. In fact, MP-RW-LSH can scale to the one-billion-point dataset SIFT1B (without sampling) [3] with an index size of roughly 24 GB, whereas the other four algorithms cannot (using the 128 GB memory the computer has) while achieving the same query accuracy as MP-RW-LSH.

Query Efficiency. As shown in Table 3, for achieving similar (or better) query accuracies, MP-RW-LSH has shorter query times on all the four small datasets and similar or slightly longer query times on all medium and large datasets than its baseline RW-LSH. Table 3 also shows that CP-LSH has between 1.3 and 2.2 times shorter query times than MP-RW-LSH on the eight datasets. The reason why the query time of CP-LSH is a bit shorter than that of MP-RW-LSH was explained in the third paragraph in §4. Overall, it is fair to say that MP-RW-LSH achieves a much better tradeoff between scalability and query efficiency than CP-LSH.



(a) GIST **(b) Glove**

Figure 2: Number of hash tables vs. recall.

6.4 Comparison with SRS

Scalability. As shown in Table 3, MP-RW-LSH has smaller index sizes on the two large datasets, but has larger index sizes on the six small and medium datasets, than SRS.

Query Efficiency. As shown in Table 3, MP-RW-LSH has much shorter query times than SRS for achieving similar query accuracies, especially on large datasets. Therefore, it is fair to say that overall

Table 3: Experiment results. CR indicates that the index structure crashed due to out of memory.

		MP-RW-LSH	CP-LSH	RW-LSH	SRS	QALSH	FLANN
Audio	Query Time (ms)	5.2	2.4	13.5	18.0	53.9	3.4
	Recall	0.9438	0.9298	0.8445	0.9140	0.9168	0.9426
	Index Size (MB)	65.6 (+13.7)	968.0	984.4 (+52743.4)	3.0	128.6	3.7
MNIST	Query Time (ms)	10.7	5.6	37.1	47.7	45.6	3.3
	Recall	0.9491	0.9309	0.9221	0.9314	0.9303	0.9492
	Index Size (MB)	66.1 (+4.4)	2644.2	1487.4 (+3230.0)	3.8	118.1	4.7
Enron	Query Time (ms)	47.0	40.8	**CR**	64.9	91.1	18.0
	Recall	0.9339	0.9376	**CR**	0.9323	0.7034	0.9339
	Index Size (MB)	75.3 (+13.0)	836.2	**CR**	5.4	268.9	6.5
Trevi	Query Time (ms)	69.6	40.4	147.1	86.8	110.1	31.8
	Recall	0.9210	0.9162	0.9055	0.9168	0.8283	0.9205
	Index Size (MB)	58.7 (+7.0)	2681.9	1005.7 (+3825.0)	5.7	412.0	13.7
GIST	Query Time (ms)	361.9	273.8	364.1	1045.8	2193.2	373.9
	Recall	0.9696	0.9640	0.9557	0.9602	0.9456	0.9669
	Index Size (MB)	94.5 (+5.4)	1654.0	2599.2 (+4834.0)	52.6	2928.2	171.7
Glove	Query Time (ms)	140.4	119.2	143.6	557.1	2486.4	357.4
	Recall	0.9809	0.9753	0.9751	0.9648	0.6606	0.9770
	Index Size (MB)	100.4 (+9.6)	3764.7	1886.6 (+5722.0)	63.6	3530.1	204.7
Deep10M	Query Time (ms)	1045.0	560.8	825.6	5338.8	48310.4	136.1
	Recall	0.9756	0.9758	0.9737	0.9565	0.7932	0.9750
	Index Size (MB)	323.0 (+32.7)	6922.0	6922.0 (+700.4)	525.2	23656.4	1716.6
SIFT50M	Query Time (ms)	5475.7	2445.0	3615.4	28302.9	126715.8	673.5
	Recall	0.9807	0.9809	0.9668	0.9595	0.5873	0.9801
	Index Size (MB)	1192.4 (+5.2)	19873.5	17886.1 (+470.7)	2656.8	63058.4	13732.91

MP-RW-LSH achieves a much better tradeoff between scalability and query efficiency than SRS.

6.5 Comparison with QALSH

Scalability. Table 3 shows that MP-RW-LSH has better scalability than QALSH while achieving similar or better query accuracy. On all eight datasets, the index sizes of MP-RW-LSH are 1.79 to 73.2 times smaller than those of QALSH.

