# Query-Aware Locality-Sensitive Hashing for Approximate Nearest Neighbor Search

Qiang Huang,
Jianlin Feng,Yikai Zhang
School of Software
Sun Yat-sen University
Guangzhou, China
huangq2011@gmail.com
fengjlin@mail.sysu.edu.cn
echo_evenop@yahoo.com

Qiong Fang
School of Software Engineering
South China University of Technology
Guangzhou, China
sefangq@scut.edu.cn

Wilfred Ng
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Hong Kong, China
wilfred@cse.ust.hk

## ABSTRACT

Locality-Sensitive Hashing (LSH) and its variants are the well-known indexing schemes for the $c$-Approximate Nearest Neighbor ($c$-ANN) search problem in high-dimensional Euclidean space. Traditionally, LSH functions are constructed in a *query-oblivious* manner in the sense that buckets are partitioned before any query arrives. However, objects closer to a query may be partitioned into different buckets, which is undesirable. Due to the use of *query-oblivious* bucket partition, the state-of-the-art LSH schemes for external memory, namely C2LSH and LSB-Forest, only work with approximation ratio of integer $c \geq 2$.

In this paper, we introduce a novel concept of *query-aware* bucket partition which uses a given query as the "anchor" for bucket partition. Accordingly, a *query-aware* LSH function is a random projection coupled with *query-aware* bucket partition, which removes random shift required by traditional *query-oblivious* LSH functions. Notably, *query-aware* bucket partition can be easily implemented so that query performance is guaranteed. We propose a novel *query-aware* LSH scheme named QALSH for $c$-ANN search over external memory. Our theoretical studies show that QALSH enjoys a guarantee on query quality. The use of *query-aware* LSH function enables QALSH to work with any approximation ratio $c > 1$. Extensive experiments show that QALSH outperforms C2LSH and LSB-Forest, especially in high-dimensional space. Specifically, by using a ratio $c < 2$, QALSH can achieve much better query quality.

## 1. INTRODUCTION

The problem of Nearest Neighbor (NN) search in Euclidean space has wide applications, such as image and video databases, information retrieval, and data mining. In many applications, data objects are typically represented as Euclidean vectors (or points). For example, in image search applications, images can be naturally mapped into high-dimensional feature vectors with one dimension per pixel.

To bypass the difficulty of finding exact query answers in high-dimensional space, the approximate version of the problem, called the $c$-Approximate Nearest Neighbor ($c$-ANN) search, has attracted extensive studies [13, 10, 3, 7, 15, 4]. For a given approximation ratio $c$ ($c > 1$) and a query object $q$, $c$-ANN search returns the object within distance $c$ times the distance of $q$ to its exact nearest neighbor. Since the approximation ratio $c$ is an upper bound, a smaller $c$ means a better guarantee of query quality.

Locality-Sensitive Hashing (LSH) [7, 2] and its variants [12, 15, 4] are the well-known indexing schemes for $c$-ANN search in high-dimensional space. The seminal work on LSH scheme for Euclidean space was first presented by Datar et al.[2], which is named E2LSH[1] later. E2LSH constructs LSH functions based on $p$-stable distributions. For Euclidean space, 2-stable distribution, i.e., standard normal distribution $\mathcal{N}(0, 1)$, is used in E2LSH and its variants, such as Entropy-LSH [12], LSB-Forest [15] and C2LSH [4].

Under an LSH function for Euclidean space, the probability of collision (or simply collision probability) between two objects decreases monotonically as their Euclidean distance increases. An LSH function of E2LSH has the basic form as follows: $h_{\vec{a},b}(o) = \lfloor \frac{\vec{a} \cdot \vec{o} + b}{w} \rfloor$. Such an LSH function partitions an object into a bucket in the following manner: first it projects object $o$ along the random line identified by $\vec{a}$ (or simply the random line $\vec{a}$), and then gives the projection $\vec{a} \cdot \vec{o}$ a random shift of $b$, and finally uses the floor function to locate the interval of width $w$ in which the shifted projection falls. The interval is simply taken as the bucket of object $o$. In this approach, bucket partition is carried out before any query arrives, and hence it is said to be *query-oblivious*. Accordingly, the corresponding LSH function is called a *query-oblivious* LSH function. An illustration of *query-oblivious* bucket partition is given in Figure 1, where the random line is segmented into buckets $[0, w)$, $[-w, 0)$, $[w, 2w)$, $[-2w, -w)$, and so on. Due to the use of the floor function, here the origin (i.e., 0) of the random line can be viewed as the "anchor" for locating the boundary of each interval. *Query-oblivious* bucket partition has the advantage of leaving the overhead of bucket partition to the pre-

[1] `http://www.mit.edu/~andoni/LSH`

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
*Proceedings of the VLDB Endowment,* Vol. 9, No. 1
Copyright 2015 VLDB Endowment 2150-8097/15/09.

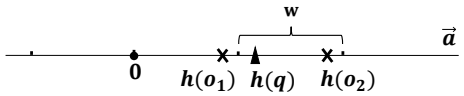



**Figure 1: Query-Oblivious Bucket Partition**

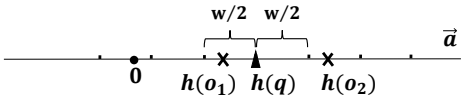



**Figure 2: Query-Aware Bucket Partition**

processing step. However, *query-oblivious* bucket partition may lead to some undesirable situation, i.e., objects closer to a query may be partitioned into different buckets. For example, as shown in Figure 1, although $o_1$ is closer to $q$ than $o_2$, $o_1$ and $q$ are segmented into different buckets.

The basic form of $h_{\vec{a},b}(o)$ has been used by the variants of E2LSH, such as Entropy-LSH and C2LSH. In LSB-Forest, even though the LSH functions ($h_{\vec{a},b}(o) = \vec{a} \cdot \vec{o} + b$) only explicitly involve random projection and random shift, its encoding hash values by Z-order also implicitly use the origin as the "anchor". *Random shift* along the random line is a prerequisite for the *query-oblivious* hash functions to be *locality-sensitive*. In a word, the state-of-the-art LSH schemes for external memory, namely C2LSH and LSB-Forest, are both built on *query-oblivious* bucket partition. As analyzed in Section 5.1, due to the use of *query-oblivious* bucket partition, C2LSH and LSB-Forest only work with integer $c \geq 2$ for $c$-ANN search, which is limited for applications that prefer a ratio as strong as $c < 2$.

Motivated by the limitations of *query-oblivious* bucket partition, we propose a novel concept of *query-aware* bucket partition and develop novel *query-aware* LSH functions accordingly. Given a pre-specified bucket width $w$, a hash function $h_{\vec{a}}(o) = \vec{a} \cdot \vec{o}$ first projects object $o$ along the random line $\vec{a}$ as before. When a query $q$ arrives, we compute the projection of $q$ (i.e., $h_{\vec{a}}(q)$) and take the query projection (or simply the query) as the "anchor" for bucket partition. Specifically, the interval $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$, i.e., a bucket of width $w$ centered at $h_{\vec{a}}(q)$ (or simply at $q$), is first imposed along the random line $\vec{a}$. And if necessary, we can impose buckets with any larger bucket width, in the same manner of using the query as the "anchor". This approach of bucket partition is said to be *query-aware*. In Section 3, we show that the hash function $h_{\vec{a}}(o)$ coupled with *query-aware* bucket partition is indeed locality-sensitive, and hence is called a *query-aware* LSH function. An example of *query-aware* bucket partition is illustrated in Figure 2, where $h(q)$ evenly splits the buckets into two half-buckets of width $\frac{w}{2}$. By applying the *query-aware* bucket partition, $o_1$ and $q$ are partitioned into the same bucket, the undesirable situation illustrated in Figure 1 is then avoided.

Notice that random shift is not necessary for *query-aware* bucket partition. Thus, compared to *query-oblivious* LSH functions, *query-aware* LSH functions are simpler to compute. However, we need to dynamically do *query-aware* bucket partition. Given a *query-aware* LSH function $h_{\vec{a}}(o) = \vec{a} \cdot \vec{o}$, in the pre-processing step, we compute the projections of all the data objects along the random line, and index all the data projections by a $B^+$-tree. When a query object $q$ arrives, we compute the query projection and use the $B^+$-tree to locate objects falling in the interval $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$. And if required by our search algorithm, we can gradually locate data objects even farther away from the query, just like performing a $B^+$-tree range search. In other words, we do not need to physically partition the whole random line at all. Therefore, the overhead of *query-aware* bucket partition is affordable.

Based on *query-aware* LSH functions, we propose a novel *Query-Aware* LSH scheme called QALSH for $c$-ANN search in high-dimensional Euclidean space. Interestingly, as analyzed in Section 5.1, *query-aware* bucket partition enables QALSH to work with any $c > 1$. In this paper, we also develop a novel approach to setting the bucket width $w$ automatically, as shown in Section 5.3. In contrast, the state-of-the-art *query-oblivious* LSH schemes depend on manually setting $w$. For example, both E2LSH and LSB-Forest manually set $w = 4.0$, while C2LSH manually sets $w = 1.0$.

In summary, we introduce a novel concept of *query-aware* bucket partition and develop novel *query-aware* LSH functions accordingly. We propose a novel *query-aware* LSH scheme QALSH for high-dimensional $c$-ANN search over external memory. QALSH works with any approximation ratio $c > 1$ and enjoys a theoretical guarantee on query quality. QALSH also solves the problem of $c$-approximate $k$-nearest neighbors ($c$-$k$-ANN) search. Extensive experiments on four real datasets show that in high-dimensional Euclidean space QALSH outperforms C2LSH and LSB-Forest which also have guarantee on query quality.

The rest of this paper is organized as follows. We first discuss preliminaries in Section 2. Then we introduce the *query-aware* LSH family in Section 3. The QALSH scheme is presented in Section 4 and its theoretical analysis is given in Section 5. Experimental studies are presented in Section 6. Related work is discussed in Section 7. Finally, we conclude our work in Section 8.

