Locality-Sensitive Hashing Scheme Based on Dynamic Collision Counting

Junhao Gan Jianlin Feng
School of Software
Sun Yat-Sen University
Guangzhou, China
junhogan@gmail.com
fengjlin@mail.sysu.edu.cn

Qiong Fang Wilfred Ng
Dept of Computer Science and Engineering
Hong Kong University of Science and Technology
HongKong, China
{fang, wilfred}@cse.ust.hk

ABSTRACT

Locality-Sensitive Hashing (LSH) and its variants are well-known methods for solving the c-approximate NN Search problem in high-dimensional space. Traditionally, several LSH functions are concatenated to form a "static" compound hash function for building a hash table. In this paper, we propose to use a base of m single LSH functions to construct "*dynamic*" compound hash functions, and define a new LSH scheme called *Collision Counting LSH* (C2LSH). If the number of LSH functions under which a data object o collides with a query object q is greater than a pre-specified collision threhold l, then o can be regarded as a good candidate of c-approximate NN of q. This is the basic idea of C2LSH.

Our theoretical studies show that, by appropriately choosing the size of LSH function base m and the collision threshold l, C2LSH can have a guarantee on query quality. Notably, the parameter m is not affected by dimensionality of data objects, which makes C2LSH especially good for high dimensional NN search. The experimental studies based on synthetic datasets and four real datasets have shown that C2LSH outperforms the state of the art method LSB-forest in high dimensional space.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing Methods

Keywords

Locality Sensitive Hashing, Dynamic Collision Counting

1. INTRODUCTION

The problem of finding the nearest neighbors (NN) in the Euclidean space has its wide applications in various fields, such as artificial intelligence, information retrieval, pattern recognition and so on. To solve the nearest neighbor search (NNS) problem, many methods have been proposed like R-tree [6], and K-D tree [1]. Given a query object, these methods all return the exact results, that are the data objects closest to the query according to some distance functions. However, as the dimensionality of data objects increases, the efficiency of these methods greatly decreases. When the dimensionality is larger than 10, they even become slower than the brute-force linear-scan approach [12, 13].

Due to the difficulty in finding an efficient method for exact NNS in high-dimensional Euclidean space, an alternative problem, called *c-approximate Nearest Neighbor search* (c-approximate NN search), has been widely studied [12, 2, 8, 10, 11, 13]. Formally, the goal of c-approximate NNS is to find the data object(s) within the distance $c \times R$ from a query object, where R is the distance between the query and its true nearest neighbor.

Locality-Sensitive Hashing (LSH) [2, 8] and its variants [11, 13] are well-known methods for solving the c-approximate NNS problem in high-dimensional space. The LSH scheme was first proposed by Indyk et al. [8] for use in binary Hamming space $\{0, 1\}^d$, and later was extended for use in Euclidean space R^d by Datar et al [2], which leads to the E2LSH package [1]. The LSH scheme makes use of a set of "distance-preserving" hash functions, also called *LSH functions*, to cluster "closer" objects into the same bucket with higher probabilities. Currently, the primary choice of constructing an LSH function for Euclidean distance is to project data objects (represented as vectors $\vec{o}$ in R^d) along a randomly chosen line (identified by a random vector $\vec{a}$) which is segmented into equi-width intervals with size w, and then data objects projected to the same interval are viewed as "colliding" in the hashing scheme, i.e., each interval is taken as a bucket. Formally, an LSH function has the form $h_{\vec{a},b}(o) = \lfloor \frac{\vec{a}\cdot\vec{o}+b}{w} \rfloor$ where b is a real number chosen uniformly from $[0, w]$.

The E2LSH exploits LSH functions in the following way: First, a set of k LSH functions $h_1, \ldots, h_k$ are randomly chosen from an LSH function family, and then they are concatenated to form a compound hash function $G(\cdot)$, i.e., $G(o) = (h_1(o), \ldots, h_k(o))$ for any object o. Then, the hash function $G(\cdot)$ is adopted to hash all the data objects into a hash table T. By using a compound hash function $G(\cdot)$ instead of a single LSH function h_i, the probability that two "distant" data objects may collide can be largely reduced. All the LSH variants, including the *Multi-Probe LSH* [11] and the LSB-tree/LSB-forest [13], follow the above ap-

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

proach of using compound hash functions. However, By using a compound hash function, the probability that two "close" data objects fall in the same bucket is also reduced, although with a small extent. To increase the "colliding" probability of two "close" data objects, these methods tend to use a relatively big w which is the interval size in an LSH function. For example, the E2LSH uses the value $w = 4$. In fact, to reduce the "colliding" probability of two "distant" data objects, we can reduce the interval size w and thus remove the need of using compound hash functions.

