# SRS: Solving $c$-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index

Yifang Sun† Wei Wang† Jianbin Qin† Ying Zhang‡ Xuemin Lin†
†University of New South Wales, Australia ‡University of Technology Sydney, Australia
{yifangs, weiw, jqin, lxue}@cse.unsw.edu.au Ying.Zhang@uts.edu.au

## ABSTRACT

Nearest neighbor searches in high-dimensional space have many important applications in domains such as data mining, and multimedia databases. The problem is challenging due to the phenomenon called "curse of dimensionality". An alternative solution is to consider algorithms that returns a $c$-approximate nearest neighbor ($c$-ANN) with guaranteed probabilities. Locality Sensitive Hashing (LSH) is among the most widely adopted method, and it achieves high efficiency both in theory and practice. However, it is known to require an extremely high amount of space for indexing, hence limiting its scalability.

In this paper, we propose several surprisingly simple methods to answer $c$-ANN queries with theoretical guarantees requiring only a *single tiny* index. Our methods are highly flexible and support a variety of functionalities, such as finding the exact nearest neighbor with any given probability. In the experiment, our methods demonstrate superior performance against the state-of-the-art LSH-based methods, and scale up well to 1 billion high-dimensional points on a single commodity PC.

## 1. INTRODUCTION

Given a $d$-dimensional query point $q$ in a Euclidean space, a nearest neighbor (NN) query returns the point $o$ in a dataset $D$ such that the Euclidean distance between $q$ and $o$ is the minimum. It is a fundamental problem and has wide applications in many domain areas, such as computational geometry, data mining, information retrieval, multimedia databases, and data compression.

While efficient solutions exist for NN queries in low dimensional space, they are challenging in high-dimensional space. This is mainly due to the phenomenon of "curse of dimensionality", where indexing-based methods are outperformed by the brute-force linear scan method [45].

The curse of dimensionality can be largely mitigated by allowing a small amount of errors. This is the $c$-ANN query, which returns a point $o'$, such that its distance to $q$ is no more than $c$ times the distance of the nearest point. Locality sensitive hashing (LSH) [21] is a widely adopted method that answers such a query in sublinear time with constant probability. It also enjoys great success in practice, due to its excellent performance and ease of implementation.

A major limitation of LSH-based methods is that their indexes are typically very large. The original LSH method requires indexing space superlinear in the number of data points [14], and the index typically includes hundreds or thousands of hash tables in practice. Recently, several LSH-based variants, such as LSB-forest [42] and C2LSH [17], are proposed to build a smaller sized index size *without* losing the performance guarantees of the original LSH method. However, their index sizes are still superlinear in the number of points; this prevents their usage for large datasets.[1]

In this paper, we propose a surprisingly simple algorithm to solve the $c$-ANN problem with (1) rigorous theoretical guarantees, (2) requiring an extremely small index, and (3) delivering superior empirical performance to existing methods. Our methods are based on projecting high-dimensional points into an appropriately chosen low-dimensional space via 2-stable random projections. The key observation is that the inter-point distance in the projected space (called *projected distance*) over that in the original high-dimensional space follows a known distribution, which has a sharp concentration bound. Therefore, given *any* threshold on the projected distance, and for any point $o$, we can compute *exactly* the probability that $o$'s projected distance is within the threshold. This observation gives rise to our basic algorithm, which reduces a $d$-dimensional $c$-ANN query to a $m$-dimensional exact $k$-NN query equipped with some early-termination test. It can be shown that the resulting algorithm answers a $c$-ANN query with at least constant probability using $\gamma_1 \cdot n$ I/Os in the worst case, using an index of $\gamma_2 \cdot n$ pages, where $\gamma_1 \ll 1$ and $\gamma_2 \ll 1$ are two small constants. For example, in a typical setting, we have $\gamma_1 = 0.0083$ and $\gamma_2 = 0.0059$.[2] We also derive several variants of the basic algorithm, which offer various new functionalities (as described in Section 5.3.2 and shown in Table 1. Our proposed algorithms can also be extended to support returning the $c$-approximate $k$-nearest neighbors (i.e., $c$-$k$-ANN queries) with conditional guarantees.

The contributions of the paper are summarized below:

[1]Note that the data size is always $O(n \cdot d)$.
[2]As will be pointed out in Section 3, existing methods using space linear in the number of data points ($n$) typically cannot outperform the linear scan method, whose query complexity is $0.25n$ in this example.

Table 1: Comparison of Algorithms

| Algorithm | Success Probability | Index Size | Query Cost | Constraint on Approximation Ratio $c$ | Comment |
|---|---|---|---|---|---|
| LSB-forest [42] | $1/2 - 1/e$ | $O((dn/B)^{1.5})$ | $O((dn/B)^{0.5} \log_B n)$ | $c \geq 4$ and must be the square of an integer | Index size superlinear in both $n$ and $d$ |
| C2LSH [17] | $1/2 - 1/e$ | $O((n \log n)/B)$ (or $O(n/B)$) | $O((n \log n)/B)$ (or $O(n/B)$) | $c \geq 4$ and must be the square of an integer | $\beta = O(1)$ (or $\beta = O(1/n)$) |
| SRS-12$(T, c, p_\tau)$ | $1/2 - 1/e$ | $\gamma_1 n$ (worst case), $\gamma_1 \ll 1$ | $\gamma_2 n$ (worst case), $\gamma_2 \ll 1$ | $c \geq 1$ | Fastest |
| SRS-1$(T, c)$ | $1/2 - 1/e$ | | | | Better quality |
| SRS-12$^+(T, c', p_\tau)$ | $1/2 - 1/e$ | | | | Tunable quality |
| SRS-2$(c, p)$ | $p$ | | $O(n/B)$ | | Support finding NN |

- We proposed a novel solution to answer $c$-ANN queries for high-dimensional points. Our methods use a tiny amount of space for the index, and answer the query with bounded I/O costs and guaranteed success probability. They also have excellent empirical performance and allow the user to fine-tune the tradeoff between query time and result quality. We present the technical details in Sections 4 and 5 and theoretical analysis in Section 6.
- We obtained several new results with our proposal: (i) Our SRS-12 algorithm is the first $c$-ANN query processing algorithm with guarantees, using linear-sized index and working with any approximation ratio. Previous methods use superlinear index size and/or cannot work with non-integer $c$ or $c < 2$ (See Table 1 for a comparison). (ii) Our SRS-2 algorithm returns exact nearest neighbor with any given probability, which is not possible for LSH-based methods since they require $c > 1$. (iii) There is no approximation ratio guarantee for $c$-$k$-ANN query results by existing LSH-based methods for $k > 1$, while our SRS-12 provides such a guarantee under certain conditions.
- We performed an extensive performance evaluation against the state-of-the-art LSH-based methods in the external memory setting. In particular, we used datasets with 8 million and 1 billion points, which is much larger than those previous reported. Experiments on these large datasets not only demonstrate the need for linear-sized index, but also debunk the myth that LSH-based method requires too much indexing space and hence cannot scale to large datasets. Sections 8 reports the experiment results and analyses.

Our methods may have several implications in theory and in practice. For one thing, using our methods and under typical settings, high-dimensional $c$-ANN problems in arbitrarily high dimensional space can be reduced to exact $k$-NN queries in a low-dimensional space with no more than 10 dimensions. This may spur interest in researching efficient indexing methods specifically for such a dimensionality-limited space. For another, our methods use standard multidimensional index (such as the $R$-tree) for indexing and query processing. Therefore, our methods are easily implemented standalone, or be readily integrated into database systems supporting multidimensional indexes [25].

## 2. PRELIMINARIES

We first give problem definitions, then introduce some statistical distributions, and finally the notations used in the rest of the paper.

### 2.1 Problem Definition

In this paper, we consider a dataset $D$ which contains $n$ points in a $d$ dimensional Euclidean space $\Re^d$. We are particularly interested in the high-dimensional case where $d$ is a fairly large number (e.g., $d \geq 50$). The coordinate value of $o$ on the $i$-th dimension is denoted as $o[i]$. The Euclidean distance between two points, $dist(o_1, o_2)$, is defined as $\sqrt{\sum_{i=1}^{d} (o_1[i] - o_2[i])^2}$. Throughout the rest of the paper, we are concerned with a point's distance with respect to a query point $q$, so we use the shorthand notation $dist(o) := dist(o, q)$, and refer to it as *distance of point o*.

For simplicity, we assume there is no draw in terms of distances. This makes the following definitions unique and deterministic. The results in this paper still hold without such assumption by a straight-forward extension.

Given a query point $q$, the *nearest neighbor* (NN) of $q$ (denoted as $o^*$) is the point in $D$ that has the smallest distance. We can generalize the concept to the $i$-th nearest neighbor (denoted as $o_i^*$). A *$k$ nearest neighbor query*, or $k$-NN query, returns the ordered set of $\{ o_1^*, o_2^*, \ldots, o_k^* \}$.

Given an approximation ratio $c > 1$, a *$c$-approximate nearest neighbor query*, or $c$-ANN query, returns a point $o \in D$, such that $dist(o) \leq c \cdot dist(o^*)$; such a point $o$ is also called a $c$-ANN point.[3] Similarly, a *$c$-approximate $k$-nearest neighbor query*, or $c$-$k$-ANN query, returns $k$ points $o_i \in D$ $(1 \leq i \leq k)$, such that $dist(o_i) \leq c \cdot dist(o_i^*)$.

In this paper, we consider *probabilistic* algorithms to correctly answer $c$-ANN and $c$-$k$-ANN queries with at least a constant probability, which is called the *success probability.* Using the standard *boosting* trick, we can increase the success probability of the algorithm to $1 - \delta$ by repeating the algorithm $O(\log \delta)$ times.