# 2. PRELIMINARIES

## 2.1 Problem Setting

Let $D$ be a database of $n$ data objects in $d$-dimensional Euclidean space $\mathcal{R}^d$ and let $\|o_1, o_2\|$ denote the Euclidean distance between two objects $o_1$ and $o_2$. Given a query object $q$ in $\mathcal{R}^d$ and an approximation ratio $c$ ($c > 1$), $c$-ANN search is to find an object $o \in D$ such that $\|o, q\| \leq c\|o^*, q\|$, where $o^*$ is the exact NN of $q$ in $D$. Similarly, $c$-$k$-ANN is to find $k$ objects $o_i \in D$ ($1 \leq i \leq k$) such that $\|o_i, q\| \leq c\|o_i^*, q\|$, where $o_i^*$ is the exact $i$-th NN of $q$ in $D$.

## 2.2 Query-Oblivious LSH Family

A family of LSH functions is able to partition "closer" objects into the same bucket with an accordingly higher probability. If two objects $o$ and $q$ are partitioned into the same bucket by a hash function $h$, we say $o$ and $q$ collide under $h$. Formally, an LSH function family (or simply an LSH family) in Euclidean space is defined as:

*Definition 1.* Given a search radius $r$ and approximation ratio $c$, an LSH function family $H = \{h : \mathcal{R}^d \to U\}$ is said to be $(r, cr, p_1, p_2)$-sensitive, if, for any $o, q \in \mathcal{R}^d$ we have

- if $\|o, q\| \leq r$, then $Pr_H[o \text{ and } q \text{ collide under } h] \geq p_1$;
- if $\|o, q\| > cr$, then $Pr_H[o \text{ and } q \text{ collide under } h] \leq p_2$.

where $c > 1$ and $p_1 > p_2$. For ease of reference, $p_1$ and $p_2$ are called positively-colliding probability and negatively-colliding probability, respectively.

A query-oblivious LSH family is an LSH family $H = \{h : \mathcal{R}^d \to \mathcal{Z}\}$ where each hash function $h$ exploits *query-oblivious* bucket partition, i.e., buckets in the hash table of $h$ are statically determined before any query arrives. Normally, for a query-oblivious LSH function $h$, two objects $o$ and $q$ collide under $h$ means $h(o) = h(q)$, where $h(o)$ identifies the bucket of $o$. A typical query-oblivious LSH function is formally defined as follows [2].

$$h_{\vec{a},b}(o) = \left\lfloor \frac{\vec{a} \cdot \vec{o} + b}{w} \right\rfloor, \tag{1}$$

where $\vec{o}$ is a $d$-dimensional Euclidean vector representing object $o$, $\vec{a}$ is a $d$-dimensional random vector with each entry drawn independently from standard normal distribution $\mathcal{N}(0,1)$. $w$ is the pre-specified bucket width, and $b$ is a real number uniformly drawn from $[0, w)$.

For two objects $o_1$ and $o_2$, and a uniformly randomly chosen hash function $h_{\vec{a},b}$, let $s = \|o_1, o_2\|$, and then their collision probability is computed as follows [2]:

$$\begin{aligned} \xi(s) &= Pr_{\vec{a},b}[h_{\vec{a},b}(o_1) = h_{\vec{a},b}(o_2)] \\ &= \int_0^w \tfrac{1}{s} f_2(\tfrac{t}{s})(1 - \tfrac{t}{w})\, dt \end{aligned} \tag{2}$$

where $f_2(x) = \frac{2}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$. For a fixed $w$, $\xi(s)$ decreases monotonically as $s$ increases. With $\xi_1 = \xi(r)$ and $\xi_2 = \xi(cr)$, the family of hash functions $h_{\vec{a},b}$ is $(r, cr, \xi_1, \xi_2)$-sensitive. Specifically, if we set $r = 1$ and $cr = c$, we have Lemma 1 as follows [2] :

LEMMA 1. *The* query-oblivious *LSH family identified by Equation 1 is* $(1, c, \xi_1, \xi_2)$*-sensitive, where* $\xi_1 = \xi(1)$ *and* $\xi_2 = \xi(c)$.

# 3. QUERY-AWARE LSH FAMILY

In this section we first introduce the concept of *query-aware* LSH functions. Then we make a computational comparison of positively- and negatively-colliding probabilities between *query-oblivious* and *query-aware* LSH families. Finally, we show that *query-aware* LSH family is able to support *virtual rehashing* in a simple and quick manner.

## 3.1 $(1, c, p_1, p_2)$-sensitive LSH Family

Constructing LSH functions in a *query-aware* manner consists of two steps: random projection and *query-aware* bucket partition. Formally, a *query-aware* hash function $h_{\vec{a}}(o) : \mathcal{R}^d \to \mathcal{R}$ maps a $d$-dimensional object $\vec{o}$ to a number along the real line identified by a random vector $\vec{a}$, whose entries are drawn independently from $\mathcal{N}(0,1)$. For a fixed $\vec{a}$, the corresponding hash function $h_{\vec{a}}(o)$ is defined as follows:

$$h_{\vec{a}}(o) = \vec{a} \cdot \vec{o} \tag{3}$$

For all the data objects, their projections along the random line $\vec{a}$ are computed in the pre-processing step. When a query object $q$ arrives, we obtain the query projection by computing $h_{\vec{a}}(q)$. Then, we use the query as the "anchor" to locate the *anchor bucket* with width $w$ (defined by $h_{\vec{a}}(\cdot)$), i.e., the interval $[h_{\vec{a}}(q) - \frac{w}{2},\ h_{\vec{a}}(q) + \frac{w}{2}]$. If the projection of an object $o$ (i.e., $h_{\vec{a}}(o)$), falls in the anchor bucket with width $w$, i.e., $|h_{\vec{a}}(o) - h_{\vec{a}}(q)| \le \frac{w}{2}$, we say $o$ collides with $q$ under $h_{\vec{a}}$.

We now show that the family of hash functions $h_{\vec{a}}(o)$ coupled with *query-aware* bucket partition is locality-sensitive. In this sense, each $h_{\vec{a}}(o)$ in the family is said to be a *query-aware* LSH function. For objects $o$ and $q$, let $s = \|o, q\|$. Due to the stability of standard normal distribution $\mathcal{N}(0,1)$, we have that $(\vec{a} \cdot \vec{o} - \vec{a} \cdot \vec{q})$ is distributed as $sX$, where $X$ is a random variable drawn from $\mathcal{N}(0,1)$ [2]. Let $\varphi(x)$ be the probability density function (PDF) of $\mathcal{N}(0,1)$, i.e., $\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$. The collision probability between $o$ and $q$ under $h_{\vec{a}}$ is computed as follows:

$$\begin{aligned} p(s) &= Pr_{\vec{a}}[|h_{\vec{a}}(o) - h_{\vec{a}}(q)| \le \tfrac{w}{2}] = Pr[|sX| \le \tfrac{w}{2}] \\ &= Pr[-\tfrac{w}{2s} \le X \le \tfrac{w}{2s}] = \int_{-\frac{w}{2s}}^{\frac{w}{2s}} \varphi(x)\, dx \end{aligned} \tag{4}$$

Accordingly, we have Lemma 2 as follows:

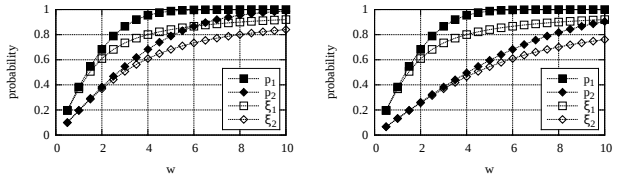
LEMMA 2. *The* query-aware *hash family of all the hash functions* $h_{\vec{a}}(o)$ *that are identified by Equation 3 and coupled with* query-aware *bucket partition is* $(1, c, p_1, p_2)$*-sensitive, where* $p_1 = p(1)$ *and* $p_2 = p(c)$.

PROOF. Referring to Equation 4 , a simple calculation shows that $p(s) = 1 - 2norm(-\frac{w}{2s})$, where $norm(x) = \int_{-\infty}^{x} \varphi(t)\, dt$. Note that $norm(x)$ is simply the cumulative distribution function (CDF) of $\mathcal{N}(0,1)$, which increases monotonically as $x$ increases. For a fixed $w$, $norm(-\frac{w}{2s})$ increases monotonically as $s$ increases, and hence $p(s)$ decreases monotonically as $s$ increases. Therefore, according to Definition 1, the *query-aware* hash family identified by Equation 3, is $(1, c, p_1, p_2)$-sensitive, where $p_1 = p(1)$ and $p_2 = p(c)$, respectively. □

## 3.2 Comparison of Colliding Probabilities

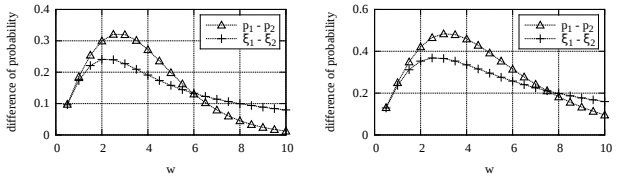
The effectiveness of an $(r, cr, p_1, p_2)$-sensitive hash family depends on the difference between the positively-colliding probability and negatively-colliding probability, i.e., $(p_1 - p_2)$, since the difference measures the degree that positively-colliding data objects of a query $q$ can be discriminated from negatively-colliding ones. We now show that the novel *query-aware* hash family leads to larger $(p_1 - p_2)$ under typical settings of bucket width $w$. For *query-aware* LSH family, from the proof of Lemma 2, we have $p_1 = 1 - 2norm(-\frac{w}{2})$ and $p_2 = 1 - 2norm(-\frac{w}{2c})$. For *query-oblivious* LSH family, we have $\xi_1 = 1 - 2norm(-w) - \frac{2}{\sqrt{2\pi}w}(1 - e^{-(w^2/2)})$ and $\xi_2 = 1 - 2norm(-w/c) - \frac{2}{\sqrt{2\pi}w/c}(1 - e^{-(w^2/2c^2)})$ [2].