Query Efficiency. Table 3 shows that MP-RW-LSH has between 1.8 to 16.4 times shorter query times than QALSH on all small and medium datasets. On the two large datasets, query times of QALSH are 23.1 and 46.2 times larger than those of MP-RW-LSH respectively, for achieving recall values (0.7932 and 0.5873) that are much lower than those of MP-RW-LSH (0.9756 and 0.9807). This asymptotic behavior (that query time advantage grows with dataset size n) is expected for the following reason. The query time of QALSH is roughly equal to memory reads/writes time and computation time adding up. Our measurements show that computation time dominates query time, especially on two large datasets. For example, computation time accounts for roughly 90% of the query time of QALSH on Deep10M. However, the computation time complexity of QALSH is $O(n \log n)$ [26], whereas the query time complexity of MP-RW-LSH is $O(n^\rho)$ (where $\rho < 1$) as explained in § 4, so MP-RW-LSH's advantage in query time grows with n.

6.6 Comparison with FLANN

Finally, we compare MP-RW-LSH with a prominent non-LSH-based solution suite called FLANN [6]. FLANN is mainly designed for ANNS-L_2, but can be configured to answer ANNS-L_1 queries, which we do here. The suite includes a randomized k-d tree algorithm [46] and a priority search k-means tree algorithm [21]. For this comparison, we use the former algorithm since the latter algorithm has a larger index size and is dataset-dependent (whereas MP-RW-LSH is dataset-independent). For each dataset, we set the number of trees to a value for the algorithm to achieves a nice tradeoff between query accuracy and index size, both of which increase when this number increases. For each dataset, we adjust the number of leaf nodes to be examined (as suggested in [37]) to match the query accuracy of MP-RW-LSH. We report the average query time and recall of FLANN over n_q queries, and its index size in Table 3.

Scalability. As shown in Table 3, the index sizes of MP-RW-LSH are larger than those of FLANN on the four small datasets, but are 1.8 to 11.5 times smaller on the four medium and large datasets.

Query Efficiency. As shown in Table 3, the query times of MP-RW-LSH are 1.5 to 8.1 times longer than those of FLANN on all eight datasets except GIST and Glove. The query time of MP-RW-LSH is slightly shorter on GIST and 2.5 times shorter on Glove. All these results can be explained by the intrinsic dimension (ID) values [8] of these datasets as follows. The original k-d tree algorithm [12] has a query time complexity of roughly $O(2^m + \log n)$ as shown in [31]. However, when randomized [17, 46, 51] as done in FLANN, its empirical query time complexity is known to remain the same asymptotic formula except that the dimension m therein is replaced by the much smaller ID of the dataset. The query time of MP-RW-LSH, on the other hand, is not sensitive to this ID. This explains why FLANN is faster on the six datasets whose ID's are relatively small (no more than 12.2 for all six [8, 34]) and slower on GIST and Glove whose ID's are relatively large (18.9 and 20 respectively [34]). It also predicts that the outperformance of MP-RW-LSH will grow larger when the ID (of the dataset) grows larger.

We note that MP-RW-LSH has a significant performance advantage over FLANN when performing queries over a dynamic dataset. In the case of FLANN, if a large number of new data points are inserted into the dataset after the k-d tree was built, the query efficiency can deteriorate significantly due to the k-d trees becoming severely unbalanced [29]. While rebuilding the trees solves this problem, it is very time-consuming. In comparison, in the case of MP-RW-LSH, inserting new data points has virtually no impact on query efficiency.

7 ANNS-E PERFORMANCE EVALUATION

In this section, we evaluate the two variants (multiset and context) of our solution, which we call "MP-RW-LSH" in the sequel, against the two variants of A-iDEC, in terms of ANNS-E query efficacy. For MP-RW-LSH, the long-jump technique (described in §3.3) is used with $J = 128$ in all four evaluation scenarios; the total size of the resulting precomputed tables is negligible compared to that of the hash tables and hence not accounted for in the index size.

7.1 Experiment Settings

Evaluation Datasets. We use four string datasets: GEN50KS [57], UNIREF [57], TREC [57] and Enron-E [16] (a string dataset derived from the aforementioned Enron dataset [4]). Table 4 summarizes the number of strings (excluding those in the query workload), the alphabet size $|\Sigma|$, the minimum, average and maximum lengths of the strings, and the type (DNA, protein sequences, medical or text) on the 4 datasets.