Note that the compound hash functions $G(\cdot)$ are "*static*" in the sense that, once a compound hash function is designed based on a set of k randomly-chosen LSH functions, this compound hash function is applied to all the data objects to construct the corresponding hash table. It is always difficult to design good compound hash functions such that every pair of "close" objects can fall in the same bucket at least in one hash table. This is why usually more than one hundred, and sometimes up to several hundred compound hash functions and their corresponding hash tables are needed in the E2LSH method to guarantee a good search accuracy. In this paper, we propose to make use of "*dynamic*" compound hash functions for c-approximate NN search, and define a new LSH scheme, called *Collision Counting LSH* (C2LSH).

The C2LSH method first randomly chooses a set of m LSH functions with appropriately small interval size w (say, $w = 1$), which form a function base, denoted as $\mathcal{B}$ with $|\mathcal{B}| = m$. Intuitively, if a data object o is close to a query object q, then the two objects are very likely to collide under every single LSH function in $\mathcal{B}$. Accordingly, o should collide with q under a large number of LSH functions. Only data objects with large enough collison counts need to have their distances computed. More formally, by properly setting a *collision threshold* l, if a data object o collides with a query object q under at least l LSH functions in $\mathcal{B}$, then it is a good candidate of being the c-approximate NN of q. This is actually the principle for designing compound hash functions. Given a query q, two different data objects i and j can both collide with q under l LSH functions, but they may collide with q under two different sets of l LSH functions. If two compound hash functions are designed respectively based on these two sets of l LSH functions, i and j will collide with q in the corresponding hash tables, and both of them can be identified as good candidates of being the c-approximate NN for q. However, the two compound hash functions are usually not known in advance. In this paper, we seek to construct "dynamic" compound hash functions based on a base $\mathcal{B}$ of LSH functions, in order to find at least one good compound hash function for every data object that is close to a query object. In essence, we consider $\binom{m}{l}$ "dynamic" compound hash functions that are the concatenations of every possible set of l LSH functions in $\mathcal{B}$. Interestingly, we only need to physically build hash tables for each single LSH function in the base $\mathcal{B}$.

Our theoretical studies show that, by appropriately choosing the cardinality m of the LSH function base $\mathcal{B}$ and the collision threshold l, C2LSH can have a guarantee on query quality. Notably, the parameter m is not affected by dimensionality of data objects, which makes C2LSH especially good for high dimensional NN search. The experimental studies based on synthetic datasets and four real datasets have shown that C2LSH outperforms the state of the art method LSB-forest in high dimensional space.

The rest of this paper is organized as follows. We first discuss preliminaries in Section 2, and then introduce C2LSH in Section 3. The theoretical analysis of C2LSH is given in Section 4. We describe experimental studies in Section 5. Related work is discussed in Section 6. Finally, we conclude our work in Section 7.

2. PRELIMINARIES

2.1 Problem Settings

In this paper we consider the *c-approximate NN search* problem and its k-NN version in Euclidean space. Let D be the database of n data objects in d-dimensional Euclidean space R^d and let $\|o_1, o_2\|$ denote the Euclidean distance between two objects o_1 and o_2. Formally, a data object o is a c-approximate NN of a query object q if the distance between o and q is at most c times the distance between q and its exact NN o^*, i.e., $\|o, q\| \leq c\|o^*, q\|$, where $c \geq 1$ is the *approximation ratio*. The c-approximate NN problem is to find a data object that is a c-approximate NN of a query object q. Similarly, the c-approximate k-NN problem is to find k data objects that are respectively the c-approximation of the exact k-NN objects of q.

2.2 Locality-Sensitive Hashing Functions

Our method depends on locality-sensitive hashing (LSH) functions and hence we review the notion of LSH functions first. LSH functions are hash functions that can hash closer objects into the same bucket with higher probability. Let the *ball* of radius r centered at a data object $o \in D$ be defined as $B(o, r) = \{q \in R^d | \, \|o, q\| \leq r\}$. Then, an LSH family can be formally defined as follows [2].

Definition 1. An LSH function family $\mathcal{H} = \{h : R^d \to U\}$ is called (r, cr, p_1, p_2)-sensitive if for any $v, q \in R^d$
- if $v \in B(q, r)$, then $Pr_{\mathcal{H}}[h(v) = h(q)] \geq p_1$;
- if $v \notin B(q, cr)$, then $Pr_{\mathcal{H}}[h(v) = h(q)] \leq p_2$.