We focus on the external memory setting, where both the dataset and index reside on external memory. The page size is $B$ machine words. We follow the convention that every integer or real number is represented by one machine word.

### 2.2 2-Stable Distribution and $\chi^2$ Distribution

$p$-stable distribution is defined as follows: for any $n$ real numbers $v_1, \ldots, v_n$ and independently and identically distributed (i.i.d.) random variables $X_1, \ldots, X_n$ following the $p$-stable distribution, $\sum_i v_i X_i$ has the same distribution as $\left(\sum_{i=1}^{n} v_i^p\right)^{1/p} \cdot X$, where $X$ is a random variable with the $p$-stable distribution [20]. $p$-stable distribution exists for $p \in (0, 2]$, and when $p = 2$, it is the normal distribution.

[3]We distinguish $k$-NN, which refers to the $k$ nearest points, and $c$-ANN, which refers to a single point whose approximation ratio is within $c$.

Let $f(o) := \vec{v} \cdot \vec{o}$, where each entry of $\vec{v}$ is i.i.d. random variable following the standard normal distribution $\mathcal{N}(0,1)$, then we have the following Lemma (stated as Fact 1 in [35]).

LEMMA 1. *For any $o_1, o_2 \in \Re^d$, $f(o_1) - f(o_2)$ is distributed according to the normal distribution $\mathcal{N}(0, dist^2(o_1, o_2))$.*

If $X_1, \ldots, X_m$ are i.i.d. random variables following $\mathcal{N}(0,1)$, let $S^2 = \sum_{i=1}^{m} X_i^2$, then $S^2$ follows the $\chi^2$ distribution with $m$ degrees of freedom *by definition*, denoted as $S^2 \sim \chi^2(m)$.

## 2.3 Notations

We summarize the notations used in the paper in Table 2, and the parameters used in our algorithms in Table 3.

**Table 2: Notations**

| Symbol | Explanation |
|---|---|
| $n$ | number of points in $D$ |
| $d$ | dimensionality of each point |
| $q$ | the query point |
| $o^*$, $o_i^*$ | the first and $i$-th nearest point in $D$ to $q$ |
| $dist(o_i, o_j)$ | the $l_2$ distance between $o_i$ and $o_j$ |
| $dist(o)$ | the $l_2$ distance between $o$ and $q$ |
| $\pi_m(o)$ | an $m$-dimensional signature of $o$, i.e., $\pi_m(o) := \langle f_1(o), f_2(o), \ldots, f_m(o) \rangle$, where $f_i(o)$ is the $i$-th projected value |
| $\Delta_m(o)$ | the $l_2$ distance between $o$ and $q$'s signatures, i.e., $\Delta_m(o) := dist(\pi_m(o), \pi_m(q))$ |
| $\chi^2(m)$ | $\chi^2$ distribution with $m$ degrees of freedom |
| $\Psi_m(x)$, $\Psi_m^{-1}(p)$ | cumulative distribution function and its inverse of $\chi^2(m)$. $\Psi_m^{-1}(\Psi_m(x)) = x$ |
| $o_{\min}$ | the point with the minimum distance (among points accessed by Algorithm 1) |

**Table 3: Parameters**

| Role | Symbol | Explanation |
|---|---|---|
| input | $n$ | number of data points |
| input | $T$ | maximum number of data points to be accessed |
| input | $c$ | approximation ratio |
| input | $c'$ | desirable approximation ratio (optional) |
| derived | $m$ | the number of 2-stable random projections needed, given $n$, $T$ and $c$ |
| derived | $T'$ | $T' \leq T$ is the maximum number of data points to be accessed by our algorithm |
| derived | $p'_\tau$ | threshold of early-termination condition |

## 3. RELATED WORKS

There is a vast amount of related literature given that (approximate) nearest neighbor search is a classic and fundamental problem. We will briefly survey most related areas below. We refer readers to the excellent book [37] for a comprehensive coverage of multidimensional indexes.

## 3.1 Nearest Neighbor Search

The exact nearest neighbor (NN) problem is also known as the *post office problem* in computational geometry and the distance measure is typically Euclidean distance. Voronoi diagrams gives the optimal solution for $d = 2$ with $O(n)$ space and $O(\log n)$ query time. It remains an open problem to find a solution for $d = 3$ with linear space and near logarithmic query time. The best result is [32], which gives a data structure with $O(d^5 \log n)$ query time using $O(n^{2d+\delta})$ space. The exact NN problem is conjectured to be hard, though the results on the lower bound complexity are still weak [10, 7]. [36] showed that the average query time with an optimal metric indexing scheme is superpolynomial in dimensionality $d$.

Given the hardness of solving the exact NN problem, and the fact that approximate NNs are also desirable in many applications, the current focus is on efficient solutions for the (1+$\epsilon$)-approximate NN problem. We distinguish two cases. The easy case is when $\epsilon$ is sufficiently large. [11] gives a randomized algorithm with $O(d^2 n)$ space and $O(d^2 \log n)$ query time for $O(d^{3/2})$-approximate NN. The other case when $\epsilon$ is small is much harder. Most early works still require either space or query time exponential in $d$. E.g., [3] gives a scheme that can tune the trade-off between index space and query time, which results in an algorithm with $O(\log n + 1/\epsilon^{(d-1)/2})$ query time and $O(n \log(1/\epsilon))$ space, and another with $O(\log(n/\epsilon))$ query time and $O(n/\epsilon^{d-1} \log(1/\epsilon))$ space. Later, better results were obtained by using the idea of *probabilistic test*: Two points with distances $r$ and $(1+\epsilon)r$ to a query point have certain probability of having different test results. Random projections were used in [26, 28, 21], and $p$-stable random projections were used in [14], which results in algorithms whose space and time complexity is only polynomial in $d$. Recently, fast JL transformation was used to answer the query in $O(d \log d + \epsilon^{-3} \log^2 n)$ time with $O(n^{\max(2, \epsilon^{-2})})$ space [1]. Among them, the Locality Sensitive Hashing (LSH) is the most widely used due to its excellent theoretical guarantees (it is the most efficient with sub-quadratic index space [2]) and empirical performance. More discussions appear in the next section.

To scale up to very large datasets, sub-quadratic space complexity is still not acceptable; we have to settle with methods that use space linear in $n$ and $d$. There are only few methods that we are aware of. The brute-force linear scan algorithm has a trivial query time $O(dn)$. [47]'s query complexity is $O(dn^{1-\varepsilon_d})$, but $\varepsilon_d$ goes to 0 rapidly with $d$. [4]'s query time is $O(1/\epsilon^d \log n)$. Therefore, given $n$ and a sufficiently large $d$, the brute-force method will practically outperform other methods. In contrast, we will show that the space and time complexities of our methods are both linear in $n$ (independent of $d$), and the constant factor is very small. Furthermore, we can perform exact NN search probabilistically using a fraction of I/Os required by the linear scan method (Section 8.6).

## 3.2 Locality Sensitive Hashing

The LSH technique is firstly introduced by Indyk and Motwani [21]. It employs locality-sensitive hash functions as the probabilistic test. LSH functions for some commonly used metrics are known, e.g., minhash for Jaccard [8], simhash for arccos [12], and $p$-stable random projection with quantization for $l_p$ norms for $p \in (0, 2]$ [14].

A key theme in theoretical LSH research is to improve the bounds. Let $c = 1 + \epsilon$. LSH methods return a $c$-ANN point in time $O(dn^{\rho + o(1)})$ with space complexity of $O(n^{1+\rho+o(1)} + nd)$. The initial scheme [14] has $\rho = 1/c + o_c(1)$.[33] gives the optimal lower bounds of $\rho$ as $1/c - O_d(1)$. Most recently, [2] obtains a scheme with $\rho \leq 7/(8c) + O(1/c^{1.5}) + o_c(1)$ by using two-level LSH.

The main drawback of LSH is that it has to build multiple indexes with different distance thresholds, resulting in indexes of enormous sizes. There are two major directions to address this weakness: one is to adapt LSH to external memory, which will be discussed in the next section, and the other is to trade query time with space by either posing more queries to the index of reduced sizes [35], or accessing more buckets [31].

There are many other works that improve or adapt LSH in various aspects. For example, using a prior [38], using sampled data [15], on distributed computing platforms [5, 41], and taking advantages from specific hardware [39, 34]. Parameter tuning for LSH is important, and this is discussed in detail in [40].

#### 3.2.1 *LSH for External Memory*

There are a few works focusing on adapting LSH to extern memory. [18] and [48] are two methods based on LSH and multi-probe LSH, respectively, but do not have any theoretical guarantee [42]. We mainly focus on two state-of-the-art methods, LSB-forest [42] and C2LSH [17], which achieve both high quality and efficiency *without* losing the theoretical guarantee, though with different ideas. LSB-forest adapts to the distance of the nearest point and hence only needs to build one index that works for all NN distances. This can be deemed as generalizing the reduction method in [12] from $\ell_1$ to $\ell_2$. Multiple such indexes (each called an LSB-tree) need to be built to return a $c$-ANN point with constant probability. C2LSH has the novel idea of not combining signatures from individual hash functions, hence can fully utilize the information in each projection. It performs collision counting with increasing granularities to determine the candidate points.

## 3.3 Transformation-based Approaches

A closely related area is related to small distortion embedding of Euclidean norm, with the seminal work in [24] known as the *Johnson-Lindenstrauss Lemma* (JL Lemma). We refer readers to [1] for a recent survey.