Bucket width $w$ is a critical parameter of an LSH function. While E2LSH and LSB-Forest manually set $w = 4.0$, C2LSH manually sets $w = 1.0$. For $w$ in the range $[0, 10]$, starting from 0.5 and with a step of 0.5, we show the variations of the colliding probabilities $p_1$, $p_2$, $\xi_1$, and $\xi_2$ for two different $c$ values in Figure 3. We find that all the colliding probabilities monotonically increase as $w$ increases, and get very close to 1 as $w$ gets close to 10. In addition, $p_1$ and $p_2$ are consistently larger than $\xi_1$ and $\xi_2$, respectively. Thus, we also show the two differences $(p_1 - p_2)$ and $(\xi_1 - \xi_2)$ with respect to $w$ in Figure 4. We have two interesting observations: (1) $(p_1 - p_2)$ is larger than $(\xi_1 - \xi_2)$ under typical bucket widths, namely $w = 4.0$ and $w = 1.0$. (2) Both $(p_1 - p_2)$ and $(\xi_1 - \xi_2)$ tend to have maximum values in the $w$ range $[0, 10]$. Observation (1) indicates that our novel *query-aware* LSH family can be used to improve the performance of *query-oblivious* LSH schemes such as C2LSH by leveraging a larger $(p_1 - p_2)$. Observation (2) inspires us to automatically set




(a) c = 2.0 (b) c = 3.0

**Figure 3: Positively-colliding probability and negatively-colliding probability**




(a) c = 2.0 (b) c = 3.0

**Figure 4: Difference between positively-colliding probability and negatively-colliding probability**

bucket width $w$ by maximizing the difference $(p_1 - p_2)$, which actually leads to the minimization of the number of hash tables in QALSH as analyzed in Section 5.3.1.

Since C2LSH has been shown to outperform LSB-Forest in high-dimensional space, and the *query-aware* bucket partition is easy to implement for a single *query-aware* LSH function, in this paper we propose to follow C2LSH's general framework to demonstrate the desirability of *query-aware* LSH families.

## 3.3 Virtual Rehashing

LSH schemes such as C2LSH do not solve the $c$-ANN search problem directly. This is because an $(R, cR, p_1, p_2)$-sensitive LSH family requires $R$ to be pre-specified so as to compute $p_1$ and $p_2$. It is the decision version of the $c$-ANN search problem, i.e., the $(R, c)$-NN search problem, that can be directly solved by exploiting an $(R, cR, p_1, p_2)$-sensitive LSH family. Given a query object $q$ and a search radius $R$, the $(R, c)$-NN search problem is to find a data object $o_1$ whose distance to $q$ is at most $cR$ if there exists a data object $o_2$ whose distance to $q$ is at most $R$.

The $c$-ANN search of a query $q$ is reduced to a series of the $(R, c)$-NN search of $q$ with properly increasing search radius $R \in \{1, c, c^2, c^3, ...\}$. Therefore, for each $R$, we need an $(R,cR,p_1,p_2)$-sensitive LSH family. For each $R \in \{c, c^2, ...\}$ , by deriving an $(R,cR,p_1,p_2)$-sensitive hash family from the $(1,c,p_1,p_2)$-sensitive hash family for $R = 1$, hash tables for all the subsequent radii can be virtually imposed on the physical hash tables for $R = 1$. This is the underlying idea of *virtual rehashing* of C2LSH.

We now show that QALSH can also do virtual rehashing by deriving $(R, cR, p_1, p_2)$-sensitive functions from Equation 3. Virtual rehashing of QALSH enables it to work with any $c > 1$, while both C2LSH and LSB-Forest only work with integer $c \geq 2$. A formal proof of this advantage is given in Section 5.1.

PROPOSITION 1. *The* query-aware *hash family*

$$H^R_{\vec{a}}(o) = \frac{h_{\vec{a}}(o)}{R}$$

*is $(R, cR, p_1, p_2)$-sensitive, where $c$, $p_1$, $p_2$ and $h_{\vec{a}}(\cdot)$ are the same as defined in Lemma 2, and $R$ is a power of $c$ (i.e., $c^k$ for some integer $k \geq 1$).*

PROOF. Let $\vec{o'} = \frac{\vec{o}}{R}$, from Equation 3, we have $H^R_{\vec{a}}(o) = \frac{\vec{a}\cdot\vec{o}}{R} = \vec{a} \cdot \vec{o'} = h_{\vec{a}}(o')$. By Lemma 2, we assert that $h_{\vec{a}}(o')$ is $(1, c, p_1, p_2)$-sensitive. The assertion implies, objects $o'_1$ and $o'_2$ collide under $h_{\vec{a}}(\cdot)$ with a probability at least $p_1$ if $\|o'_1, o'_2\| \leq 1$. $\|o'_1, o'_2\| \leq 1$ is equivalent to $\|o_1, o_2\| \leq R$. Thus, it follows that $o_1$ and $o_2$ collide with a probability at least $p_1$ under $H^R_{\vec{a}}(\cdot)$ if $\|o_1, o_2\| \leq R$. Similarly, the assertion also implies, $o_1$ and $o_2$ collide under $H^R_{\vec{a}}(\cdot)$ with a probability at most $p_2$ if $\|o_1, o_2\| \geq cR$. Therefore, the *query-aware* hash family $H^R_{\vec{a}}(o)$ is $(R, cR, p_1, p_2)$-sensitive. □

Given a query $q$ and a pre-specified bucket width $w$, for $R \in \{1, c, c^2, ...\}$, we now define the round-$R$ anchor bucket $B^R$ as the anchor bucket with width $w$ defined by $H^R_{\vec{a}}(\cdot)$, i.e., the interval $[H^R_{\vec{a}}(q) - \frac{w}{2}, H^R_{\vec{a}}(q) + \frac{w}{2}]$, which is centered at $q$'s projection $H^R_{\vec{a}}(q)$ along the random line $\vec{a}$. In other words, the round-$R$ anchor bucket is located by *query-aware* bucket partition with bucket width $w$ as before. Specifically, $B^1$ is simply the interval $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$, which is the anchor bucket with width $w$ defined by $h_{\vec{a}}(\cdot)$.

As shown in Section 4.1, to find the $(R, c)$-NN of a query $q$, we only need to check the round-$R$ anchor bucket $B^R$ for the specific $R$. To find the $c$-ANN of $q$, we check the round-$R$ anchor buckets round by round for gradually increasing $R$ ($R \in \{1, c, c^2, ...\}$). All the round-$R$ anchor buckets defined by $H^R_{\vec{a}}(\cdot)$ are centered at $q$ along the same $\vec{a}$, and can be located along $\vec{a}$ with properly adjusted bucket width. Therefore, we only need to keep one physical copy of the data projections along $\vec{a}$. Using the results of the following Propositions 2 and 3, we can virtually impose $B^R$ over $B^1$, and hence $B^{cR}$ over $B^R$. This is the underlying idea of virtual rehashing of QALSH. Here we only show the proof of Proposition 2 since the proof of Proposition 3 is similar.

PROPOSITION 2. *Given $q$ and $w$, $B^R$ contains $B^1$, and the width of $B^R$ is $R$ times the width of $B^1$, i.e., $wR$.*

PROOF. According to the definition of $B^R$, for each object $o$ in $B^R$, we have $|H^R_{\vec{a}}(o) - H^R_{\vec{a}}(q)| \leq \frac{w}{2}$, i.e., $|\frac{h_{\vec{a}}(o)}{R} - \frac{h_{\vec{a}}(q)}{R}| \leq \frac{w}{2}$. Hence, we have $|h_{\vec{a}}(o) - h_{\vec{a}}(q)| \leq \frac{wR}{2}$, which means that $o$ falls into the interval $[h_{\vec{a}}(q) - \frac{wR}{2}, h_{\vec{a}}(q) + \frac{wR}{2}]$ along the random line $\vec{a}$. This interval is simply the anchor bucket with width $wR$ defined by $h_{\vec{a}}(\cdot)$, which obviously contains the sub-interval $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$, i.e., $B^1$. And the width of $B^R$ is $R$ times the width of $B^1$ □

PROPOSITION 3. *Given $q$ and $w$, $B^{cR}$ contains $B^R$, and the width of $B^{cR}$ is $c$ times the width of $B^R$.*

Referring to Figure 5, on the random line $\vec{a_1}$, the interval of width $w$ centered at $q$ is $B^1$, which is indicated by "00". $B^2$ and $B^4$ are indicated by "1001" and "32100123", respectively. Virtual rehashing of QALSH is equal to symmetrically searching half-buckets of length $\frac{w}{2}$ one by one on both sides of $q$. A detailed example is given in Section 4.2.

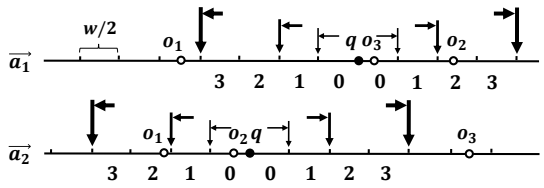



**Figure 5: Virtual Rehashing of QALSH for $c = 2$**

## 3.4 Preparing for Bucket Partition

Given a *query-aware* LSH function $h_{\vec{a}}$, to perform virtual rehashing along $\vec{a}$, we need to quickly locate a series of anchor buckets via *query-aware* bucket partition. Therefore, in the pre-processing step, we prepare the hash table $T$ of $h_{\vec{a}}$. $T$ is a list of the pairs $(h_{\vec{a}}(o), ID_o)$ for each object $o$ in the database $D$, where $ID_o$ is the object id referring to $o$. The list is sorted in ascending order of $h_{\vec{a}}(o)$, and is then indexed by a $B^+$-tree.

Given a pre-specified bucket width $w$. When a query $q$ arrives, to conduct an $(R, c)$-NN search, we perform a range search $[h_{\vec{a}}(q) - \frac{wR}{2}, h_{\vec{a}}(q) + \frac{wR}{2}]$ to locate the round-$R$ anchor bucket using the $B^+$-tree over the hash table $T$. To conduct an $c$-ANN search, we first perform a range search $[h_{\vec{a}}(q) - \frac{w}{2}, h_{\vec{a}}(q) + \frac{w}{2}]$ to locate the round-1 anchor bucket $B^1$. Then we use virtual rehashing to check both sides of $B^1$ to locate the round-$R$ anchor buckets in need. In this manner, we can implement *query-aware* bucket partition quickly, without physically partitioning the whole random line into buckets of width $w$.