Table 4: Datasets summary.

Dataset	n	$\lvert\Sigma\rvert$	Length			Type	n_q
			Min	Avg	Max		
GEN50KS	49K	4	4,844	5,000	5,109	DNA	1,000
TREC	232K	37	80	1,217	3,947	Medical	1,000
Enron-E	244K	37	12	885	59,420	Text	1,000
UNIREF	399K	24	200	445	35,213	Protein	1,000

Performance Metrics. Like in the ANNS-L_1 evaluation, we evaluate the performances of MP-RW-LSH versus A-iDEC in three aspects: scalability, query efficiency, and query accuracy. For scalability and query efficiency, we use the same metrics as ANNS-L_1 in §6.1. Due to the aforementioned extreme difficulty of achieving high ANNS-E query accuracy, each algorithm being evaluated needs to find only $k = 1$ nearest neighbor with approximation ratio $c = 1.3$ (called c-ANN in the literature [22, 26]). In this case, *recall* is defined as follow. For a query string q, let the query result be o_1 and $o_1{}^*$ be the actual nearest neighbor. The recall value is 1 if $\|q, o_1\|_E \le c \cdot \|q, o_1{}^*\|_E$, where $\|\cdot,\cdot\|_E$ denotes the edit distance; otherwise the recall value is 0. Like in the ANNS-L_1 evaluation, each query time or recall value is the average over n_q queries, where n_q values are shown in Table 4.

Implementation Details. For the implementation of multiset feature, context feature and A-iDEC, we use the C++ source code provided by their authors [23], and run it on the same workstation as in the ANNS-L_1 evaluation (§6.1).

7.2 Benchmark Algorithms

For a fair comparison, for each dataset and feature (multiset or context) combination, the same parameter settings are used for MP-RW-LSH and A-iDEC. Specifically, for the multiset feature, we use the gram (shingle) size of $q = 3$ on GEN50KS and of $q = 1$ in on other three datasets; for the context feature, we use $q = 3$ on GEN50KS and $q = 1$ on the other three datasets respectively. Another parameter in the context feature is the window size w (which translates into a context size of $2w + 1$ symbols), which we set to 12 for all four datasets.

Now we describe the parameter settings for the two benchmark algorithms. For MP-RW-LSH, like in the ANNS-L_1 evaluation (§6.2), we have three parameters to tune: M and W which are already defined in §6.2 and $\mathcal{K}$, the number of finalists for the third (verification) step. For A-iDEC, we have three parameters to tune: M (dimension of the "projection image" $\overrightarrow{\xi(\mu(\mathcal{D}))}$), t (the number of candidates produced by the first step of searching $\overrightarrow{\xi(\mu(\mathcal{D}))}$ for ANNS-L_1) and the number of finalists $\mathcal{K}$.

Table 5: Query time (in *ms*) and index size (in MB) results.

Dataset	Feature	Algorithm	Time	Recall	Size
GEN50KS	Multiset	MP-RW-LSH	0.3+0.3+5.5	0.9980	32+0.8+6.0
		A-iDEC	10.9+0.2+5.5	0.9970	2.3+6.0
	Context	MP-RW-LSH	1.0+0.6+2.5	0.9970	32+0.8+6.0
		A-iDEC	39.7+0.4+2.5	0.9970	2.3+6.0
TREC	Multiset	MP-RW-LSH	1.4+5.8+11.9	0.9960	32+3.5+15.7
		A-iDEC	79.6+0.8+11.9	0.9960	10.9+15.7
	Context	MP-RW-LSH	1.2+5.3+7.6	0.9970	32+3.5+15.7
		A-iDEC	96.5+1.8+7.6	0.9970	10.8+15.7
Enron-E	Multiset	MP-RW-LSH	1.7+6.4+10.7	0.9050	32+3.7+17.3
		A-iDEC	76.3+0.7+10.7	0.9050	11.6+17.3
	Context	MP-RW-LSH	1.8+6.4+13.6	0.9050	32+3.7+17.3
		A-iDEC	240.3+3.0+13.6	0.9050	10.7+17.3
UNIREF	Multiset	MP-RW-LSH	6.6+21.9+1.7	0.9500	32+6.1+17.5
		A-iDEC	1648.7+4.7+1.7	0.9550	17.1+17.5
	Context	MP-RW-LSH	3.7+15.7+3.1	0.9550	32+6.1+17.5
		A-iDEC	1131.2+3.2+3.1	0.9550	18.2+17.5