While JL transform is data-independent, there also exist other *data-dependent* embeddings. PCA and compressed sensing assume or exploit certain distributional characteristics of the data. Other approaches, such as Spectral Hashing [46], map points into a Hamming cube which maximally preserves some desirable distances between points.

## 3.4 Other Methods for NN Queries

There are efficient solutions that assume or exploit the low intrinsic dimensionalities of the data to answer (approximate) NN queries. Representative methods include those based on navigating nets [27], cover tree [6], and recently RBC [9]. There are also heuristic methods that strive to answer NN queries approximately with good empirical performance, albeit having no guarantees. Representative methods include Spill Tree [30], SASH [19], and iDistance [22], and product quantization [23].

## 4. OVERVIEW

In this section, we give an informal but intuitive explanation of our method, followed by an overview of our methods.

Our method is based on projecting points from the original $d$-dimensional space to a $m$-dimensional space. We refer to the former as the **original space**, and the latter as the **projected space**. In both spaces, we are concerned with Euclidean distances, indicated by different notations ($dist(o)$ in the original space called simply as *distance*, and $\Delta_m(o)$ in the projected space called *projected distance*). See Figure 1 for an illustration.

$$\begin{tikzcd}
d\textbf{-dimension:} & q \arrow[d, "{f_1, f_2, \ldots, f_m}"'] & & o \arrow[ll, dashed, "dist(o)"'] \arrow[dr, "{f_1, f_2, \ldots, f_m}"] & \\
m\textbf{-dimension:} & \pi_m(q) & & & \pi_m(o) \arrow[llll, dashed, "\Delta_m(o)"']
\end{tikzcd}$$

**Figure 1: Two Distances: $dist(o)$ and $\Delta_m(o)$. $\frac{\Delta_m^2(o)}{dist^2(o)}$ follows the $\chi^2(m)$ distance whose mean is $m$.**

Based on the property of 2-stable random projections, our key observation is that for any point $o$, $\frac{\Delta_m^2(o)}{dist^2(o)}$ (which can be intuitively interpreted as distance distortions due to the projection) follows the standard $\chi^2(m)$ distribution (Lemma 2), which has sharp concentration bounds around the mean (which is $m$).

This observation is exploited in two very different ways in our methods. (i) Firstly, given two points $o_1$ and $o_2$ whose distances to the query are $r$ and $c \cdot r$, respectively, it is less likely that the projected distance of $o_2$ is smaller than $o_1$. We can compute the above probability *exactly* and then upper bound the total number of such "erroneous" points in the projected space probabilistically. Therefore, a $k$-NN search on the projected space with a carefully chosen $k$ will guarantee us a $c$-ANN point with the desired probability. (ii) Secondly, given a point $o_{\min}$ that has the minimum distance to the query among the first $i$ points according to the projected distance, we want to determine whether it is a $c$-ANN point. This is related to the probability that there exists a point that is $c$ times closer to the query than $o_{\min}$, yet its projected distance is larger than any of the $i$ points accessed. This leads to a novel and effective *early-termination condition.*

The above ideas lead to our basic method to answer a $c$-ANN query, which conceptually consists of two steps:

- Obtain an ordered set of candidates by issuing a $k$-NN query with $k = T'$ (also called a $T'$-NN query in the following) on the $m$-dimensional projections of data points;
- Examine the distance of these candidate points in order and return the point with the smallest distance so far if it satisfies the early-termination test or the algorithm has exhausted the $T'$ points.

With carefully chosen parameters ($m$ and $T'$), we can show that the returned point is a $c$-ANN point with a constant probability. There are several variants of the basic algorithm that have distinct features; for example, it can be tuned to return exact NN with constant probabilities, as well as obtaining a different trade-off between I/O cost and approximation ratio. Finally, the algorithm can easily be extended to support $c$-$k$-ANN queries with a novel conditional quality guarantee.

## 5. OUR METHODS

We describe our query processing methods in this section. We leave the detailed proof and analyses to Section 6.

## 5.1 Computing Internal Parameters

Our method is designed to use a tiny amount of space to index high-dimensional data. To this end, it first needs to compute the best internal parameter setting that achieves the goal specified by the user, before the indexing and query processing can be performed. Our methods has a default success probability guarantee of $p_\tau := 1/2 - 1/e \approx 0.1321$. Although it can be changed (hence it appears as a parameter of our algorithms), we shall treat it as a constant to simplify the presentation.

There are *three* input parameters, (a) $n$ is the number of points of the data points, (b) $c$ is the approximation ratio for (future) queries, and (c) $T$ is the maximum number of data points the query processing algorithm can access in the worst case. The worst case I/O cost is also bounded by a linear function of $T$ (See Section 5.3.1.1). The default value we used in our experimental evaluation is $T = 0.005 \cdot n$, which will result in an index whose size is linear in $n$.

Given the input parameters $n$, $c$, and $T$, we compute our internal parameters: (a) $m$ is the number of 2-stable random projections we use, and this is also the dimensionality of the projected space. (b) $T' \le T$ is an improved worst case guarantee of the number of points accessed by our algorithms, resulting from our automatic parameter optimization procedure. (c) $p'_\tau > p_\tau$ is the threshold used in the early-termination test. The detailed introduction to the computation of $m$, $T'$ and $p'_\tau$ values is deferred to Section 6.2, yet no prior knowledge is needed to implement it following the pseudo-code given in Algorithm 6. For example, with $T = 0.005 \cdot n$ and $c = 4$, we compute $m = 6$, $T' = 0.00242 \cdot n$ and $p'_\tau = 0.1809$.

## 5.2 Indexing

The indexing process essentially projects each point from $d$-dimensional space into an $m$-dimensional space (with $m$ computed from the previous section), and indexes the $n$ $m$-dimensional points in a multi-dimensional index that supports *incremental k-NN search*.

To perform the projection, we first generate $m$ 2-stable random projection vector $v_i$, where each of their entries are randomly and independently sampled from $\mathcal{N}(0, 1)$. Given a point $o$, we compute its projection $\pi_m(o)$ as $\pi_m(o) = \langle f_1(o), f_2(o), \ldots, f_m(o)\rangle$, where $f_i(o) = \vec{v_i} \cdot \vec{o}$.

We then use a multi-dimensional index to map the $m$-dimensional projections to their corresponding point IDs. The only requirement we need is that the index supports incremental $k$-NN search, i.e., the $(k+1)$-th nearest data point with respect to a query point can be computed efficiently after it returns the $k$-th nearest data point. In this paper, we simply choose $R$-tree as the index. In the following, we refer to such an index indexing $n$ $m$-dimensional projections as an **SRS-tree**.

## 5.3 Algorithms for $c$-ANN Queries

We first introduce the basic algorithm, named SRS-12, for answering $c$-ANN queries, followed by its variants.[4]

### 5.3.1 The Basic Algorithm

We give the pseudo-code of SRS-12 in Algorithm 1, which guarantees to return a $c$-ANN point with constant probability of $p_\tau$. The algorithm takes as the $T'$ and $p'_\tau$ precomputed from input parameters as discussed in Section 5.1. It then calls the interal function incSearch. Shortly we will see other variants of the algorithm by passing different parameter values to incSearch.

Now consider the incSearch function in Algorithm 2. Given the query $q$, we firstly project $q$ to the $m$-dimensional space

[4]The digital suffix of the algorithm names indicates if the corresponding stopping conditions are used. Therefore, SRS-12 means both conditions are used.

**Algorithm 1**: SRS-12$(T, c, p_\tau)$

```
1  $(T', p'_\tau) \leftarrow$ the values precomputed from $(n, c, T)$;
2  return incSearch$(T', c, p'_\tau)$;
```

**Algorithm 2**: incSearch$(maxPts, c, threshold)$

```
   Input: $maxPts$ is the maximum number of nearest points
          to access in the projected space, $c$ is the (desired)
          approximation ratio, and $threshold$ is the threshold of
          early-termination condition.
   Output: Returns a $c$-ANN point with probability at least $p_\tau$.
1  Compute $\pi_m(q)$ using the same $m$ 2-stable random projection
   vectors as those used in indexing;
2  $i \leftarrow 1$;
3  $o_{\min} \leftarrow$ nil;                /* assume $dist(\textbf{nil}) = \infty$ */;
4  while $i \le maxPtrs$ do
       /* get the $i$-NN point in $m$-dimensional space */
5      $cand \leftarrow$ ID of the $i$-th nearest neighbor of $\pi_m(q)$;
       /* early-termination test */
6      if $\Psi_m\left(\frac{c^2 \cdot \Delta_m^2(cand)}{dist^2(o_{\min})}\right) > threshold$ then
7          return $o_{\min}$;
       /* Update $o_{\min}$ and $i$ */
8      if $dist(cand) \le dist(o_{\min})$ then
9          $o_{\min} \leftarrow cand$;
           /* redo the test since $o_{\min}$ has changed */
10         if $\Psi_m\left(\frac{c^2 \cdot \Delta_m^2(cand)}{dist^2(o_{\min})}\right) > threshold$ then
11             return $o_{\min}$;
12     $i \leftarrow i + 1$;
13 return $o_{\min}$;
```

using the same $m$ 2-stable random projection vectors $v_i$, and calculate its projection $\pi_m(q)$. Then we perform an *exact* $k$-NN query centered at $\pi_m(q)$ *incrementally* until we've accessed *maxPts* points. In each iteration, we obtain the $i$-th nearest projections into the variable *cand* (Line 5). We compute the distance of *cand* by fetching its coordinates from the data file. We also maintain the data point, $o_{\min}$, that has the smallest distance (in the $d$-dimensional space) to the query so far. For each newly found *cand* point, we first check if *early-termination condition* for the current $o_{\min}$ point is satisfied (Line 6). If the condition is satisfied, we simply return $o_{\min}$ as the answer. Otherwise, we compute the distance of *cand* to the query (in the $d$ dimensional space), and update $o_{\min}$ accordingly (Lines 8–9). If $o_{\min}$ has changed, we perform the early-termination test again. By potentially performing the same test twice inside a loop makes sure that we do not waste additional I/Os and stop as early as possible.