Essentially, a hash table of QALSH can be viewed as a Secondary $B^+$-tree, which enables QALSH to support updates and to enhance the performance of relational databases.

# 4. QUERY-AWARE LSH SCHEME

Given a *query-aware* LSH function $h$, if a data object $o$ is close to a query $q$ in the original Euclidean space, then it is very likely they will collide in the anchor bucket with width $w$ defined by $h$. However, under a specific function, they may not collide at all. Therefore, QALSH exploits a collection of $m$ independent *query-aware* LSH functions to achieve quality guarantee. A good candidate $o$ for query answers is expected to collide with $q$ frequently under the $m$ functions. QALSH identifies final query answers from a collection of such candidates.

## 4.1 QALSH for (R, c)-NN Search

QALSH directly solves the $(R, c)$-NN problem by exploiting a base $\mathcal{B}$ of $m$ *query-aware* LSH functions $\{H^R_{\vec{a_1}}(\cdot), H^R_{\vec{a_2}}(\cdot), \ldots, H^R_{\vec{a_m}}(\cdot)\}$. Those LSH functions are mutually independent, and are uniformly selected from an $(R, cR, p_1, p_2)$-sensitive *query-aware* LSH family. For each $H^R_{\vec{a_i}}(\cdot)$, we build a hash table $T_i$ which is indexed by a $B^+$-tree, as described in Section 3.4.

To find the $(R, c)$-NN of a query $q$, we first compute the hash values $H^R_{\vec{a_i}}(q)$ for $i = 1, 2, \ldots, m$, and then use the $B^+$-trees over $T_i$s to locate the $m$ round-$R$ anchor buckets. For each object $o$ that appears in some of the $m$ anchor buckets, we collect its *collision number* $\#Col(o)$, which is formally defined as follows:

$$\#Col(o) = |\{H^R_{\vec{a}} \mid H^R_{\vec{a}} \in \mathcal{B} \wedge |H^R_{\vec{a}}(o) - H^R_{\vec{a}}(q)| \leq \frac{w}{2}\}| \quad (5)$$

Given a pre-specified collision threshold $l$, object $o$ is called *frequent* (with respect to $q$, $w$ and $\mathcal{B}$) if $\#Col(o) \geq l$. We prefer to collecting collision numbers first for objects whose projections are closer to the query projection. We only need to find the "first" $\beta n$ frequent objects (where $\beta$ is clarified later and $n$ is $D$'s cardinality) and compute the Euclidean distances to $q$ for them. If there is some frequent object whose distance to $q$ is less than or equal to $cR$, we return YES and the object; Otherwise, we return NO.

The base cardinality $m$ is one of the key parameters for QALSH, which need to be properly chosen so as to ensure that the following two properties hold at the same time with a constant probability:

- $\mathcal{P}_1$: If there exists an object $o$ whose distance to $q$ is within $R$, then $o$ is a frequent object.
- $\mathcal{P}_2$: The total number of *false positives* is less than $\beta n$, where each false positive is a *frequent* object whose distance to $q$ is larger than $cR$.

The above assertion is assured by Lemma 3 as follows, which guarantees correctness of QALSH for the $(R, c)$-NN search. Let $l$ be the collision threshold, $\alpha$ be the collision threshold in percentage, we have $l = \alpha m$. Let $\delta$ be the error probability, $\beta$ be the percentage of *false positives*.

LEMMA 3. *Given $p_1 = p(1)$ and $p_2 = p(c)$, where $p(\cdot)$ is defined by Equation 4. Let $\alpha, \beta$ and $\delta$ be defined as above. For $p_2 < \alpha < p_1$, $0 < \beta < 1$ and $0 < \delta < \frac{1}{2}$, $\mathcal{P}_1$ and $\mathcal{P}_2$ hold at the same time with probability at least $\frac{1}{2} - \delta$, provided the base cardinality $m$ is given as below:*

$$m = \left\lceil \max\left( \frac{1}{2(p_1 - \alpha)^2} \ln \frac{1}{\delta}, \frac{1}{2(\alpha - p_2)^2} \ln \frac{2}{\beta} \right) \right\rceil \quad (6)$$

Lemma 3 is a slightly different version of Lemma 1 of C2LSH in the sense that the joint probability of $\mathcal{P}_1$ and $\mathcal{P}_2$ is explicitly bounded from below. Therefore, we only give a sketch of the proof in Appendix A.

## 4.2 QALSH for c-ANN Search

Given a query $q$ and a pre-specified bucket width $w$, in order to find the $c$-ANN of $q$, QALSH first collects frequent objects from round-1 anchor buckets using $R = 1$; if frequent objects collected so far are not enough, QALSH *automatically* updates $R$, and hence collects more frequent objects from the round-$R$ anchor buckets via virtual rehashing, and etc., until finally enough frequent objects have been found or a *good enough* frequent object has been identified. The $c$-ANN of $q$ must be one of the frequent objects.

QALSH is quite straightforward, as shown in Algorithm 1. A candidate set $C$ is used to store the frequent objects found so far, and is empty at the beginning.

**Terminating condition.** QALSH terminates in one of the two following cases which are supported by the two properties $\mathcal{P}_1$ and $\mathcal{P}_2$ of Lemma 3 respectively:

- $\mathcal{T}_1$: At round-$R$, there exists at least 1 frequent object whose Euclidean distance to $q$ is less than or equal to $cR$ (referring to Lines 9 – 11 in Algorithm 1).
- $\mathcal{T}_2$: At round-$R$, at least $\beta n$ frequent objects have been found (referring to Line 2 and Line 13 in Algorithm 1).

**Algorithm 1** QALSH

**Input:**

$c$ is the approximation ratio, $\beta$ is the percentage of false positives, $\delta$ is the error probability. $m$ is the number of hash tables, $l$ is the collision threshold.

**Output:**

the nearest object $o_{min}$ in the set $C$ of frequent objects.

```
1: R = 1; C = ∅;
2: while |C| < βn do
3:    for each i = 1 to m do
4:       increase #Col(o) by 1 if o is found in the round-R
         anchor bucket, i.e., |H^R_{ā_i}(o) − H^R_{ā_i}(q)| ≤ w/2;
5:       if #Col(o) ≥ l then
6:          C = C ∪ o;
7:       end if
8:    end for
9:    if |{o | o ∈ C ∧ ‖o, q‖ ≤ c × R}| ≥ 1 then
10:      break;
11:   end if
12:   update radius R;
13: end while
14: return the nearest object o_min ∈ C;
```

**Update of Search Radius $R$.** It can be checked that, in Algorithm 1, if the terminating condition $\mathcal{T}_1$ is still not satisfied at the moment, i.e., we have not found a good enough frequent object, then we need to update $R$ in Line 12. For ease of reference, let $R$ and $R'$ denote current and next search radius, respectively.

Since C2LSH statically set $R' = c \times R$, it conducts one by one a series of $(R, c)$-NN search with $R \in \{1, c, c^2, ...\}$. Actually, some round of the search could be wasteful. Given $c = 2$ and $l = 2$, an example can be illustrated in Figure 5 and Algorithm 1. After Algorithm 1's first round (i.e., after the $(R = 1, c = 2)$-NN search), we have $\#Col(o_2) = 1$ and $\#Col(o_3) = 1$, since only $o_2$ and $o_3$ respectively appears once in the two round-1 anchor buckets labeled by "00". Since both $o_2$ and $o_3$ are not frequent at the moment, Algorithm 1 needs to update $R$. Since there is no new data object in round-2 anchor buckets which are labeled by "1001", we do not need to update $R$ to be $R = 2$ (i.e., $R = c$) as what C2LSH chooses to do.

In contrast, QALSH chooses to skip such wasteful rounds by leveraging the projections of data objects to properly update $R$. Recall that each of the $m$ hash tables of QALSH is simply a $B^+$-tree, we can easily find the object $o$ which is closest to $q$ and exists outside of the current round-$R$ anchor bucket. Thus we have $m$ such objects in total. Suppose their distances to $q$ (in terms of projections) are sorted in ascending order and denoted as $d_1$, $d_2$, ..., and $d_m$, i.e., $d_1$ is the smallest and $d_m$ is the biggest. Let $d_{med}$ denote the median of $d_1$, $d_2$, ..., and $d_m$. QALSH automatically set $R'$ to be $R' = c^k$ such that $\frac{wR'}{2} \geq d_{med}$ and integer $k$ is as small as possible. Therefore, there are at least $\frac{m}{2}$ objects for collecting collision number in the next round of search.

The underlying intuition is as follows. If we set $R'$ according to $d_1$, the round-$R'$ anchor buckets may contain too few data objects for collecting collision number and hence we waste the scan of the round-$R'$ anchor buckets. On the other hand, if we set $R'$ according to $d_m$, since $d_m$ may be too large, round-$R'$ anchor buckets may contain too many data objects, and hence we may do unnecessary collision number collection and Euclidean distance computation. In addition, $R'$ has to be $R' = c^k$ for integer $k$, so that the theoretical framework of QALSH is still assured.

## 4.3 QALSH for c-k-ANN Search

To support the $c$-$k$-ANN search, QALSH only needs to change its terminating conditions of $c$-ANN:

- $\mathcal{T}_1'$: At round-$R$, there exist at least $k$ frequent objects whose Euclidean distance to $q$ is within $cR$.
- $\mathcal{T}_2'$: At round-$R$, there are at least $\beta n + k - 1$ frequent objects that have been found.

# 5. THEORETICAL ANALYSIS

In this section, we first show that QALSH works with any approximation ratio $c > 1$, and then we give the bound on approximation ratio for $c$-ANN search. Then we discuss the parameter setting of QALSH and propose an automatic way to set bucket width $w$. Finally, we show the time and space complexity of QALSH.