An interesting LSH family for Euclidean distance consists of LSH functions in the following form [2]:

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

Here $\vec{o}$ is the vector representation of a data object $o \in R^d$, $\vec{a}$ is a d-dimensional vector where each entry is drawn independently from the standard normal distribution $N(0, 1)$. w is a user-specified constant, and b is a real number uniformly randomly drawn from $[0, w)$.

Intuitively, an LSH function $h_{\vec{a},b}(o)$ works in the following way. It first projects the object o onto a line L_a whose direction is identified by $\vec{a}$. Then, the projection of o is shifted by a constant b. With the line L_a being segmented into intervals with size w, the hash function returns the number of the interval that contains the shifted projection of o.

For two data objects o_1 and o_2, let $s = \|o_1, o_2\|$. The probability that o_1 and o_2 collide under a uniformly randomly chosen hash function $h_{\vec{a},b}$, denoted as $p(s)$, can be computed as follows [2]:

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

where $f_2(x) = \frac{2}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$.

For a fixed w, the collision probability $p(s)$ decreases monotonically with s. Hence, the family of hash functions $h_{\vec{a},b}$ is (r, cr, p_1, p_2)-sensitive with $p_1 = p(r)$ and $p_2 = p(cr)$. When r is set to 1, the function family is $(1, c, p_1, p_2)$-sensitive with $p_1 = p(1)$ and $p_2 = p(c)$.

2.3 The E2LSH Method

The E2LSH method was proposed in [8, 2]. It does not solve the c-approximate NN problem directly. Instead, it tackles the (R, c)-NN problem, which is a decision version of the c-approximate NN problem. Given a query object q and a distance R, the (R, c)-NN problem is formally defined as follows:

(1) A data object o_1 within $B(q, cR)$ is returned, if there exists a data object o_2 within $B(q, R)$;

(2) no object is returned, if $B(q, cR)$ does not contain any data object in the database D.

To find the (R, c)-NN of a query object q, the E2LSH method uses an (R, cR, p_1, p_2)-sensitive LSH function family $\mathcal{H}$. First, a set of k LSH functions $h_1, \ldots, h_k$ are randomly chosen from $\mathcal{H}$, and they are concatenated to form a compound hash function $G(\cdot)$, with $G(o) = (h_1(o), \ldots, h_k(o))$ for any object o. Then, the hash function $G(\cdot)$ is used to hash all the data objects into a hash table T. The above two steps are repeated for L times, and accordingly, L compound hash functions $G_1(\cdot), \ldots, G_L(\cdot)$ and L hash tables are generated.

Given a query object q, the E2LSH method checks $3L$ data objects that collide with q under at least one of the L compound hash functions. If there exists a data object o such that the distance between o and q is smaller than or equal to cR, object o is returned as the (R, c)-NN of q; otherwise, no data object is returned even though there may indeed exist an (R, c)-NN of q. In other words, the E2LSH method has a nonzero error probability. Suppose the upper bound of the error probability is denoted as δ. The parameters k and L must be chosen to ensure that the following two properties be held with a constant probability at least $\frac{1}{2}$.

- $\mathcal{P}_1$: If there exists a data object $o \in B(q, R)$, there must exist at least one $G_i(\cdot)$ under which o and q collide.
- $\mathcal{P}_2$: The total number of data objects, which collide with q but their distances to q are greater than cR, is less than $3L$.

Note that only when both properties hold at the same time, the E2LSH method is sound for solving the (R, c)-NN problem.

The c-approximate NN problem can be solved by issuing a series of (R, c)-NN search with increasing radius R. Accordingly, we have to build hash tables in advance by varying R to be $\{1, c, c^2, c^3, \ldots\}$, which leads to extremely large space consumption and expensive query cost. Gionis et al.[4] proposed a heuristic method to tackle the large space consumption problem by adopting a single "magic" radius r_m to process different query objects. However, as Tao et al. [13] show that, the "magic" radius r_m may not exist at all. Instead, they propose the LSB-tree/LSB-forest methods to avoid building hash tables at different radii.

2.4 The LSB-tree/LSB-forest Methods

The construction of an LSB-tree consists of the following two steps:

- First, each data object $o \in R^d$ is converted to a k-dimensional data object $G(o)$, using a compound hash function $G(\cdot) = (h_1(\cdot), \ldots, h_k(\cdot))$, where $h_i(\cdot)$ is an (r, cr, p_1, p_2)-sensitive LSH function [13].
- Then, each $G(o)$ is converted to a Z-order value $z(o)$, and a conventional B-tree is built over all the $z(o)$ values.

The resultant B-tree is called an LSB-tree. To achieve a theoretical guarantee for search accuracy, an LSB-forest structure is proposed, which consists of L independent LSB-trees with $L = \sqrt{dn/B}$. Here B is the size of page for storing Z-order values and data coordinates in external memory.