The parameter settings of MP-RW-LSH are described as follows. We find that $L = 4$ (hash tables) and $T = 100$ leads to the best tradeoffs between query accuracy and query efficiency on both two features. We use the following value combinations of (M, W) that lead to the best such tradeoffs for the four datasets listed in Table 4 (from top to bottom) respectively: $(11, 78)$, $(9, 76)$, $(12, 62)$, $(8, 40)$ for MP-RW-LSH with the multiset feature and (8, 282), (9, 170), (10, 238), (8, 200) for MP-RW-LSH with the context feature. We adjust the number of finalists $\mathcal{K}$ to achieve a certain level of query accuracy for each dataset.

In all ANNS-L_1 and ANNS-E evaluations, to avoid introducing another tunable parameter, we fix the number of hash buckets in each hash table to $2^{21} \approx 2.1$ million in MP-RW-LSH, which results in a fixed (i.e., not growing with n) cost of 8 MB per hash table. In ANNS-L_1 evaluations, we include this fixed cost in index size comparisons because its impact is small on large datasets. We must exclude this fixed cost here for a fair comparison, because all four ANNS-E datasets are tiny (so using 2.1 million buckets is very wasteful), and as a result this fixed cost dominates all other costs. To this end, we divide the index size of MP-RW-LSH into three parts: this fixed cost of 32 MB (for the 4 hash tables), the total size of the 4 hash tables excluding this fixed cost, and the size of the feature vectors set $\vec{\mu}(\mathcal{D})$ (the filter that was described in §5).

The parameter settings of A-iDEC are as follows. We find that $M = 12$ strikes roughly the best tradeoffs between query accuracy and query efficiency on all four datasets and both features. For each feature (multiset or context) and dataset combination, the number of finalist $\mathcal{K}$ is set to the same value (that can vary from one combination to another) in both A-iDEC and MP-RW-LSH, for a fair comparison between their ANNS-L_1 query efficacies as explained in §5. We adjust the parameter t to reach the same level of query accuracy as achieved by MP-RW-LSH for each feature on each dataset. For a fair comparison, like what we did to MP-RW-LSH, we divide the index size of A-iDEC into two parts: the size of the "projection image" $\overrightarrow{\xi(\mu(\mathcal{D}))}$, and that of the filter $\vec{\mu}(\mathcal{D})$.

7.3 Comparison with A-iDEC

In Table 5, we report the average query times and the index sizes needed by MP-RW-LSH and A-iDEC respectively for achieving similar query accuracies when applying the same feature (multiset or context) on each dataset. For a more insightful comparison, for all four variants (algorithm+feature combinations), we break down each query time into three parts: the amount of time for performing the first (candidate-generating) step (t_1), that for performing the second (filtering) step (t_2), and that for performing the third (verification) step (t_3). Every query time entry in Table 3 is written as "t_1+t_2+t_3". Note that, for each dataset, t_3 is the same for each corresponding variant pair (MP-RW-LSH and A-iDEC with the same feature), since they use the same number of finalist $\mathcal{K}$ as explained earlier. Hence, our focus in the rest of this section is on comparing $t_1 + t_2$.

Table 5 shows that MP-RW-LSH is both more time-efficient and more scalable than A-iDEC. On one hand, for each of the four datasets, the query time of an MP-RW-LSH variant excluding t_3 is between 9.5 and 59.2 times shorter than that of the corresponding A-iDEC variant (for the same feature), for achieving similar query accuracies. On the other hand, for each of the four datasets, the index size of an MP-RW-LSH variant excluding the 32 MB fixed cost (the first number) is roughly 3 times smaller than that of the corresponding A-iDEC variant, when the cost of $\vec{\mu}(\mathcal{D})$ (the last number), is excluded from both.