## 5.1 Working with Any Approximation Ratio

C2LSH physically builds hash tables for search radius $R = 1$, where the buckets are called level-1 buckets and are statically partitioned before any query arrives. Each level-1 bucket is identified by an integer called *bid*. Referring to Observation 3 of C2LSH, when C2LSH performs virtual rehashing for search radius $R \in \{c, c^2, c^3, \ldots\}$, each level-$R$ bucket consists of exactly $R$ level-1 buckets identified by consecutive level-1 bids. When approximation ratio $c$ is not an integer, search radius $R$ is not an integer either, which implies some level-1 bucket must be further partitioned. However, the level-1 bucket has already been set up as the smallest granularity of bucket partition. Therefore, C2LSH only works with integer $c \geq 2$.

LSB-Forest suffers from the same problem of static bucket partition as C2LSH. Before $k$-dimensional objects are converted into Z-order values, grids must be imposed on the $k$ coordinates of the $k$-dimensional space. Each cell of the grids is equal to a bucket. A Z-order value virtually imposes grids at different levels. A high-level cell (bucket) is used for larger search radius and consists of an integral number of low-level cells (buckets) which are used for smaller search radius. Therefore, LSB-Forest also only works with integer $c \geq 2$. Now we show:

LEMMA 4. *Algorithm 1 works with any approximation ratio $c > 1$.*

PROOF. As shown in Algorithm 1, only anchor buckets at different rounds are needed. Instead of using a fixed bucket id to identify a pre-partitioned bucket, all the anchor buckets at different rounds are virtually imposed by specifying a bucket range, i.e., $|H_{\vec{a}}^R(o) - H_{\vec{a}}^R(q)| \leq \frac{w}{2}$. We can use any bucket range to decide an anchor bucket, since the hash values, i.e., the projections, have been recorded in the hash tables. According to Proposition 3, the enlargement of a round-$R$ anchor bucket by $c$ times, which generates a round-$cR$ anchor bucket, is realized by enlarging the corresponding bucket range by $c$ times, where $c$ is not required to be an integer. Therefore, Algorithm 1 works with any $c > 1$. □

## 5.2 Bound on Approximation Ratio

For $c$-ANN search, we now present the bound on approximation ratio for Algorithm 1.

THEOREM 1. *Algorithm 1 returns a $c^2$-approximate NN with probability at least $\frac{1}{2} - \delta$.*

Since both QALSH and C2LSH use the technique of virtual rehashing, this theorem is a stronger version of Theorem 1 of C2LSH in the sense that the probability in this theorem is explicitly bounded from below. This theorem simply follows from the combination of Theorem 1 of C2LSH and Lemma 3 of QALSH.

## 5.3 Parameter Settings

The accuracy of QALSH is controlled by error probability $\delta$, approximation ratio $c$ and false positive percentage $\beta$, where $\delta$, $c$ and $\beta$ are constants specified by users. $\delta$ controls the success rate of any LSH-based method for $c$-ANN search. In this paper, we set $\delta = \frac{1}{e}$. A smaller $c$ means a higher accuracy. Intuitively, a bigger $\beta$ allows C2LSH and QALSH to check more frequent objects, and hence enables them to achieve a better search quality, with higher costs in terms of random I/Os. Similar to C2LSH, QALSH sets $\beta = 100/n$ to restrict the number of random I/Os.

We now consider the base cardinality $m$, collision threshold percentage $\alpha$ and collision threshold $l$. Referring to Equation 6 of Lemma 3, let $m_1 = \left\lceil \frac{1}{2(p_1-\alpha)^2} \ln \frac{1}{\delta} \right\rceil$ and $m_2 = \left\lceil \frac{1}{2(\alpha-p_2)^2} \ln \frac{2}{\beta} \right\rceil$, we have $m = \max(m_1, m_2)$. Since $p_2 < \alpha < p_1$, $m_1$ increases monotonically with $\alpha$ and $m_2$ decreases monotonically with $\alpha$. Since $m = \max(m_1, m_2)$, $m$ is smallest when $m_1 = m_2$. Then, $\alpha$ can be determined by:

$$\alpha = \frac{\eta \cdot p_1 + p_2}{1 + \eta}, \text{ where } \eta = \sqrt{\frac{\ln \frac{2}{\beta}}{\ln \frac{1}{\delta}}} \tag{7}$$

Replacing $\alpha$ in $m_1$ by Equation 7, we have:

$$m = \left\lceil \frac{\left(\sqrt{\ln \frac{2}{\beta}} + \sqrt{\ln \frac{1}{\delta}}\right)^2}{2(p_1 - p_2)^2} \right\rceil \tag{8}$$

After setting the values of $m$ and $\alpha$, we compute the integer collision threshold $l$ as follows:

$$l = \lceil \alpha m \rceil \tag{9}$$

The base cardinality $m$ is simply the number of hash tables in QALSH. A small $m$ leads to small time and space overhead in QALSH, as shown in Section 5.4. However, $m$ must be set to satisfy the requirement of Lemma 3 for quality guarantee. It follows from Equation 8 that $m$ decreases monotonically with the difference $(p_1 - p_2)$ for fixed $\delta$ and $\beta$. From Section 3.2, we know there is a value of $w$ in the range $[0, 10]$ to maximize $(p_1 - p_2)$. Both E2LSH and LSB-Forest manually set bucket width $w = 4.0$, while C2LSH manually set $w = 1.0$. In the next section, we propose to automatically decide $w$ so as to minimize the base cardinality $m$.

### 5.3.1 Automatically Setting w by Minimizing m

The strategy of minimizing $m$ is to select the value of $w$ that maximizes the difference $(p_1 - p_2)$. Formally, we have Lemma 5 to minimize $m$.

LEMMA 5. *Suppose $\delta$ and $\beta$ are user-specified constants, for any approximation ratio $c > 1$, the base cardinality $m$ of QALSH is minimized by setting*

$$w = \sqrt{\frac{8c^2 \ln c}{c^2 - 1}} \tag{10}$$

PROOF. Let $\mu(w) = p_1 - p_2$. From Equation 4, we have:

$$\begin{aligned} \mu(w) &= p_1 - p_2 \\ &= \int_{-\frac{w}{2}}^{\frac{w}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}}\, dt - \int_{-\frac{w}{2c}}^{\frac{w}{2c}} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}}\, dt \\ &= \frac{2}{\sqrt{2\pi}} \int_{-\infty}^{\frac{w}{2}} e^{-\frac{t^2}{2}}\, dt - \frac{2}{\sqrt{2\pi}} \int_{-\infty}^{\frac{w}{2c}} e^{-\frac{t^2}{2}}\, dt \end{aligned}$$

Using the basic techniques of calculus, we take the derivative and obtain the following equation:

$$\mu'(w) = \frac{1}{\sqrt{2\pi}} \left(e^{-\frac{w^2}{8}} - \frac{1}{c} \cdot e^{-\frac{w^2}{8c^2}}\right)$$

Let $\mu'(w) = 0$. Since $w > 0$ and $c > 1$, we have the expression $w^* = \sqrt{\frac{8c^2 \ln c}{c^2 - 1}}$. When $0 < w < w^*$, $\mu'(w) > 0$ and when $w > w^*$, $\mu'(w) < 0$. Thus, $\mu(w)$ monotonically increases with $w$ for $0 < w < w^*$, and monotonically decreases with $w$ for $w > w^*$. Therefore, $\mu(w) = p_1 - p_2$ achieves its maximum value when $w = w^*$. From Equation 8, $m$ decreases monotonically with the difference $(p_1 - p_2)$ since $\beta$ and $\delta$ are constants. Thus, $m$ achieves its minimum value when $w = w^*$. Since Equation 8 is derived from Lemma 3, the minimum value of $m$ satisfies the quality guarantee. □

## 5.4 Time and Space Complexity

Since we set $\beta = \frac{100}{n}$, $\beta n$ is constant. From Equations 8 and 9, we have $m = O(\log n)$ and $l = O(\log n)$, respectively.

The time cost of QALSH consists of four parts: First, computing the projection of a query for $m$ hash tables costs $md = O(d \log n)$; Second, locating the $m$ round-1 anchor buckets in $B^+$-tree costs $m \log n = O((\log n)^2)$; Third, in the worst case, finding the frequent objects as candidates needs to do collision counting for all the $n$ objects over each hash table, which costs $ln = O(n \log n)$; Finally, calculating Euclidean distance for candidates costs $\beta nd = O(d)$. Therefore, the time complexity of QALSH is $O(d \log n + (\log n)^2 + n \log n + d) = O(d \log n + n \log n)$.

The space complexity of QALSH consists of two parts: the space of dataset $O(nd)$ and the space of index $mn = O(n \log n)$ for $m$ hash tables which store $n$ data objects' id and projection. Thus, the total space consumption of QALSH is $O(nd + n \log n)$.

# 6. EXPERIMENTS

In this section, we study the performance of QALSH using four real datasets. Since QALSH has quality guarantee and is designed for external memory, we take two state-of-the-art schemes of the same kind as the benchmark, namely, LSB-Forest and C2LSH.

## 6.1 Experiment Setup

### 6.1.1 Benchmark Methods

- **LSB-Forest**. LSB-Forest uses a set of $L$ LSB-Trees to achieve quality guarantee, which has a success probability at least $\frac{1}{2} - \frac{1}{e}$. LSB-Forest requires $2L$ buffer pages for $c$-ANN search. Since LSB-Forest has been shown to

outperform iDistance [8] and MEDRANK [3], they are omitted for comparison here.

- **C2LSH**. C2LSH is most related to QALSH. It requires a buffer of $2m$ pages for $c$-ANN search, where $m$ is the number of hash tables used in C2LSH. We consider C2LSH with $l$ as the collision threshold, as only under this case it has quality guarantee.

Our method is implemented in C++. All methods are compiled with gcc 4.8 with -O3. All experiments were done on a PC with Intel Core i7-2670M 2.20GHz CPU, 8 GB memory and 1 TB hard disk, running Linux 3.11.

### 6.1.2 Datasets and Queries

We use four real datasets in our experiments. We scale up values to integers as required by LSB-Forest and C2LSH, while QALSH is able to handle real numbers directly. We set page size $B$ according to what LSB-Forest requires for best performance.