We note that the algorithm has *two stopping conditions*: (1) *the normal termination condition* where it has accessed *maxPts* data points, and (2) the *early-termination condition* on Lines 6 and 10. We will show that the algorithm stops due to either stopping condition will return a $c$-ANN point with probability at least $p_\tau$ in Sections 6.1 and 6.3, respectively. These lead to the main theorem about the SRS-12 algorithm (Theorem 1). The proof of success probability is given in Section 6.4, and the cost part is given in Section 5.3.1.1.

THEOREM 1. *Algorithm 1 returns a $c$-ANN point with probability at least $p_\tau = 1/2 - 1/e$. More specifically, if the algorithm stops due to the early-termination condition, it returns a $c$-ANN point with probability at least $p'_\tau$. It processes a query* *using $\gamma_1 \cdot n$ I/Os in the worst case, with an index of size $\gamma_2 \cdot n$ pages, where $\gamma_1 \ll 1$ and $\gamma_2 \ll 1$ are two constants.*

#### 5.3.1.1 Index Space and Query Cost Analyses.

First, we analyze the space cost. In our SRS-tree, we index the $m$ projections of $n$ data points. The fan-out of the $R$-tree internal nodes is $f = \frac{B}{2m+1}$. Hence the total size of the $R$-tree is $\frac{nm}{B-2m-1}$ pages.As will be shown in Corollary 3, by choosing $T = O(n)$, we have $m = O(1)$. Therefore, the space complexity of our index is $O(n)$ disk pages, and the constant hidden is a small value.

Next, we analyze the I/O cost for the query. In the worst case, the algorithm stops by the normal termination condition. Given that $maxPts \leq T$, the cost is upper bounded by the sum of (1) executing a $T$-NN query on the SRS-tree, and (2) the cost of fetching $T$ points for distance computation. The cost of latter is at most $T$ (assuming $d \leq B$). For the former cost, while there are cost models to predict the $T$-NN search cost [43], we opt to use a crude worst-case estimate here: we assume all the $R$-tree nodes are accessed. Therefore, the total I/O costs is at most $T + \frac{nm}{B-2m-1}$. With $T = O(n)$ and $m = O(1)$, the worst case I/O cost is $O(n)$, and the constant hidden is a small value.

We also note that the above is the worst-case analysis. In addition, the early-termination condition, if used, can be shown to be both theoretically and empirically effective in stopping the execution well before $T$ points are examined.

For example, consider the typical setting where $B = 1024$, $d = 256$, $c = 4$, $m = 6$, $T = 0.00242 \cdot n$, our index is $0.0059n$ pages, and the query cost is at most $0.0084n$.

### 5.3.2 Three Variants of SRS-12

The SRS-12 has several interesting variants that have different features or address different but related problems.

In the **first variant**, called SRS-1, we do not use the early-termination tests in Lines 6–7 and Lines 10–11 of Algorithm 2. For presentation simplicity, we achieve this by passing a value greater than 1 to the *threshold* parameter (see pseudo-code in Algorithm 3). Obviously, the algorithm has the same success probability guarantee as Algorithm 1 due to Theorem 1. The average I/O cost will be higher (but still bounded by Theorem 1), but the approximation ratio will be better, as it accesses more points.

**Algorithm 3**: SRS-1$(T, c)$

```
1 threshold ← 1.6180;            /* any number > 1 will do */;
2 return incSearch($T'$, c, threshold);
```

In the **second variant**, called SRS-2, we allow the algorithm to potentially examine all the points. This can be implemented as passing the $n$ as the *maxPts* parameter into the incSearch function. The algorithm can also take any success probability $p$, and just pass it on to the *threshold* parameter (see Algorithm 4). Theorem 2 shows that it returns a $c$-ANN with probability at least $p$.

This algorithm is flexible in its parameter settings in that it works with any $p \in [0, 1)$ and $c \geq 1$. Hence, a perhaps surprising by-product is that we can find the nearest neighbor point (NN) with constant probability by setting $c = 1$ , albeit the expected I/O cost is $\Psi_m\left(\Psi_m^{-1}(p)/c^2\right) \cdot n + \frac{nm}{B-2m-1}$ in the worst case. The actual number according to our experimental evaluation is much lower. For example, it uses about 15% of the I/Os used by the linear scan method to find the NN with about 71% probability in Section 8.6. Note that LSH-based methods *cannot* handle the case of $c = 1$.

**Algorithm 4**: SRS-2$(c, p)$

```
1 return incSearch($n, c, p$);
```

THEOREM 2. *Algorithm 4 called with any $c \geq 1$ and $p \in [0, 1)$ returns a c-ANN point with probability at least p.*

The **third variant** is to pass a value $c' \in [1, c)$ as the $c$ parameter when invoking the incSearch function. This instructs the algorithm to look for "better" quality approximate nearest neighbors. If the incSearch function stops due to the early-termination test with $c'$, then we can assert the returned point is a $c'$-ANN with probability at least $p'_\tau$; otherwise, we can only guarantee the approximation ratio to be no more than $c$ with probability $p_\tau$.This variant allows the user to have fine-granularity control over the result quality by paying additional query processing costs (i.e., I/Os). It has a similar spirit as any-time algorithms [44], and may be desirable in applications such as interactive sessions or query processing with a hard time or I/O limit.

**Algorithm 5**: SRS-12$^+(T, c', p_\tau)$

```
1 ($T', p'_\tau$) ← the values precomputed from ($n, c, T$);
2 return incSearch($T', c', p'_\tau$);
```

## 5.4 An Example

Consider the four points in 3-dimensional space in Table 4. Let the query point be $(0, 0, 0)$.

**Table 4: Example (Data Points Ordered Based on $\Delta_m^2(o_i)$, $\pi_m(q) = (0, 0)$)**

| | $o_2$ | $o_3$ | $o_1$ | $o_4$ |
|---|---|---|---|---|
| $o_i$ | (1, 1, 1) | (4, 2, 3) | (1, 0, 1) | (9, 2, 3) |
| $\pi_m(o_i)$ | (0.1, -0.2) | (0.4, 0.5) | (0.5, 0.5) | (2.5, 2.5) |
| $\Delta_m^2(o_i)$ | 0.05 | 0.41 | 0.50 | 12.50 |
| $dist^2(o_i)$ | 3 | 29 | 2 | 94 |

User A specifies $c = 2$ and $T = 3$. Assume that our parameter computation gives $m = 2$, $T' = T$ and $p'_\tau = 0.1809$, and we build the index with two 2-stable random projection vectors $(0.3, -0.4, 0.2)$ and $(0.4, -0.7, 0.1)$. As we compute $\pi_m(q) = (0, 0)$, the 3-NN points to $\pi_m(q)$ are $[o_2, o_3, o_1]$.

Algorithm 1 will check them in order, and check the early-termination condition.

- In the first iteration, $o_{\min}$ is **nil**, we initialize it to $o_2$ with the $dist^2(o_2) = 3$, and perform the early-termination test of it. Since $\Psi_m\left(2^2 \cdot 0.05/3\right) = 0.0328$, which is smaller than $p'_\tau$. So the algorithm continues.
- In the second iteration, we use the current projection distance to test $o_{\min}$. Since $\Psi_m\left(2^2 \cdot 0.41/3\right) = 0.2392$, which is larger than $p'_\tau$. So the algorithm stops and returns the current $o_{\min}$, i.e., $o_2$, as the answer. In this case, it is indeed a 2-ANN.

## 5.5 Answering $c$-$k$-ANN Queries with Partial Guarantees

Our method can also be easily extended to support $c$-$k$-ANN queries. The major changes to Algorithm 1 are:

- The *maxPts* parameter passed in to the incSearch function needs to be modified to $maxPts + k - 1$.
- In each iteration of the incSearch function (Algorithm 2), we need to maintain the $k$ points with the minimum distances to the query so far. Let the $k$-th such point be $o_{\min}^{(k)}$, then we use $o_{\min}^{(k)}$, instead of $o_{\min}$, in the early-termination tests.

Unlike existing algorithms that have *no* guarantee on the result quality, we can guarantee that we return $c$-$k$-ANN points under the condition specified in Theorem 3.

THEOREM 3. *If the modified version of Algorithm 1 (according to the above description) stops due to the early-termination condition, the returned $k$ results are the $c$-$k$-ANN points with probability at least $p_\tau$.*

## 6. THEORETICAL ANALYSIS

In this section, we illustrate the underlying reasons that our proposed algorithms can achieve constant success probabilities, and how internal parameters are computed. Note that all the probabilities are with respect to the random choices of projections.

### 6.1 Normal Stopping Condition of Algorithm 1

#### 6.1.1 *Probability Distribution Function of Projected Distances*

LEMMA 2. *$\frac{\Delta_m^2(o)}{dist^2(o)}$ follows the $\chi^2(m)$ distribution.*

PROOF. By definition, $\Delta_m^2(o) = \sum_{i=1}^{m}(f_i(o) - f_i(q))^2$. The lemma follows by a simple derivation based on Lemma 1. $\square$

COROLLARY 1. *For any $x \geq 0$ and $o$, we have* $\mathbf{Pr}\left[\Delta_m(o) \leq x\right] = \Psi_m\left(\frac{x^2}{dist^2(o)}\right)$.