As shown in Table 5, in the case of MP-RW-LSH, t_3 is larger than $t_1 + t_2$ on all four datasets except UNIREF, which confirms the importance of keeping $\mathcal{K}$ (the number of finalists) small for better ANNS-E query efficiency in general. UNIREF is an exception (on which t_3 is small), since the average string length in UNIREF is much smaller than those in the other three datasets (as shown in Table 4). For this same reason and the fact that t_1 is generally much longer than t_3 in the case of A-iDEC, for UNIREF, it is possible to reduce the total query time $t_1 + t_2 + t_3$ of A-iDEC by increasing its $\mathcal{K}$ (which increases t_3) and reducing the number of candidates generated in the first step (which decreases t_1). However, even with this optimization, the query times of A-iDEC are still several times longer than those of MP-RW-LSH on UNIREF. For example, the query time of A-iDEC with the context feature is 1131.2 + 3.2 + 3.1 = 1137.5 ms on UNIREF (see Table 5) when $\mathcal{K}$ is set to 88; and when $\mathcal{K}$ is increased to a near-optimal value of 2000, we can significantly reduce t_1 (while achieving the same recall) so that the query time of A-iDEC decreases to 29.7 + 1.2 + 66.9 = 97.8 ms. On the other three datasets, this optimization reduces the query times of A-iDEC only slightly, which is consistent with the reason explained earlier.

8 RELATED WORK

The ANNS literature is vast. Here in the interest of space we briefly survey only LSH-based solutions that we have not described earlier. Most of these solutions are for ANNS-L_2 and all are designed for external-memory operations. Examples of these solutions include LSB-forest [50], C2LSH [22], LazyLSH [60], and PM-LSH [59] (a variant of QALSH). Some of them, such as C2LSH and LazyLSH, can be adapted for ANNS-L_1. However, they all need a large number of hash tables (e.g. $L = 213$ for C2LSH as reported in [22]), so their index sizes are too large to fit in memory when the dataset is large. Therefore, they have to use disk-resident data structures, which result in long query time.

Another family of popular ANNS-L_1 solutions is tree-based algorithms. They have been widely used for both exact NNS and ANNS. Representative tree-based ANNS-L_1 algorithms include random k-d tree algorithm [46] and Annoy [1].

9 CONCLUSION

In this paper, we propose MP-RW-LSH, the first and so far only multi-probe LSH solution for ANNS-L_1 distance, and show that it achieves a better tradeoff between scalability and query efficiency than all existing LSH-based solutions. We also explain why CP-LSH, a state-of-the-art ANNS-L_1 solution, is fundamentally not suitable for multi-probe extension. As a use case, we construct, using MP-RW-LSH as the underlying "ANNS-L_1 engine", a new ANNS-E solution that beats the state-of-the-art solution called iDEC.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1909048 and CNS-2007006. We thank the reviewers for their suggestions that have greatly improved the quality and the readability of this paper.

REFERENCES

[1] [n.d.]. Annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk. https://github.com/spotify/annoy.
[2] [n.d.]. Audio Dataset. http://www.cs.princeton.edu/cass/audio.tar.gz.
[3] [n.d.]. Datasets for ANN neighbor search. http://corpus-texmex.irisa.fr/.

[4] [n.d.]. Enron Email Dataset. http://www.cs.cmu.edu/~enron/.

[5] [n.d.]. FALCONN - FAst Lookups of Cosine and Other Nearest Neighbors. https://github.com/FALCONN-LIB/FALCONN.

[6] [n.d.]. FLANN - Fast Library for Approximate Nearest Neighbors. https://github.com/flann-lib/flann.

[7] Noga Alon, Yossi Matias, and Mario Szegedy. 1999. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences* 58, 1 (Feb. 1999), 137–147.

[8] Laurent Amsaleg, Oussama Chelly, Teddy Furon, Stéphane Girard, Michael E. Houle, Ken-ichi Kawarabayashi, and Michael Nett. 2015. Estimating Local Intrinsic Dimensionality. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2783258.2783405

[9] Alexandr Andoni, Piotr Indyk, and Robert Krauthgamer. 2008. Earth Mover Distance over High-Dimensional Spaces. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '08)*. Society for Industrial and Applied Mathematics, USA, 343–352.

[10] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 1225–1233.

[11] Artem Babenko and Victor Lempitsky. [n.d.]. Deep: Datasets of deep descriptors. http://sites.skoltech.ru/compvision/noimi/.