- **Mnist**[2]. This 784-dimensional dataset has $60,000$ objects. We follow [15, 4] and consider the top-50 dimensions with the largest variance. $B$ is set to be 4KB.
- **Sift**[3] We use $1,000,000$ 128-dimensional base vectors of Sift as dataset. $B$ is set to be 4KB.
- **LabelMe**[4]. This 512-dimensional dataset has $181,093$ objects. The coordinates are normalized to be integers in a range of $[0, 58104]$. $B$ is set to be 8KB.
- **P53**[5]. The $5,408$-dimensional biological dataset in 2012 version has $31,420$ objects. We removed all objects that have missing values, so that the cardinality of the dataset is reduced to $31,159$. The coordinates are normalized to be integers in a range of $[0, 10000]$. $B$ is set to be 64KB.

Both LSB-Forest and C2LSH study the performance by averaging the query results of 50 random queries , while SRS uses 100 random queries. We conduct the experiments using three sets of queries, which,respectively, contain 50, 100, and 200 queries. Since the experimental results over the three query sets exhibit similar trends, we only report the results over the set of 100 queries due to space limitation. For the datasets Mnist and Sift, the queries are uniformly randomly chosen from their corresponding test sets. For the datasets LabelMe and P53, the queries are uniformly randomly chosen from the data objects. Mnist and Sift are regarded as low-dimensional datasets. LabelMe and P53 are regarded as medium- and high-dimensional datasets, respectively.

### 6.1.3 Evaluation Metrics

We use the following metrics for performance evaluation.

- **Index Size**. Since the size of datasets are constant for all methods, we use the size of the index generated by a method to evaluate the space overhead of the method.
- **Overall Ratio**. Overall ratio [15, 4] is used to measure the accuracy of a method. For the $c$-$k$-ANN search, it is defined as $\frac{1}{k}\sum_{i=1}^{k}\frac{\|o_i,q\|}{\|o_i^*,q\|}$, where $o_i$ is the $i$-th object returned by a method and $o_i^*$ is the true $i$-th nearest object, $i = 1, 2, ..., k$. Intuitively, a smaller overall ratio means a higher accuracy.
- **I/O Cost**. We follow LSB-Forest and C2LSH to use I/O cost to evaluate the efficiency of a method. It is defined as the number of pages to be accessed. I/O cost consists of two parts: the cost of finding candidates (i.e. frequent objects) and the cost of distance computation of candidates in the original space.
- **Running Time**. Since *query-aware* bucket partition introduces extra overhead, we also consider the running time cost for processing a query. It is defined as the wall-clock time for a method to solve the $c$-$k$-ANN problem.

[2] http://yann.lecun.com/exdb/mnist/
[3] http://corpus-texmex.irisa.fr/
[4] http://labelme.csail.mit.edu/inctructions.html
[5] http://archive.ics.uci.edu/ml/datasets/p53+Mutants

**Table 1: Index Size of QALSH vs. Bucket Width $w$**

| $w$ | Mnist | Sift | LabelMe | P53 |
|---|---|---|---|---|
| 1.000 | 49.6 MB | 1.0 GB | 163.8 MB | 68.6 MB |
| 2.000 | 19.1 MB | 388.6 MB | 63.7 MB | 26.5 MB |
| 2.719 | 16.5 MB | 336.0 MB | 54.6 MB | 23.1 MB |
| 3.000 | 16.8 MB | 344.1 MB | 56.1 MB | 23.5 MB |
| 4.000 | 23.2 MB | 473.6 MB | 77.4 MB | 32.2 MB |

**Table 2: Index Size of C2LSH vs. Bucket Width $w$**

| $w$ | Mnist | Sift | LabelMe | P53 |
|---|---|---|---|---|
| 1.000 | 61.2 MB | 1.2 GB | 435.2 MB | 83.5 MB |
| 2.000 | 29.9 MB | 597.7 MB | 193.1 MB | 41.6 MB |
| 2.184 | 29.5 MB | 589.6 MB | 188.5 MB | 41.0 MB |
| 3.000 | 33.1 MB | 669.4 MB | 197.5 MB | 45.4 MB |
| 4.000 | 46.1 MB | 945.6 MB | 258.3 MB | 62.1 MB |

## 6.2 Parameter Settings

For the sake of fairness, the success probability of all methods is set to $\frac{1}{2} - \frac{1}{e}$, i.e., $\delta$ of QALSH and C2LSH is set to $\frac{1}{e}$. We use setting $c = 2.0$, so that LSB-Forest and C2LSH can achieve their best performance. Both QALSH and C2LSH set false positive percentage $\beta$ to be $100/n$ to limit the number of candidates and hence the corresponding number of random I/Os. Other parameters of LSB-Forest and C2LSH are set to their default values [15, 4].

We compute bucket width $w$ for QALSH by Equation 10, and get $w = 2.719$ for $c = 2$. Since $w$ is manually set to 1.0 and 4.0 in C2LSH and LSB-Forest respectively, we also consider two intermediate values $w = 2.0$ and $w = 3.0$. Table 1 shows the index size of QALSH under the five settings of $w$. We observe that the index size under setting $w = 2.719$ is indeed the smallest. Since each hash table has the same size, the difference in index size reflects the difference in the number of hash tables, i.e., the base cardinality $m$. In other words, setting $w = 2.719$ minimizes $m$ among the five settings of $w$. We also evaluate the overall ratio, I/O cost and running time of QALSH under the five settings of $w$. We observe that the overall ratios under different settings are basically equal to each other. Due to the smallest index size under setting $w = 2.719$, both the I/O cost and running time under this setting are the smallest. Due to space limitation, we omit those results here.

Since the base cardinality $m$ of both QALSH and C2LSH is computed by Equation 8, we also automatically compute

**Table 3: Statistics of Index Size**

| | Mnist | Sift | LabelMe | P53 |
|---|---|---|---|---|
| $L$ | 55 | 354 | 213 | 102 |
| LSB-Forest | 858.1 MB | 246.3 GB | 106.6 GB | 69.4 GB |
| $m$ | 115 | 147 | 128 | 107 |
| C2LSH | 29.5 MB | 589.6 MB | 188.5 MB | 41.0 MB |
| $m$ | 65 | 83 | 72 | 61 |
| QALSH | 16.5 MB | 336.0 MB | 54.6 MB | 23.1 MB |

$w$ for C2LSH to minimize $m$ (or to maximize $(\xi_1 - \xi_2)$), and get $w = 2.184$ for $c = 2$. Table 2 shows the index size of C2LSH under the five settings of $w$. Interestingly, our experimental results show that C2LSH performs better under the setting $w = 2.184$ than $w = 1.0$, which is the default value of C2LSH [4]. Due to space limitation, we also omit the results here.

Our experiments demonstrate the effectiveness of automatically determining the bucket width $w$ by minimizing the base cardinality $m$. In the subsequent experiments, we only show the results of both QALSH and C2LSH with $w$ set to the automatically determined values. Specifically, we have $w = 2.719$ for $c = 2$ for QALSH, and $w = 2.184$ for $c = 2$ for C2LSH. Since the number of hash functions of LSB-Forest is not affected by $w$, we still use its manually set value $w = 4.0$.

## 6.3 Index Size and Indexing Time

We list the index sizes of all the three methods over the four datasets in Table 3, where $L$ is the number of LSB-Trees used by LSB-Forest, and $m$ is the number of hash tables used by C2LSH and QALSH. Each method needs $2m$ or $2L$ buffer pages for performing $c$-ANN search, in the experiments we set the number of buffer pages to be $2\max(m, L)$ so as to make LSB-Forest or C2LSH have enough buffer pages. Referring to Table 3, the $m$ value of QALSH is consistently smaller than that of C2LSH, and is also consistently smaller than the $L$ value of LSB-Forest except on the dataset Mnist. In other words, QALSH only needs a smaller number of buffer pages.

For each dataset, the index sizes of QALSH and C2LSH are smaller than the index size of LSB-Forest by about two or three orders of magnitude. LSB-Forest stores coordinates of objects and Z-order values in leaf pages in each LSB-Tree. Large data dimensionality $d$ leads to large overhead for storing coordinates. Moreover, each Z-order value has $uv$ bits where $u = O(\log_2 d)$ and $v = O(\log dn)$. In total, the index size of LSB-Forest grows at the rate of $O(d^{1.5}n^{1.5})$. Therefore, LSB-Forest incurs extremely large space overhead on high-dimensional datasets. In contrast, the index sizes of QALSH and C2LSH are independent of $d$. Meanwhile, QALSH and C2LSH only store object ids and projections in their hash tables, at the expense of using random I/Os to access coordinates for computing Euclidean distance. The index size of QALSH is about 29% to 57% of that of C2LSH. The difference between their index size is mainly due to the different number of hash tables needed by each method. Simpler *query-aware* LSH functions used by QALSH result in smaller number of hash tables.

The wall-clock time for building the index, i.e., the indexing time, is generally proportional to the index size. On every dataset, the indexing time of QALSH is the smallest while that of LSB-Forest is the largest. Specifically, on the dataset Sift with one million data objects, LSB-Forest takes more than 2.5 hours in building the index, and C2LSH takes about 3 minutes, while QALSH only takes about 50 seconds.

## 6.4 Overall Ratio

We evaluate the overall ratio for 2-$k$-ANN search by varying $k$ from 1 to 100. Results are shown in Figure 6.

All the methods get satisfactory overall ratios, which are much smaller than the theoretical bound $c^2 = 4$. Compared to LSB-Forest, QALSH and C2LSH achieve significantly higher accuracy. The overall ratios of QALSH and C2LSH are always smaller than 1.05, while the smallest overall ratio of LSB-Forest on the four datasets is still larger than 1.24. The overall ratios of QALSH are basically the same as those of C2LSH. This is because the parameters which affect accuracy are set to be the same for both methods.

As $k$ increases, the overall ratios of QALSH and C2LSH tend to increase while the overall ratio of LSB-Forest tends to decrease. In fact, both QALSH and C2LSH return the best $k$ objects out of a candidate set of size $\beta n + k - 1$. As $k$ increases, only $k - 1$ additional candidates are checked for possible improvement on the ratios. In contrast, LSB-Forest tends to check relatively more objects.