Corollary 2 is the key to our methods. For any $r > 0, c > 1$, we call data points within distance $r$ from $q$ "near points", and those beyond distance $c \cdot r$ "far points". Then for any $\kappa > 0$, the corollary bounds the probability of near and far points whose projected distances to the query are within $\kappa \cdot r$.

COROLLARY 2. *Given approximation ratio $c > 1$ and distance threshold $r$ (in $d$-dimensional space), for any $\kappa \geq 0$, we define the following two events for any $o \in D$:*
- ***E1:** $\Delta_m(o) \leq \kappa \cdot r$, given $dist(o) \leq r$.*
- ***E2:** $\Delta_m(o) \leq \kappa \cdot r$, given $dist(o) > c \cdot r$.*

*Then we have*

$$\mathbf{Pr}\left[\mathbf{E1}\right] \geq \Psi_m\left(\kappa^2\right) \qquad \textit{and} \quad \mathbf{Pr}\left[\mathbf{E2}\right] < \Psi_m\left(\kappa^2/c^2\right)$$

Figure 2 gives an example. We show the probabilistic distribution function (pdf) of the projected distance of any near point (e.g., $a$), as well as that of any far point (e.g., $b$). With a carefully chosen $\kappa$, we make a cut at $\kappa \cdot r$, and the areas under the pdf of the two curves to the left of the cut-off line are the total probability mess, which are at least $\Psi_m\left(\kappa^2\right)$ and at most $\Psi_m\left(\kappa^2/c^2\right)$ according to Corollary 2.

#### 6.1.2 *Proof of the Normal Stopping Condition*

THEOREM 4. *Assume Algorithm 1 with $c > 1$ stops due to the normal condition, i.e., returns $o_{\min}$ whose distance is among the nearest $T$ point projections. With carefully chosen $m$, $o_{\min}$ is a $c$-ANN of $q$ with probability probability at least $1/2 - 1/e$.*

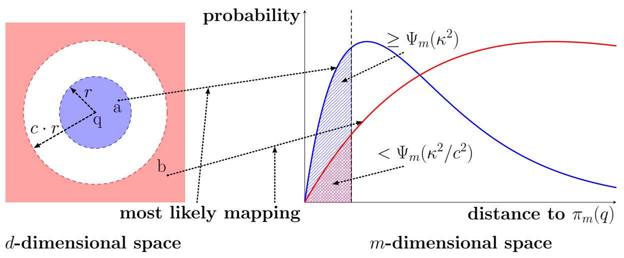


**Figure 2: Illustration of Corollary 2 (Best viewed in color)**

PROOF. Recall that $o^*$ refers to the nearest point. Let $r^* = dist(o^*)$. We choose a constant $\kappa$ whose value is to be specified at the end of this proof. According to Corollary 2, we have

$$\mathbf{Pr}\left[\mathbf{E1}\right] \geq \Psi_m\left(\kappa^2\right) \qquad \text{and} \quad \mathbf{Pr}\left[\mathbf{E2}\right] < \Psi_m\left(\kappa^2/c^2\right)$$

We define the additional event **E2a** as follows:
- **E2a:** $|\{\ \Delta_m(o) \leq \kappa \cdot r^* \mid dist(o) > c \cdot r^*\ \}| < T$

Since there are at most $n$ points whose distances are outside $c \cdot r^*$, using Markov's inequality, we have $\mathbf{Pr}\left[\mathbf{E2a}\right] > 1 - \frac{\Psi_m\left(\kappa^2/c^2\right) \cdot n}{T}$.

In the following, we only consider the case when both Events **E1** and **E2a** are true. This occurs with probability at least $p := \mathbf{Pr}\left[\mathbf{E1}\right] + \mathbf{Pr}\left[\mathbf{E2a}\right] - 1$. We will show at the end of the proof that, with appropriate choices of $m$ and $\kappa$, $p \geq 1/2 - 1/e$.

Let Algorithm 1 access the first $T$ points with respect to their projection distance to $\pi_m(q)$. Denote these points as $o_1, \ldots, o_T$. There are only two possible cases regarding the relationship between $\Delta_m(o_T)$ and $\kappa \cdot r^*$:
- **Case A:** $\Delta_m(o_T) > \kappa \cdot r^*$. Since **E1** is true, and given $dist(o^*) = r^*$, we know $\Delta_m(o^*) \leq \kappa \cdot r^*$. Based on the two inequalities above, we have $\Delta_m(o^*) < \Delta_m(o_T)$. This means Algorithm 1 must have accessed $o^*$, and hence $o_{\min} = o^*$ and the algorithm returns the NN of the query.
- **Case B:** $\Delta_m(o_T) \leq \kappa \cdot r^*$. Since **E2a** is true, and given above, we know that

  $$|\{\ \Delta_m(o) \leq \Delta_m(o_T) \mid dist(o) > c \cdot r^*\ \}| < T$$

  Since the algorithm has accessed exactly $T$ points $o_i$ ($1 \leq i \leq T$) with $\Delta_m(o_i) \leq \Delta_m(o_T)$, they must include at least one point $o_j$ which is from the region where $dist(o_j) \leq c \cdot r$. Hence $dist(o_{\min}) \leq dist(o_j) \leq c \cdot r$, and the algorithm returns a $c$-ANN point.

Therefore, in both cases, $o_{\min}$ is a $c$-ANN of $q$.

Finally, we only need to show that for any $c > 1$ and $1 \leq T \leq n$, we can guarantee $\mathbf{Pr}\left[\mathbf{E1}\right] + \mathbf{Pr}\left[\mathbf{E2a}\right] - 1 > 1/2 - 1/e$. Let $X$ be a random variable following the $\chi^2(m)$ distribution, we have the following tail inequalities [29]

$$\mathbf{Pr}\left[X - m \geq 2\sqrt{t \cdot m} + 2t\right] \leq \exp(-t)$$

$$\mathbf{Pr}\left[m - X \geq 2\sqrt{t \cdot m}\right] \leq \exp(-t).$$

Hence, for any $T$, let $\epsilon := \min(\frac{T}{2n}, 1/e)$, $t = \ln(1/\epsilon)$, $y_1 = m + 2\sqrt{t \cdot m} + 2t$, and $y_2 = m - 2\sqrt{t \cdot m}$. $\Psi_m(y_1) \geq 1 - \epsilon \geq 1 - 1/e$ and $\Psi_m(y_2) \leq \epsilon$. Now we only need to select a $m$, such that $\frac{y_1}{y_2} \leq c^2$. By solving the inequality, we can show that there is always a positive $m_0 = O(\log(n/T))$ such that the above inequality holds for any integer $m \geq m_0$, provided that $c > 1$. Hence, by choosing $m = \lceil m_o \rceil$ and $\kappa = \sqrt{y_2} \cdot c$, we have $\mathbf{Pr}[\mathbf{E1}] + \mathbf{Pr}[\mathbf{E2a}] - 1 > 1/2 - 1/e$. □

COROLLARY 3. *$m = \Theta(\log(n/T))$. Specifically, when $T = O(n)$, $m = O(1)$.*

## 6.2 Computing Parameter Values

In order to find a set of feasible parameters for our method, the first step is to find the smallest $m$ value that satisfies the two constraints (so that both events **E1** and **E2a** defined in the proof in Section 6.1 are true). It can be shown that $m$ is

$$\min\{\, i \in \mathbb{Z}^+ \mid \exists \kappa, \Psi_i(\kappa^2) \geq 1 - 1/e \wedge \Psi_i(\kappa^2/c^2) \leq T/(2n) \,\}.$$

We give the pseudo-code in Algorithm 6. We perform the search from $m = 1$ going upwards. For each $m$ value considered, we use the second constraint to find the $\kappa$ value and then test whether the first constraint is satisfied. We stop at the first $m$ value where both constraints are satisfied.

Consider the $m$ value chosen above, and it corresponds to a $\kappa$ value. It is likely that the first probability (i.e., $\Psi_m(\kappa^2)$) is more than $1 - 1/e$. A further optimization (Lines 6–9) is is to set $\kappa$ to $\Psi_m^{-1}(1 - 1/e)$, and this gives us a smaller value for $T$, and this is the $T'$ value we return and the value to be fed into Algorithm 1.

Finally we also need to compute $p'_\tau$ value, which can be shown to be

$$\min\{\, p \in [0,1] \mid p - \Psi_m(\Psi_m^{-1}(p)/c^2) \cdot n/T \geq 1/2 - 1/e \,\}.$$

The above optimization problem can be solved numerically (we use MATLAB), and is referred to as calcThreshold in Algorithm 6.

**Algorithm 6**: Calc-m$(T, c)$

```
1  $m \leftarrow 1, T' \leftarrow T$;
2  while true do
3      $\kappa^2 \leftarrow \Psi_m^{-1}(T/(2n)) \cdot c^2$;
4      $p_1 \leftarrow \Psi_m^{-1}(\kappa^2)$;
5      if $p_1 = 1 - 1/e$ then break;
6      else if $p_1 > 1 - 1/e$ then
7          $\kappa^2 \leftarrow \Psi_m^{-1}(1 - 1/e)$;
8          $T' \leftarrow n \cdot \Psi_m(\kappa^2/c^2)/2$;
9          break;
10     $m \leftarrow m + 1$;
11 $p'_\tau \leftarrow$ calcThreshold$(p_\tau)$;
12 return $(m, T', p'_\tau)$;
```

## 6.3 Early-Termination Condition

We will consider the success probability when Algorithm 1 exits because of the early-termination condition is true.