[12] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (Sept. 1975), 509–517. https://doi.org/10.1145/361002.361007

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

[14] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *22nd International Conference on Data Engineering (ICDE'06)*. 5–5. https://doi.org/10.1109/ICDE.2006.9

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[16] Xinyan DAI, Xiao Yan, Kaiwen Zhou, Yuxuan Wang, Han Yang, and James Cheng. 2020. Convolutional Embedding for Edit Distance. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20)*. Association for Computing Machinery, New York, NY, USA, 599–608. https://doi.org/10.1145/3397271.3401045

[17] Sanjoy Dasgupta and Yoav Freund. 2008. Random Projection Trees and Low Dimensional Manifolds. In *Proceedings of the ACM Symposium on Theory of Computing*. New York, NY, USA, 537–546. https://doi.org/10.1145/1374376.1374452

[18] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. 253–262.

[19] Joan Feigenbaum, Sampath Kannan, Martin J. Strauss, and Mahesh Viswanathan. 2003. An Approximate L1-Difference Algorithm for Massive Data Streams. *SIAM J. Comput.* 32, 1 (Jan. 2003), 131–151. https://doi.org/10.1137/S0097539799361701

[20] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. *PVLDB* 12, 5 (2019), 461–474. https://doi.org/10.14778/3303753.3303754

[21] Keinosuke Fukunaga and Patrenahalli M. Narendra. 1975. A Branch and Bound Algorithm for Computing k-Nearest Neighbors. *IEEE Trans. Comput.* C-24, 7 (July 1975), 750–753. https://doi.org/10.1109/T-C.1975.224297

[22] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-Sensitive Hashing Scheme Based on Dynamic Collision Counting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Scottsdale, Arizona, USA, 541–552. https://doi.org/10.1145/2213836.2213898 Source code: https://github.com/fengjl18/C2LSH-Code.

[23] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. 2020. iDEC: Indexable Distance Estimating Codes for Approximate Nearest Neighbor Search. *PVLDB* 13, 9 (2020), 1483–1497.

[24] David Gorisse, Matthieu Cord, and Frederic Precioso. 2012. Locality-Sensitive Hashing for Chi2 Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34, 2 (2012), 402–409. https://doi.org/10.1109/TPAMI.2011.193

[25] Raymond J Hickey. 1983. Majorisation, randomness and some discrete distributions. *Journal of applied probability* (1983), 897–902.

[26] Qiang Huang, Jianlin Feng, Qiong Fang, Wilfred Ng, and Wei Wang. 2017. Query-aware locality-sensitive hashing scheme for l_p norm. *The VLDB Journal* 26, 5 (2017), 683–708.

[27] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *STOC*. ACM, 604–613.

[28] Piotr Indyk and Nitin Thaper. 2003. Fast image retrieval via embeddings. In *3rd international workshop on statistical and computational theories of vision (at ICCV)*, Vol. 2. 5.

[29] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. 2017. A progressive k-d tree for approximate k-nearest neighbors. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*. 1–5. https://doi.org/10.1109/DSIA.2017.8339084

[30] Yannis Kalantidis, Lyndon Kennedy, and Li-Jia Li. 2013. Getting the look: clothing recognition and segmentation for automatic product suggestions in everyday photos. In *ICMR*. 105–112.

[31] Carl Kingsford. [n.d.]. kd-Trees. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf. [Online; accessed 18-Oct-2019].

[32] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings to Document Distances. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 957–966.

[33] Yifan Lei, Qiang Huang, Mohan Kankanhalli, and Anthony K. H. Tung. 2020. Locality-Sensitive Hashing Scheme Based on Longest Circular Co-Substring. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2589–2599. https://doi.org/10.1145/3318464.3389778

[34] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2020. Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2020), 1475–1488. https://doi.org/10.1109/TKDE.2019.2909204

[35] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, and Chu-Song Chen. 2015. Deep learning of binary hash codes for fast image retrieval. In *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 27–35. https://doi.org/10.1109/CVPRW.2015.7301269

[36] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *PVLDB*. 950–961.

[37] Marius Muja and David G Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (Nov 2014), 2227–2240. https://doi.org/10.1109/TPAMI.2014.2321376

[38] Rafail Ostrovsky and Yuval Rabani. 2005. Low Distortion Embeddings for Edit Distance. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing (STOC '05)*. Association for Computing Machinery, New York, NY, USA, 218–224. https://doi.org/10.1145/1060590.1060623

[39] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. [n.d.]. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/.