## 6.5 I/O Cost

We evaluate the I/O cost for 2-$k$-ANN search by varying $k$ from 1 to 100. The results[6] are shown in Figure 7.

Compared to QALSH and C2LSH, LSB-Forest requires much smaller I/O costs on low- and medium-dimensional datasets, i.e., Mnist, Sift and LabelMe. However, its overall ratio is much larger than those of QALSH and C2LSH. For the high-dimensional dataset P53, the I/O cost of QALSH is smaller than that of LSB-Forest. This is because the I/O cost of LSB-Forest monotonically increases as data dimensionality $d$ increases, while the I/O costs of QALSH and C2LSH are independent of $d$. Compared to C2LSH, QALSH uses about 49% to 76% of the I/O costs of C2LSH, but still achieves the same accuracy.

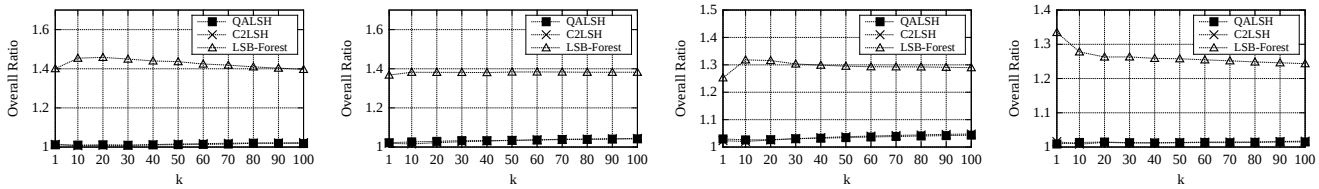
When $k$ increases, the I/O cost of LSB-Forest increases gently, while the I/O costs of QALSH and C2LSH increase more apparently. This is because LSB-Forest already stores the coordinates of objects in each LSB-Tree, and hence it computes the Euclidean distance without extra I/O costs. However, neither QALSH nor C2LSH stores the coordinates in hash tables, and thus one random I/O is needed for every candidate in the worst case. As $k$ increases, the number of candidates increases, and accordingly the I/O costs of both QALSH and C2LSH increase.

## 6.6 Running Time

We study the running time for 2-$k$-ANN search by varying $k$ from 1 to 100. The results are shown in Figure 8.

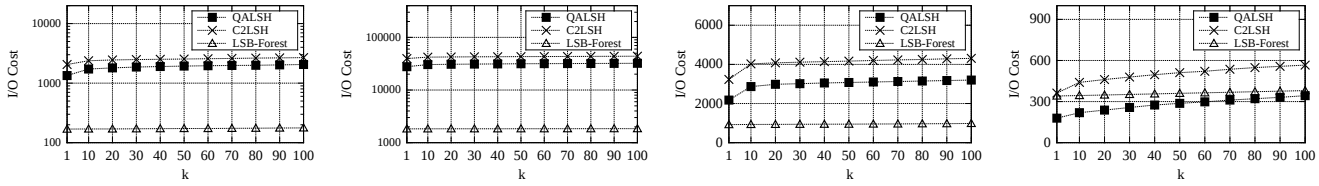
Interestingly, the running time of LSB-Forest is larger than that of QALSH on the medium-dimensional dataset LabelMe, even though its I/O cost is smaller than that of QALSH. While the I/O cost of LSB-Forest is slightly larger than that of QALSH on the high-dimensional dataset P53, the running time of LSB-Forest is surprisingly larger than that of QALSH by more than two orders of magnitude. In fact, as data dimensionality $d$ increases, LSB-Forest tends

[6]I/O costs of brute-force linear scan method over the datasets of Mnist, Sift, LabelMe and P53 are 3000, 125000, 45249 and 10353, respectively.




(a) Ratio on Mnist (b) Ratio on Sift (c) Ratio on LabelMe (d) Ratio on P53

**Figure 6: Overall Ratio of QALSH, C2LSH and LSB-Forest**




(a) I/O on Mnist (b) I/O on Sift (c) I/O on LabelMe (d) I/O on P53

**Figure 7: I/O Cost of QALSH, C2LSH and LSB-Forest**

to use more CPU time for finding the candidates whose Z-order values are closest to the Z-order value of the query. As already explained in Section 6.3, larger $d$ leads to longer Z-order values and hence leads to more time cost for processing Z-order values. It is worth mentioning that while QALSH is more efficient than LSB-forest on the medium- and high-dimensional datasets, it also achieves much higher searching accuracy than LSB-Forest.

The running time of QALSH is larger than that of LSB-Forest on the low-dimensional datasets, i.e., Mnist and Sift, but the searching accuracy of QALSH is much higher than that of LSB-Forest. Note that in this set of experiments, we set $c$ to 2.0 so that LSB-Forest can achieve the best performance. Actually, we can trade the accuracy of QALSH for efficiency by setting a larger $c$ value. More explanation will be given in Section 6.7.

The running time of QALSH is consistently smaller than that of C2LSH on all the four datasets. Although QALSH may use more time in locating anchor buckets, its I/O cost is significantly smaller than that of C2LSH as shown in Figure 7. As the I/O cost is the main overhead, the total running time of QALSH is smaller than that of C2LSH.

## 6.7 Performance vs. Approximation Ratio

We study how approximation ratio $c$ affects the performance of QALSH. Due to space limitation, we only show results on Mnist and P53 in Figure 9. We observe similar trends from the results on the other two datasets.

QALSH achieves better query quality with smaller $c$ value. From Figures 9(a) and 9(b), the overall ratio of QALSH decreases monotonically as $c$ decreases. When $c$ is set to 1.5, the overall ratio of QALSH is very close to 1.0, even for $k = 100$. This means, by using $c < 2.0$, QALSH is able to return extremely accurate results. Meanwhile, when $c$ is set to 3.0, the overall ratios of QALSH on both datasets are still smaller than 1.07.

From Figures 9(c) to 9(f), both the I/O cost and the running time of QALSH decrease monotonically as $c$ increases. Specifically, the I/O costs under setting $c = 3.0$ are about 25% and 50% of the I/O costs under setting $c = 1.5$ over the datasets Mnist and P53, respectively. Similar trends can be observed for the running time of QALSH. Therefore, under certain circumstances where the searching efficiency is a critical requirement, we can trade the accuracy of QALSH for efficiency by setting a larger $c$ value. For example, for the low-dimensional dataset Mnist, the running time of QALSH with $c = 3.0$ is comparable to the running time of LSB-Forest shown in Figure 8(a), but the overall ratio of QALSH with $c = 3.0$ is still much smaller than that of LSB-Forest.

## 6.8 QALSH vs. C2LSH

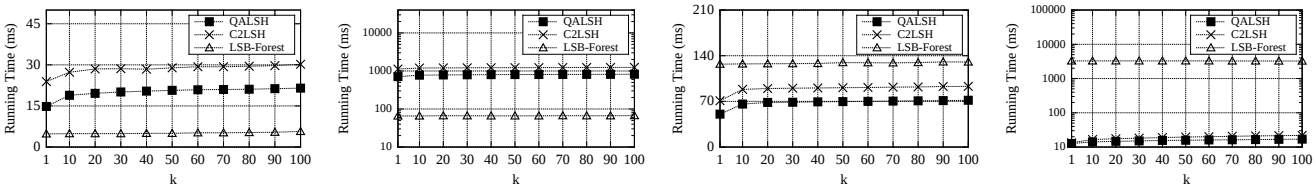
We study the performance of QALSH and C2LSH on the four datasets by setting $m$ and $l$ of C2LSH to be the same as those of QALSH. Due to space limitation, we only show results on Mnist and P53 in Figure 10. Similar trends are observed from the results on the other two datasets.

From Figures 10(a) and 10(b), the overall ratio of QALSH is much smaller than that of C2LSH. In fact, by setting the same values of $m$ and $\beta$, the maximum value of $(\xi_1 - \xi_2)$ is smaller than that of $(p_1 - p_2)$ according to Equation 8 as discussed in Section 3.2. Hence, the error probability $\delta$ of C2LSH is forced to increase. QALSH accordingly enjoys higher accuracy under the same $m$ and $l$. From Figures 10(c) to 10(f), QALSH also enjoys less I/O cost and running time. For the special case in P53, the running time of C2LSH is slightly less than that of QALSH when $k \leq 30$. This is because their I/O costs are close to each other, but QALSH needs more time in locating the anchor buckets.

We also study the performance of QALSH and C2LSH by setting $m$ and $l$ of QALSH to be the same as those of C2LSH, and observe similar trends from the results on the four datasets.