THEOREM 5. *If Algorithm 1 stops due to the early-termination condition, it returns a c-ANN with probability at least $p'_\tau$.*

PROOF. Let the early-termination condition becomes true at the $i$-th iteration, and $o_{\min}$ is the point with the minimum distance among all the points examined so far by the algorithm. We consider the relationship between $\Delta_m(o^*)$ and $\Delta_m(o_i)$:

1. If $\Delta_m(o^*) \leq \Delta_m(o_i)$, then $o^*$ has been accessed, thus the termination condition is *always* correct.
2. If $\Delta_m(o^*) > \Delta_m(o_i)$, then the algorithm may produce incorrect result if $o_{\min}$ may not be a $c$-ANN, i.e., $dist(o_{\min}) > c \cdot dist(o^*)$). Nevertheless, the probability of such problematic cases can be bounded (thanks to Corollary 1) as:

$$\mathbf{Pr}[\Delta_m(o^*) > \Delta_m(o_i)] = 1 - \Psi_m\left(\frac{\Delta_m^2(o_i)}{dist^2(o^*)}\right)$$
$$< 1 - \Psi_m\left(\frac{c^2 \cdot \Delta_m^2(o_i)}{dist^2(o_{\min})}\right) \leq 1 - p'_\tau.$$

Hence, the theorem follows. □

Note that Theorem 2 follows from Theorem 5.

## 6.4 Main Theorem about Algorithm 1

PROOF (SKETCH). Theorem 1 can be proved by considering *only* the case when Algorithm 1 stops due to the early-termination condition. □

## 7. DISCUSSIONS

In this section, we provide more discussions on our methods and differentiate them with existing ones.

## 7.1 More on Our Methods

*Update.* It is obvious that our methods support efficient update as our index is just an $R$-tree. In addition, since our index is typically very small, updates cost is accordingly low.

*About m.* Our method essentially reduces a $c$-ANN query in a $d$-dimensional space into a $T$-NN query in a $m$-dimensional space. Our experimental evaluation demonstrates that current solution with $R$-tree and $m = 6$ seems to be sufficient for a large spectrum of settings. Note also that due to the tiny size of our index (See Table 5 for example), it is likely that a large part of the index, if not the entire index, can be loaded into the main memory, in a way reminiscent of the VA-File [45]. Not only does this save many I/Os, it also enables us to exploit recent development for efficient in-memory indexes for low to medium dimensional spaces (e.g., Cover Tree [6]) and indexes exploiting low intrinsic dimensionalities (e.g., RBC [9]), such that the in-memory processing can be performed efficiently.

## 7.2 Comparison with Existing Methods

Both LSB-forest [42] and C2LSH [17] are designed to answer $c$-ANN queries with an small index in external memory.

Compared with them, our method uses a much smaller index (typically 2–5% of C2LSH and at least 2 orders of magnitude smaller than LSB-forest in our experiments), while achieving the same error bound (i.e., $c$) and confidence (i.e., $p_\tau$). This is because LSB-forest uses $O((dn)^{1.5})$ space to achieve the query complexity of $O((dn)^{1/2})$ I/Os. C2LSH uses $O(n \log n)$ space to achieve the query complexity of $O(n \log n)$ I/Os, as in its default setting, $\beta = O(1/n)$. If we set $\beta = O(1)$ (as we did in the experiments), then C2LSH can use $O(n)$ space with query complexity $O(n)$, hence matching the complexity of our methods. However, the constants for C2LSH are much larger than ours. For example, when set to access at most $0.00242n$ points during the query processing, our methods use $m = 6$ projections while C2LSH needs to use $m = 215$ projections. There are at least two reasons: (1) C2LSH applies quantization for the projections (into buckets), hence needs more projections to distinguish points falling into the same bucket. (2) Chernoff tail bounds are applied to derive $m$ while we use the c.d.f. of the $\chi^2$ distribution to compute $m$ exactly.

Furthermore, our method is more flexible in the following aspects.

- Both LSB-forest and C2LSH need preprocessing to scale floating numbers to integers; this is not required by our method.
- Both LSB-forest and C2LSH can only work with $c = i^2$, for integer $i \geq 2$. Hence, the smallest such $c$ is 4. They need to make $O(1/\epsilon)$ copies to handle $c = 2+\epsilon$. Therefore, they cannot handle $c \leq 2$. In contrast, our methods work with any $c > 1$, and SRS-2 can handle the case of $c = 1$.

Part of our method is similar to the proposal in [20]. However, there are a few important differences:

- To support $c$-ANN queries, [20] needs to use a small $\epsilon < \frac{c-1}{c+1}$. This entails $m = C \cdot \log(n)/\epsilon^2$, for $C$ at least 4 [13]. It is easy to verify that this number is still too large to admit an efficient indexing method to answer exact NN queries, and does not scale with $n$. By using $k$-NN queries and computing the tail probabilities exactly rather than using tail bounds, we can drastically reduce the resulting $m$ to a small constant that admits reasonably efficient indexing.
- Our early-termination condition fully exploits the knowledge of the distribution of the projected distances, and it is both novel and effective.

## 8. EXPERIMENTAL EVALUATION

We report experimental results in this section.

### 8.1 Experiment Setup

We consider the state-of-the-art methods that have theoretical guarantees and can work on external memory. Hence, the following algorithms are used.

- We use the four algorithms proposed in this paper. We mainly use SRS-12 and SRS-1 for $c$-ANN queries, along with SRS-2 for ANN queries (i.e., $c = 1$), and also evaluate SRS-12$^+$ for varying degrees of desirable approximation ratios. $m$ is set to 6 in all the experiments.
- **LSB-forest** is a state-of-the-art method for $c$-ANN queries for high-dimensional data with guarantees of $1/2-1/e$ [42]. Note that LSB-forest has been shown to outperform earlier approaches such as iDistance[22] and MedRank [16]. Therefore, we do not compare with them here.
- **C2LSH** [17] is another recent solution with theoretical guarantee. We mainly consider C2LSH without its optimization as otherwise it loses the theoretical guarantees.[5] We do consider C2LSH with $c = 9$ and its optimization (denoted as **C2LSH**$^*$) on Tiny dataset.

To compare these algorithms fairly, the success probability (i.e., $p_\tau$) of all algorithms is set to be $1/2-1/e$. The approximation ratio $c$ is set to 4, which is the smallest $c$ C2LSH and LSB-forest can support. We use the `C++` source codes provided by the authors of [42] and [17]. Our algorithms were implemented in `C++`. All programs were complied with `gcc` 4.7 with `-O3`. All experiments were conducted on a PC with Intel Xeon X3330@2.66GHz, 4GB memory, 500GB hard disk, running Linux 2.6.

We use five publicly available real-world datasets. We set page size $B$ according to what LSB-forest requires. For each dataset, we first remove the duplicated points, then reserve 100 random data points as queries, finally, we scale up values to integers as required by LSH-forest and C2LSH.

[5]The optimization substantially lowers the count threshold. While $\mathcal{P}_1$ still holds, $\mathcal{P}_2$ does not. Also see Section 8.5.

- **Audio** has about 0.05 million 192-dimensional audio feature vectors extract by the Marsyas library from the DARPA TIMIT audio speech database.[6] $B$ is set to be 1,024.
- **SUN** contains about 0.08 million 512-dimensional GIST features of images.[7] $B$ is set to 2,048.
- **Enron** origins from a collection of emails.[8] We extract bi-grams and form feature vectors of 1,369 dimensions. $B$ is set to 4,096.
- **Tiny** contains over 8 million 384-dimensional GIST feature vectors.[9] $B$ is set to 2,048.
- **ANN_SIFT1B** contains nearly 1 billion 128-dimension SIFT feature vector from the ANN_SIFT1B dataset[10]. $B$ is set to 2,048.

Dataset statistics are listed on the LHS of Table 5.

We evaluate the following metrics.

- We follow previous methods [42, 17] and measure the **I/O costs** of algorithms. Since our index is very small, we deliberately turn off buffering in all experiments.
- We use the **overall ratio** defined in [42] to measure the accuracy of the results. For a $c$-$k$-ANN query, it is defined as $\frac{1}{k}\sum_{i=1}^{k}(dist(o_i)/dist(o_i^*))$, where $o_i$ is the $i$-th returned point, and $o_i^*$ is the true $i$-th nearest point.
- We measure the sizes of indexes created by the algorithms. As each $B^+$-tree in the LSB-forest is a clustered index, we discount the data size from its index size.
- We measure the empirical **success probability** an algorithm. We run the algorithm 100 times for the same query but with different indexes built with random seeds, and measure the percentage of times the point return by the algorithm is indeed within approximation ratio $c$.

I/O costs and overall ratios are averaged over all queries.

### 8.2 Index Size and Indexing Time

We list the index sizes of all algorithms (our algorithms use the same SRS index) on the RHS of Table 5. We can see that SRS is the smallest by far: LSB-forest and C2LSH are 729–128,850 times and 22-64 times larger than SRS, respectively.

This is mainly because of the number of projections used. SRS uses only 6 projections, while C2LSH uses 215 random projections, and LSB-forest needs 100–1247 rounds of 19–29 projections. Also note that the index size of LSB-forest is especially large when $d$ or $n$ is large, as its index size grows at the rate of $O(d^{1.5}n^{1.5})$. In contrast, C2LSH and SRS are independent of $d$, and grow only *linearly* with $n$.

Due to the implementation differences, we are not able to fairly compare indexing time. Nevertheless, given the huge difference in index sizes, both theoretically and empirically, the indexing time of SRS is the smallest, followed by C2LSH, and then LSB-forest. For example, on the 8 million Tiny dataset, LSB-forest will take 48 hours to build just a single LSB-tree, while C2LSH takes about 4 hours, and SRS takes less than 2 hours.