The basic idea of the LSB-tree/LSB-forest methods is that, "close" data objects tend to have "close" Z-order values. The "closeness" between two Z-order values is captured by the notion of *Length of the Longest Common Prefix* (LLCP). For a query object q, the LSB-forest method first converts it to a Z-order value $z(q)$, and then uses $z(q)$ to search the LSB-forest. The Z-order values stored in leaf pages of all L LSB-trees are visited in decreasing order of their LLCP with $z(q)$. Intuitively, visiting Z-order values $z(o)$ in decreasing order of their LLCP with $z(q)$ simulates the process of issuing a series of (R, c)-NN search with increasing radius R. In this paper, our C2LSH method exploits a different way to realize the same simulation.

3. COLLISION COUNTING LSH (C2LSH)

In this section we propose *Collision Counting LSH* (or simply C2LSH) for both (R, c)-NN search and c-approximate NN search.

For an (R, c)-NN search, C2LSH exploits only a single base $\mathcal{B}$ of m LSH functions $\{h_1, \ldots, h_m\}$, where each h_i is randomly selected from an (R, cR, p_1, p_2)-sensitive LSH family. Here m is called the *base cardinality* of $\mathcal{B}$. When $h_i(\cdot)$ is used to build a hash table T_i, each data object o in the database D is hashed by $h_i(\cdot)$ to an integer, i.e., $h_i(o)$, which is taken as the bucket ID (or simply *bid*) of o. Then, all the data objects are sorted in increasing order of their bids along the real line. In other words, each hash table T_i is indeed a sorted list of buckets, and each bucket contains a set of object IDs representing the objects that fall in the bucket. If two objects are hashed by $h_i(\cdot)$ into the same bucket, we say they collide with each other under $h_i(\cdot)$. To search NN for a query object q, C2LSH only considers distance computation for data objects that collide with q under a large enough number of functions in $\mathcal{B}$.

For a c-approximate NN search, C2LSH exploits a series of LSH function bases $\mathcal{B}_i$ in order to simulate that E2LSH issues a series of (R, c)-NN search with increasing radius $R = \{1, c, c^2, c^3, \ldots\}$. We only need to materialize m hash tables which correspond to the m $(1, c, p_1, p_2)$-sensitive functions in the *initial base* $\mathcal{B}_1$. By carefully deriving new LSH functions from those in $\mathcal{B}_1$ to form each subsequent base $\mathcal{B}$, we can do *virtual rehashing* (referring to Section 3.4) without physically building $\mathcal{B}$'s corresponding hash tables.

In the following, we will illustrate in detail how the C2LSH method solves the (R, c)-NN problem and the c-approximate NN problem.

For ease of discussion, we first introduce the concept of *collision number* and then describe the details of the LSH functions used in C2LSH.

3.1 Collision Number and Frequent Object

The collision number of a data object o with respect to a query q and a base $\mathcal{B}$ is the number of LSH functions in $\mathcal{B}$ that hash o and q into the same bucket. Formally, let $\#collision(o, q, \mathcal{B})$ (or simply, $\#collision(o)$) denote the collision number of o with respect to q and $\mathcal{B}$, and it is defined as follows:

$$\#collision(o) = |\{h | h \in \mathcal{B} \wedge h(o) = h(q)\}|. \qquad (2)$$

For each h_i ($i = 1, ..., m$), o and q collide with a probability $p(\|o, q\|)$. A data object o is called *frequent* (with respect to q and $\mathcal{B}$) if its collision number $\#collision(o)$ is greater than or equal to a pre-specified *collision threshold* l.

To decide if a data object o is frequent, logically we need to perform a scan of the base $\mathcal{B}$. Of course we can choose to stop the scan whenever we have collected enough collisions. This collision counting procedure of o can be viewed as a dynamic formation of a good compound hash function $G(\cdot)$ which only consists of those LSH functions that put o and q into the same bucket.

3.2 LSH Functions for C2LSH

The LSH functions used in C2LSH depend on an observation that has been proved in [13].

Observation 1. *Given any integer $x \geq 1$, hash function $h'(o) = \lfloor \frac{\vec{a} \cdot \vec{o} + bwx}{w} \rfloor$ is $(1, c, p_1, p_2)$-sensitive, where $\vec{a}, b$ and w are the same as defined in Equation* (1).

From Observation 1, it is trivial to check that

$$h(o) = \lfloor \frac{\vec{a} \cdot \vec{o} + b^*}{w} \rfloor \qquad (3)$$

is $(1, c, p_1, p_2)$-sensitive, where b^* is uniformly drawn from $[0, c^{\lceil \log_c td \rceil} w^2]$, c is the approximation ratio, t is the largest coordinate value of data objects in the original space R^d and d denotes the dimensionality of data objects.