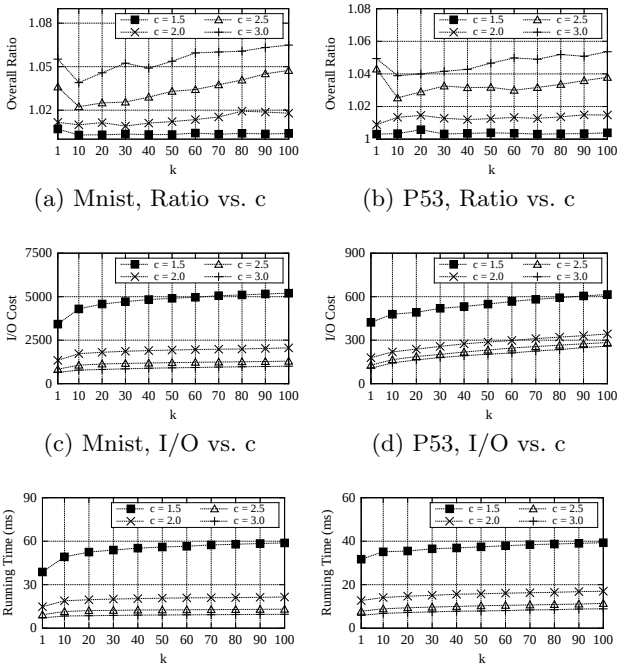
## 6.9 Summary

Based on the experiment results, we have the following findings. First, to achieve the same query quality, QALSH consumes much smaller space for index construction than C2LSH. In addition, QALSH is much more efficient than




(a) Time on Mnist (b) Time on Sift (c) Time on LabelMe (d) Time on P53

**Figure 8: Running Time of QALSH, C2LSH and LSB-Forest**

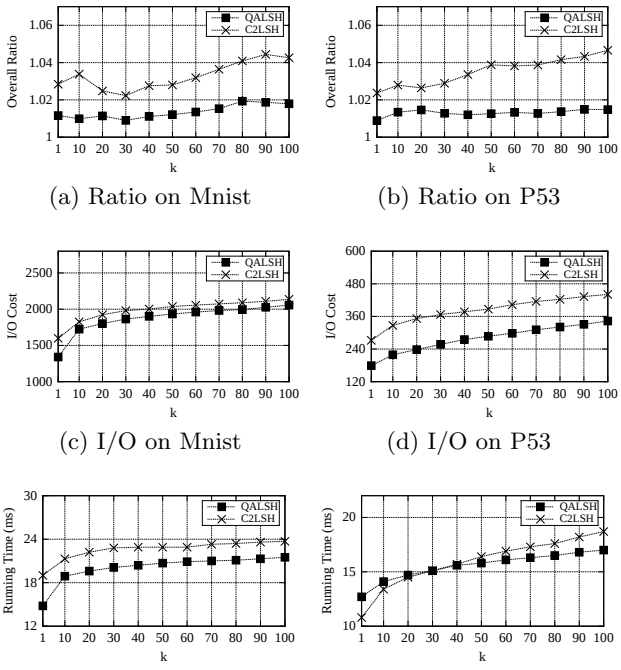



(a) Mnist, Ratio vs. c (b) P53, Ratio vs. c

(c) Mnist, I/O vs. c (d) P53, I/O vs. c

(e) Mnist, Time vs. c (f) P53, Time vs. c

**Figure 9: Performance of QALSH vs. $c$**




(a) Ratio on Mnist (b) Ratio on P53

(c) I/O on Mnist (d) I/O on P53

(e) Time on Mnist (f) Time on P53

**Figure 10: QALSH vs. C2LSH**

C2LSH since both its I/O cost and running time are much smaller than those of C2LSH. Second, when QALSH and C2LSH use the index of the same size, QALSH enjoys less I/O cost and running time, and achieves higher accuracy. Third, QALSH works with any $c > 1$. More accurate query results can be found by setting $c < 2.0$, at the expense of I/O. In contrast, LSB-Forest and C2LSH only work with integer $c \geq 2$. Finally, compared to LSB-Forest, QALSH uses much smaller index to achieve much higher accuracy, although it uses more I/O and running time on low- and medium-dimensional datasets. For high-dimensional datasets, QALSH outperforms LSB-Forest in terms of all the four evaluation metrics. This is because data dimensionality affects LSB-Forest. In general, data dimensionality affects any method depending on space-filling curve such as Z-order.

## 7. RELATED WORK

LSH functions are first introduced for use in Hamming space by Indyk and Motwani [7]. LSH functions based on $p$-stable distribution in Euclidean space are introduced by Datar et al.[2], which leads to E2LSH for processing memory dataset. E2LSH builds physical hash tables for a series of search radii, and hence results in a big consumption of storage space. One space saving alternative is to use a single "magic" radius to process different queries [5]. However, such a "magic" radius is hard to decide [15].

Virtual rehashing is implicitly or explicitly used in LSB-Forest [15] and C2LSH [4] to avoid building physical hash tables for each search radius. Virtual rehashing used in QALSH is much simpler and more effective than that of C2LSH due to the use of *query-aware* LSH function. Specifically, virtual rehashing of QALSH does not involve any random shift and floor function, and is carried out in a symmetrical manner. LSB-Forest, C2LSH and QALSH all have theoretical guarantee on query quality. Recently, a variant of LSB-Forest named SK-LSH [11] exploits linear order instead of Z-order for encoding hash values, without any theoretical guarantee on query quality.

An LSH function for Euclidean space, no matter *query-oblivious* or *query-aware*, involves random projection. Random projection is also used in MEDRANK [3] to project

objects over a set of $m$ random lines. However, MEDRANK does not segment a random line into buckets. An object that is found closest to a query along at least $\frac{m}{2}$ random lines, is reported as the $c$-ANN of the query. The median threshold of $\frac{m}{2}$ is generalized by collision threshold for finding frequent objects in both C2LSH and QALSH. A classic result on random projection is the Johnson-Lindenstrass Lemma [9], which states that by projecting objects in $d$-dimensional Euclidean space along $m$ random lines, the distance in the original $d$-dimensions can be approximately preserved in the $m$-dimensions. In a recent work on LSH for memory dataset in Euclidean space, Andoni et al.[1] propose to replace random projection (i.e., *data-oblivious* projection) by *data-aware* projection. However, the LSH scheme is still *query-oblivious*. Recently, Sun et al.[14] introduce another projection-based method named SRS. SRS uses only 6 random projections to convert high-dimensional data objects into low-dimensional ones so that they can be indexed by a single $R$-tree. While C2LSH has better overall ratio than SRS, SRS uses a rather small index and also incurs much less I/O cost. Since SRS exploits only 6 to 10 random projections, it is natural to expect one is able to perform several groups of such projections. However, it is not clear which group of projections in SRS would lead to the best overall ratio. Intuitively, SRS is less stable than C2LSH and QALSH, since SRS is based on less than 10 projections but C2LSH and QALSH take advantage of more projections.

# 8. CONCLUSIONS

In this paper, we introduce a novel concept of *query-aware* LSH function and accordingly propose a novel LSH scheme QALSH for $c$-ANN search in high-dimensional Euclidean space. A *query-aware* LSH function is a random projection coupled with *query-aware* bucket partition. The function needs no random shift that is a prerequisite of traditional LSH functions. *Query-aware* LSH functions also enables QALSH to work with any approximation ratio $c > 1$. In contrast, the state-of-the-art LSH schemes such as C2LSH and LSB-Forest only work with integer $c \geq 2$. Our theoretical analysis shows that QALSH achieves a quality guarantee for the $c$-ANN search. We also propose an automatic way to decide the bucket width $w$ used in QALSH. Experimental results on four real datasets demonstrate that QALSH outperforms C2LSH and LSB-Forest, especially in high-dimensional space.

# 9. ACKNOWLEDGMENTS

This work is partially supported by China NSF Grant 60970043, HKUST FSGRF13EG22 and FSGRF14EG31. We thank Wei Wang (UNSW) for his insightful comments.

# 10. REFERENCES

[1] A. Andoni, P. Indyk, H. L. Nguyen, and I. Razenshteyn. Beyond locality-sensitive hashing. In *SODA*, pages 1018–1028, 2014.

[2] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, pages 253–262, 2004.

[3] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *ACM SIGMOD*, pages 301–312, 2003.

[4] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD*, pages 541–552, 2012.

[5] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529. VLDB Endowment, 1999.

[6] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[7] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *ACM STOC*, pages 604–613, 1998.

[8] H. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: an adaptive b+-tree based indexing method for nearest neighbor search. *ACM TODS*, 30(2):364–397, 2005.

[9] W. Johnson and J. Lindenstrauss. Extensions of lipshitz mapping into hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

[10] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *ACM STOC*, pages 599–608, 1997.

[11] Y. Liu, J. Cui, Z. Huang, H. Li, and H. T. Shen. Sk-lsh: An efficient index structure for approximate nearest neighbor search. *VLDB*, 7(9), 2014.

[12] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *ACM-SIAM SODA*, pages 1186–1195, 2006.

[13] H. Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

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

[15] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM TODS*, 35(3):20, 2010.

# APPENDIX

# A. PROOF OF LEMMA 3

PROOF. Before bounding $Pr[\mathcal{P}_1 \cap \mathcal{P}_2]$ from below and hence proving Lemma 3, we have to prove lower bounds on $\mathcal{P}_1$ and $\mathcal{P}_2$.

We now show some details of proving $Pr[\mathcal{P}_1] \geq 1 - \delta$. Let $S_1 = \{o \mid \|o - q\| \leq R\}$. For $\forall o \in S_1$, $Pr[\mathcal{P}_1] = Pr[\#Col(o) \geq \alpha m] = \sum_{i=\lceil \alpha m \rceil}^{m} C_m^i p^i (1-p)^{m-i}$, where $p = Pr[|H_{a_j}^R(o) - H_{a_j}^R(q)| \leq \frac{w}{2}] \geq p_1 > \alpha$, $j = 1, 2, ..., m$. Then by following the same reasoning based on Hoeffding's Inequality [6] from Lemma 1 of C2LSH, we have $Pr[\mathcal{P}_1] \geq 1 - \delta$, when $m = \left\lceil \max\left(\frac{1}{2(p_1-\alpha)^2} \ln\frac{1}{\delta}, \frac{1}{2(\alpha-p_2)^2} \ln\frac{2}{\beta}\right)\right\rceil$.

Similarly, using the same $m$, we have $Pr[\mathcal{P}_2] > \frac{1}{2}$.

For the $(R, c)$-NN search, since QALSH terminates when either $\mathcal{P}_1$ or $\mathcal{P}_2$ holds, we have $Pr[\mathcal{P}_1 \cup \mathcal{P}_2] = 1$. We also have the formula: $Pr[\mathcal{P}_1 \cup \mathcal{P}_2] = Pr[\mathcal{P}_1] + Pr[\mathcal{P}_2] - Pr[\mathcal{P}_1 \cap \mathcal{P}_2]$. Therefore, we can bound $Pr[\mathcal{P}_1 \cap \mathcal{P}_2]$ from below as follows:

$$\begin{aligned} Pr[\mathcal{P}_1 \cap \mathcal{P}_2] &= Pr[\mathcal{P}_1] + Pr[\mathcal{P}_2] - Pr[\mathcal{P}_1 \cup \mathcal{P}_2] \\ &\geq 1 - \delta + \tfrac{1}{2} - 1 = \tfrac{1}{2} - \delta \end{aligned}$$

And hence Lemma 3 is proved. □