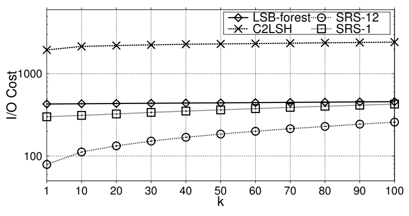
### 8.3 I/O Cost

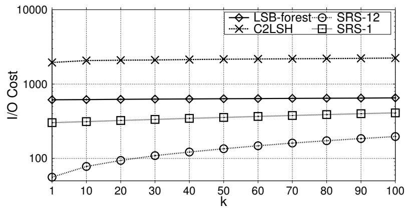
We evaluate the I/O costs for all algorithms for $c$-$k$-ANN queries with $k$ from 1 to 100. The results are shown in Figures 3(a)-3(c) and 3(g). We observe that

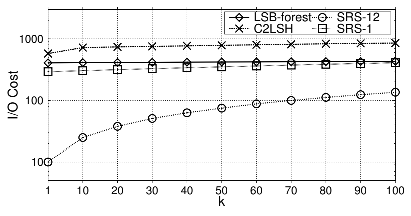
[6]`http://www.cs.princeton.edu/cass/audio.tar.gz`
[7]`http://groups.csail.mit.edu/vision/SUN/`
[8]`http://www.cs.cmu.edu/~enron/`
[9]`http://horatio.cs.nyu.edu/mit/tiny/data/`
[10]`http://corpus-texmex.irisa.fr`

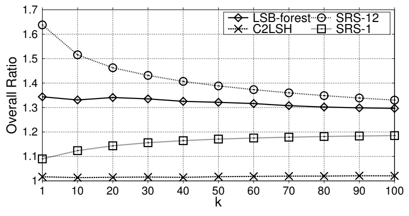
**Table 5: Statistics of the Datasets and Index Sizes (in Megabytes) (Italic numbers in parentheses are conservative estimates, as the indexes are too large to be built on the PC used in the experiment)**

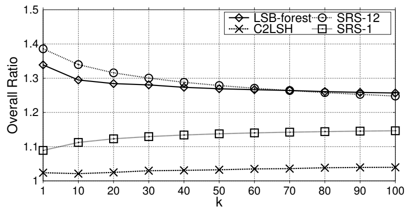
| Dataset | | Statistics | | | | Index Sizes (MB) | | |
|---|---|---|---|---|---|---|---|---|
| | | $n$ | $d$ | Domain Size | Data Size | LSB-forest | C2LSH | SRS |
| Small | **Audio** | 54,287 | 192 | [0, 100,000] | 39.8 | 1,458.7 | 127.4 | 2.0 |
| | **SUN** | 80,006 | 512 | [0, 100,000] | 156.3 | 12,162.8 | 185.0 | 2.9 |
| | **Enron** | 95,863 | 1,369 | [0, 10,000] | 500.8 | 12,745.0 | 85.9 | 3.9 |
| Medium | **Tiny** | 8,288,062 | 384 | [0, 100,000] | 12,140.7 | (*39,982,311.6*) | 7,851.2 | 310.3 |
| Large | **ANN_SIFT1B** | 999,494,170 | 128 | [0, 255] | 122,008.6 | (*12,535,703,042.7*) | (*819,745.1*) | 37,117.1 |

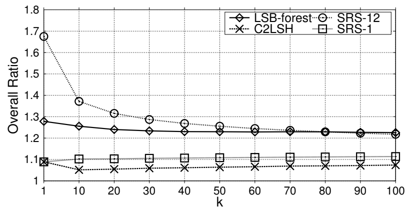

(a) Audio, I/O Cost

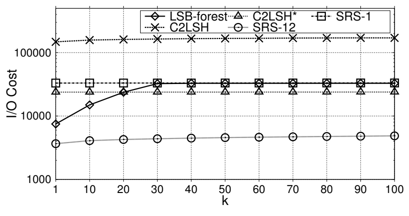

(b) SUN, I/O Cost

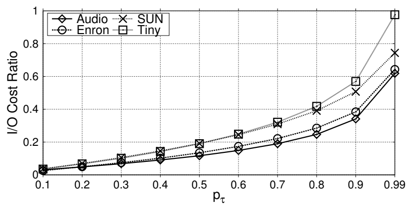

(c) Enron, I/O Cost

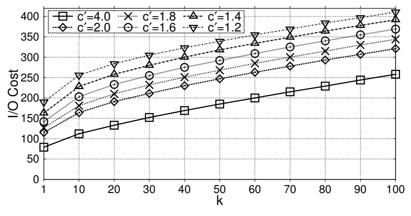

(d) Audio, Overall Ratio

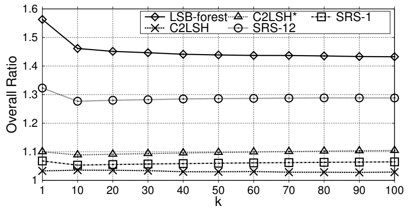

(e) SUN, Overall Ratio

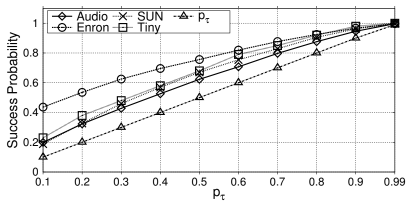

(f) Enron, Overall Ratio


(g) Tiny, I/O Cost


(h) ANN Search, I/O Cost Ratio


(i) Audio, I/O Cost vs. Different $c'$


(j) Tiny, Overall Ratio


(k) ANN Search, Success Probability

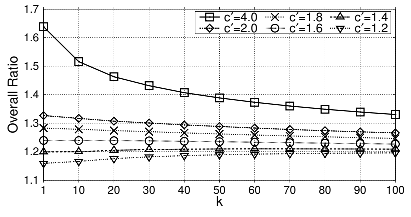

(l) Audio, Ratio vs. Different $c'$

**Figure 3: Experiment Results**

- We first consider the first three small datasets. When $k = 1$, SRS-12 always requires the smallest I/Os; its I/O cost is only 2.5%–18.5% of LSB-forest and 1.8%–4.0% of C2LSH. This is due to the effectiveness of the early-termination test. Even when the early-termination condition is disabled (i.e., SRS-1), the I/O cost is still better than other algorithms, e.g., I/O cost of SRS-1 is 49.3%–72.0% of LSB-forest and 15.4%–42.1% of C2LSH.

  LSB-forest typically uses slightly more I/Os than SRS-1, and C2LSH always has the largest I/O cost by far. This is consistent with the original report [17] when C2LSH's optimization is turned off.

- Consider the large Tiny dataset (Figure 3(g)). We cannot run LSB-forest to completion, as it requires 1,247 LSB-tree indexes, which is about 40TB. Instead, we report *extrapolated* results based on 34 LSB-trees here. We also include C2LSH$^*$, which is the C2LSH algorithm with $c = 9$ and optimization on.

  The results are slightly different. I/O cost of SRS-12 is still the minimum, followed by LSB-forest, SRS-1, C2LSH$^*$, and finally C2LSH. LSB-forest works well as its I/O cost is $O(\sqrt{n})$ (at the expense of $O(n^{3/2})$ index size), while our SRS-1's I/O cost is linear in $n$. Hence, when $n$ is sufficiently large, LSB-forest uses relatively fewer I/Os.

Note that the dilemma is that a large $n$ also means LSB-forest may not be able to be built as its index is too large.

C2LSH uses much more I/Os than SRS-1, which uses more I/Os than C2LSH$^*$. It shows that C2LSH needs more I/Os to achieve the same guarantee as our methods.

- When $k$ increases, the I/O costs of LSB-forest, C2LSH, and SRS-1 all rise gently, with SRS-12 having the steepest slope. This is because (i) SRS-12 has the lowest starting point, as it uses only a tiny amount of I/Os when $k = 1$. (ii) SRS-12 has to perform more rounds of incremental $k$-NN search on a $R$-tree, and this usually requires additional I/Os for the internal $R$-tree pages. This also explains the growth trend for SRS-1 too. Nevertheless, even with $k = 100$, SRS-12 still uses the minimum amount of I/Os among all the algorithms, on both small and large datasets.

## 8.4 Overall Ratio

We also evaluate the overall ratios of $c$-$k$-ANN queries for all algorithms with varying $k$. The results are shown in Figures 3(d)-3(f) and 3(j), and they should be viewed in conjunction with their corresponding I/O plots (i.e., figures above). We can make the following observations:

- Firstly, all algorithms return high-quality results, with approximation ratio much lower than $c = 4$. C2LSH and SRS-1 always have the best ratios, which are always below 1.2. They perform better than LSB-forest especially when $k$ is small (e.g., 1.089 vs. 1.339 on SUN when $k = 1$). SRS-12's ratio is usually higher than LSB-forest when $k = 1$, but they become comparable with large $k$; it even outperforms LSB-forest under some settings (e.g., 1.218 vs. 1.226 on Enron when $k = 100$).

  We can see clear trade-offs between the overall ratio and the I/O cost for all algorithms. The reason why C2LSH has the best ratio is because it requires an extraordinary amount of I/Os. Notably, SRS-1 uses a small fraction of I/Os of C2LSH (e.g., 1/5 on SUN), while achieves comparable qualities (e.g., 1.089 vs. 1.024 on SUN), not to mention its small index size. SRS-12 is designed to stop as soon as a $c$-ANN point is found with certain confidence. Note that it can be tuned to return a better quality result by its variant SRS-12$^+$ with a custom $c'$, which will be discussed in Section 8.7.
- In terms of trends, the ratios of SRS-1 and C2LSH grow with increasing $k$. This is because both algorithms return the best $k$ points within a candidate set with size approximately $T + k - 1$. When $k$ increases, only few additional candidates are checked, hence the overall ratio increases.