[40] Lianyong Qi, Xuyun Zhang, Wanchun Dou, and Qiang Ni. 2017. A Distributed Locality-Sensitive Hashing-Based Approach for Cloud Service Recommendation From Multi-Source Data. *IEEE J. Sel. Areas Commun.* 35, 11 (2017), 2616–2624.

[41] Anand Rajaraman and Jeffrey David Ullman. 2011. *Mining of Massive Datasets*. Cambridge University Press.

[42] Sidney I Resnick. 2007. *Heavy-tail phenomena: probabilistic and statistical modeling*. Springer Science & Business Media.

[43] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *ICCV*. IEEE, 2564–2571.

[44] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. 2000. The earth mover's distance as a metric for image retrieval. *International journal of computer vision* 40, 2 (2000), 99–121.

[45] Venu Satuluri and Srinivasan Parthasarathy. 2012. Bayesian locality sensitive hashing for fast similarity search. *PVLDB* 5, 5 (2012), 430–441.

[46] C. Silpa-Anan and R. Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*. 1–8. https://doi.org/10.1109/CVPR.2008.4587638

[47] Sadhan Sood and Dmitri Loguinov. 2011. Probabilistic Near-Duplicate Detection Using Simhash. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 1117–1126. https://doi.org/10.1145/2063576.2063737

[48] Ellen Spertus, Mehran Sahami, and Orkut Buyukkokten. 2005. Evaluating Similarity Measures: A Large-Scale Study in the Orkut Social Network. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*. Association for Computing Machinery, New York, NY, USA, 678–684. https://doi.org/10.1145/1081870.1081956

[49] Yifang Sun. 2016. *Approximate similarity search in high dimensional spaces: solutions, evaluations and applications*. Ph.D. Dissertation. University of New South Wales, Sydney, Australia. http://handle.unsw.edu.au/1959.4/56970

[50] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and Efficiency in High Dimensional Nearest Neighbor Search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 563–576. https://doi.org/10.1145/1559845.1559905

[51] Santosh Vempala. 2012. Randomly-oriented k-d Trees Adapt to Intrinsic Dimension. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*. 48–57. https://doi.org/10.4230/LIPIcs.FSTTCS.2012.48

[52] Xuerui Wang, Andrew McCallum, and Xing Wei. 2007. Topical n-grams: Phrase and topic discovery, with an application to information retrieval. In *ICDM*. IEEE, 697–702.

[53] Simon Winder, Matt Brown, Noah Snavely, Steven Seitz, and Richard Szeliski. [n.d.]. Trevi: Local Image Descriptors Data. http://phototour.cs.washington.edu/patches/default.htm.

[54] Jia Xu, Zhenjie Zhang, Anthony KH Tung, and Ge Yu. 2012. Efficient and effective similarity search over probabilistic data based on earth mover's distance. *The VLDB Journal* 21, 4 (2012), 535–559.

[55] LeCun Yann, Cortes Corinna, and J.C. Burges Christopher. [n.d.]. THE MNIST DATABASE of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[56] Yi Yu, Michel Crucianu, Vincent Oria, and Ernesto Damiani. 2010. Combining multi-probe histogram and order-statistics based LSH for scalable audio content retrieval. In *ACM MM*. 381–390. https://doi.org/10.1145/1873951.1874004

[57] Haoyu Zhang. [n.d.]. String Datasets. https://iu.box.com/s/x7hg7uxj7xmmcdvc62k7iux9txtt9doi.

[58] Haoyu Zhang and Qin Zhang. 2017. EmbedJoin: Efficient Edit Similarity Joins via Embeddings. In *SIGKDD*. ACM, 585–594. https://doi.org/10.1145/3097983.3098003

[59] Bolong Zheng, Xi Zhao, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S. Jensen. 2020. PM-LSH: A Fast and Accurate LSH Framework for High-Dimensional Approximate NN Search. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 643–655. https://doi.org/10.14778/3377369.3377374

[60] Yuxin Zheng, Qi Guo, Anthony K.H. Tung, and Sai Wu. 2016. LazyLSH: Approximate Nearest Neighbor Search for Multiple Distance Functions with a Single Index. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2023–2037. https://doi.org/10.1145/2882903.2882930