  In contrast, the ratios of SRS-12 and LSB-forest decrease with $k$. Because they are usually stopped by their respective early-termination conditions. Thus with a large $k$, more points are accessed and hence the ratios improve.
- On the large Tiny dataset, LSB-forest has the worst ratio, as the ratio is the minimum among 34 LSB-trees (out of the 1247 required). SRS-12 has reasonable ratio as it is designed to stop as early as possible. The other three algorithms all have good ratios below 1.1. C2LSH has the best ratio, followed by SRS-1, and then C2LSH$^*$.

## 8.5 Empirical Success Probability

We create a hard dataset with $n = 10,000$ and $d = 128$ to evaluate the empirical success probability for $c$-ANN query using different algorithms. We fix the query $q$ and one point with distance to $q$ as $u$, while the rest points at distance $(c + \epsilon) \cdot u$ to $q$ with a small $\epsilon$. We choose $c = 4$.

Those algorithms with guarantees all achieve much higher success probabilities than the theoretical bound. For example, SRS-1 achieves the highest success probability which is 100%, and SRS-12 achieves the lowest which is 78%. Algorithms without guarantees has much lower success probability. For example, C2LSH$^*$ only achieves 29%.

## 8.6 Approximate NN Search with SRS-2

SRS-2 can return $c$-ANN for any $c \geq 1$ and desired success probability $1 > p_\tau > 0$. Here we only focus on the interesting and also the hardest case of approximate NN search by setting $c = 1$ and varying $p_\tau$. We show the results on all datasets in Figures 3(h) and 3(k). We can observe that

- The resulting success probability of our algorithm is *always* larger than the user-given threshold $p_\tau$, thanks to our theoretical guarantees. This also shows (together with Section 8.5) that the actual performances of our algorithms are usually better than the worst-case lower bound.
- We measure the **I/O cost ratio**, defined as the I/O cost of our algorithm over that of the brute-force, linear scan algorithm. SRS-2 uses fewer I/Os than the linear scan algorithm on all the datasets, especially when $p_\tau$ is low. For example, on the Audio dataset, by using only 14.9% (resp. 61.9%) of the I/Os of linear scan, it returns the NN with 70.9% (resp. 99.7%) probability.

## 8.7 Tuning Approximation Ratio by SRS-12$^+$

We evaluate the SRS-12$^+$ algorithm which essentially is the SRS-12 but with an additional "desirable" approximation ratio $c'$. We show the results only on the Audio dataset in Figures 3(i) and 3(l), results on other datasets are similar.

We can observe that giving an increasingly smaller $c'$, the algorithm returns better quality results (note that for $c' = 4$, it is essentially identical to SRS-12). In fact, the top-1 result always has a better approximation ratio than the given $c'$. E.g., when $c' = 1.6$, the result is of ratio 1.239; when $c' = 1.2$, the result is of ratio 1.159. The algorithm achieves this by using a more stringest early-termination test, hence it entails accessing more points. The I/O cost of SRS-2 increases when $c'$ decreases.

## 8.8 Large Dataset

We run our algorithms on the large ANN_SIFT1B dataset with different scales to evaluate performance on non-trivial sized datasets (See Table 6 for results). Unfortunately, at such a scale, LSB-forest simply cannot run, as a single LSB-tree index may require 2.3TB space. C2LSH's implementation requires at least 476.6GB memory to run, so we just extrapolate its performance using smaller $n$ from 0.1M–1M. The only "reasonable" algorithm is C2LSH$^*$, whose overall ratio is about 1.145, and I/O cost on ANN_SIFT1B is 2,951,666.

Our algorithms scales linearly with $n$. SRS-12 uses fewest number of I/Os, and SRS-1 returns the best quality results, though with the largest number of I/Os (which is about 15% of the cost of linear scan). As designed, SRS-12$^+$ achieves a good and tunable balance between cost and quality.

**Table 6: Our Algorithms on ANN_SIFT1B**

| **Alg.** | SRS-12 | | SRS-12$^+$ ($c' = 1.5$) | | SRS-1 | |
|---|---|---|---|---|---|---|
| **Size** | **Ratio** | **I/O** | **Ratio** | **I/O** | **Ratio** | **I/O** |
| **500M** | 1.330 | 4,164 | 1.123 | 113,188 | 1.018 | 1,185,795 |
| **750M** | 1.344 | 5,765 | 1.126 | 160,922 | 1.017 | 1,782,967 |
| **1000M** | 1.343 | 7,096 | 1.127 | 204,907 | 1.019 | 2,452,974 |

## 9. CONCLUSIONS

In this paper, we propose several simple yet effective methods to process $c$-approximate nearest neighbor queries for high-dimensional points with provable guarantees. We designed four algorithms with various features, all operating on a tiny index. We demonstrate superior performance against the state-of-the-art LSH-based methods in our experiments, and the fact our methods scale well to 1 billion high-dimensional points using just a single commodity PC.

*Acknowledgement.* The authors would like to thank Jianlin Feng, Yikai Zhang and Qiang Huang for helpful discussions and pointing out errors in an early manuscript.

Wei Wang and Jianbin Qin are supported by ARC DP130103401 and DP130103405. Ying Zhang is supported by ARC DE140100679 and DP130103245. Xuemin Lin is supported by NSFC61232006, NSFC61021004, ARC DP120104168 and DP140103578.

## 10. REFERENCES

[1] N. Ailon and B. Chazelle. Faster dimension reduction. *Commun. ACM*, 53(2), 2010.

[2] A. Andoni, P. Indyk, H. L. Nguyen, and I. Razenshteyn. Beyond locality-sensitive hashing. In *SODA*, 2014.

[3] S. Arya, T. Malamatos, and D. M. Mount. Space-time tradeoffs for approximate nearest neighbor searching. *J. ACM*, 57(1), 2009.

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6), 1998.

[5] B. Bahmani, A. Goel, and R. Shinde. Efficient distributed locality sensitive hashing. In *CIKM*, 2012.

[6] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, 2006.

[7] A. Borodin, R. Ostrovsky, and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *STOC*, 1999.

[8] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, 1998.

[9] L. Cayton. Accelerating nearest neighbor search on manycore systems. In *IPDPS*, 2012.

[10] A. Chakrabarti, B. Chazelle, B. Gum, and A. Lvov. A lower bound on the complexity of approximate nearest-neighbor searching on the hamming cube. In *STOC*, 1999.

[11] T. M. Chan. Approximate nearest neighbor queries revisited. *Discrete & Computational Geometry*, 20(3), 1998.

[12] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[13] S. Dasgupta and A. Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Struct. Algorithms*, 22(1), 2003.

[14] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, 2004.

[15] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. In *CIKM*, 2008.

[16] R. Fagin et al. Efficient similarity search and classification via rank aggregation. In *SIGMOD Conference*, 2003.

[17] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD Conference*, 2012.

[18] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[19] M. E. Houle et al. Fast approximate similarity search in extremely high-dimensional data sets. In *ICDE*, 2005.

[20] P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 2006.

[21] P. Indyk et al. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[22] H. V. Jagadish et al. idistance: An adaptive b$^+$-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2), 2005.

[23] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1), 2011.

[24] W. B. Johnson et al. Extensions of lipschitz mapping into hilbert space. *Contemporary Mathematics*, 26, 1984.

[25] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In *SIGMOD Conference*, 2002.

[26] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *STOC*, 1997.

[27] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. In *SODA*, 2004.

[28] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *STOC*, 1998.

[29] B. Laurent and P. Massart. Adaptive estimation of a quadratic functional by model selection. *The Annals of Statistics*, 28(5), 2000.

[30] T. Liu, A. W. Moore, A. G. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.

[31] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[32] S. Meiser. Point location in arrangements of hyperplanes. *Inf. Comput.*, 106(2), 1993.

[33] R. O'Donnell et al. Optimal lower bounds for locality sensitive hashing (except when q is tiny). In *ICS*, 2011.

[34] J. Pan and D. Manocha. Bi-level locality sensitive hashing for k-nearest neighbor computation. In *ICDE*, 2012.

[35] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, 2006.

[36] V. Pestov. Lower bounds on performance of metric tree indexing schemes for exact similarity search in high dimensions. *Algorithmica*, 66(2), 2013.

[37] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufman, 2006.

[38] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5), 2012.

[39] R. Shinde, A. Goel, P. Gupta, and D. Dutta. Similarity search and locality sensitive hashing using ternary content addressable memories. In *SIGMOD Conference*, 2010.

[40] M. Slaney et al. Optimal parameters for locality-sensitive hashing. *Proceedings of the IEEE*, 100(9), 2012.

[41] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *PVLDB*, 6(14), 2013.

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

[43] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. Knowl. Data Eng.*, 16(10), 2004.

[44] K. Ueno, X. Xi, E. J. Keogh, and D.-J. Lee. Anytime classification using the nearest neighbor algorithm with applications to stream mining. In *ICDM*, 2006.

[45] R. Weber et al. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[46] Y. Weiss et al. Spectral hashing. In *NIPS*, 2008.

[47] A. C.-C. Yao and F. F. Yao. A general approach to d-dimensional geometric queries (extended abstract). In *STOC*, 1985.

[48] S. Yin, M. Badr, and D. Vodislav. Dynamic multi-probe lsh: An i/o efficient index structure for approximate nearest neighbor search. In *DEXA (1)*, 2013.