We use m $(1, c, p_1, p_2)$-sensitive functions $h(\cdot)$ as defined in Equation (3) to form the initial base $\mathcal{B}_1$, and we can derive (R, cR, p_1, p_2)-sensitive functions from these $h(\cdot)$'s based on the following observation that will be proved in Section 4:

Observation 2. *The hash function*

$$H^R(o) = \lfloor \frac{h(o)}{R} \rfloor$$

is (R, cR, p_1, p_2)-sensitive, where R is an integer power of c and $R \leq c^{\lceil \log_c td \rceil}$, $c, p_1, p_2, h(\cdot), t$ and d are the same as defined in Equation (3).

Buckets defined by an $H^R(\cdot)$ function are called level-R buckets. Specifically, $h(\cdot)$'s in the initial base $\mathcal{B}_1$ are simply $H^1(\cdot)$ functions, and buckets defined by an $h(\cdot)$ are level-1 buckets. Accordingly, when an object o is hashed to the level-R bucket identified by the integer $H^R(o)$, we call the bucket ID $H^R(o)$ a *level-R bid*, and hence o's level-1 bid is $H^1(o)$, i.e., $h(o)$.

From the definition of $H^R(\cdot)$ function, for a given integer x, it is easy to see the R consecutive level-1 bids $xR, xR+1, xR+2, \ldots, xR+(R-1)$ are mapped to the same level-R bid x. In other words, logically each level-R bucket x consists of R consecutive level-1 buckets, and multiplying the level-R bid x by R, we will get the level-1 bid xR, which identifies the leftmost level-1 bucket (along the real line) of the level-R bucket x. However, in practice, some of the level-1 buckets may be empty, and hence a level-R bucket may physically correspond to less than R level-1 buckets. In summary, we have the following observation of $H^R(\cdot)$ function which leads to our *Virtual Rehashing* technique for solving the (R, c)-NN problem without physically building hash tables at different radii.

Observation 3. *An object o's level-R bucket identified by the level-R bid $\lfloor \frac{h(o)}{R} \rfloor$ consists of R consecutive level-1 buckets identified by the level-1 bids $\lfloor \frac{h(o)}{R} \rfloor * R$, $\lfloor \frac{h(o)}{R} \rfloor * R + 1, \ldots, \lfloor \frac{h(o)}{R} \rfloor * R + (R-1)$.*

Interval Size w. For the efficiency of C2LSH, we should only do collision counting for data objects which are most promising to be an NN of a given query object q. Intuitively, data objects colliding with a query object q are most promising ones. However, the bucket that q falls in may contain too many data objects, hence we should exploit a small interval size w such as $w = 1$ in the $h(\cdot)$'s as defined in Equation (3) in order to reduce the number of data objects colliding with q in level-1 buckets.

3.3 C2LSH for (R, c)-NN Search

To find the (R, c)-NN of a query object q, C2LSH uses a (R, cR, p_1, p_2)-sensitive family $\mathcal{H}$ of LSH functions h's. From the family we randomly select m LSH functions $\{h_1, \ldots, h_m\}$ to construct an LSH function base $\mathcal{B}$. To process the query q, we first locate the buckets that q falls in by computing $h_i(q)$ for $i = 1, ..., m$ and hence find the union of data objects colliding with q. Intuitively, for every data object o we can compute its collision number $\#collision(o)$ and hence identify the set C of all the frequent objects. If C has less than βn (where β is specified later and n is the cardinality of the database D) frequent objects, we compute real distance for each member of C; otherwise, we only need to identify the "first" βn frequent objects and compute real distances for them. In eithe case, if some frequent object is within distance cR from q, then we return YES and the object; otherwise, we return NO.

Let α denote the collision threshold percentage, δ denote the error probability, β denote the allowable percentage of *false positives* which are frequent objects whose distance to q is greater than cR, and let l denote the collision threshold where $l = \alpha m$. The parameter m must be accordingly chosen so as to ensure that with a constant probability at least $\frac{1}{2}$ the following two properties hold:

$\mathcal{P}_1$: If there exists a data object o whose distance to q is at most R, i.e., $o \in B(q, R)$, then o's collision number is at least l. In other words, o is frequent if o is within distance R from q.

$\mathcal{P}_2$: The total number of false positives is less than βn.

Note that if the two properties $\mathcal{P}_1$ and $\mathcal{P}_2$ hold at the same time, then the C2LSH method is correct for solving the (R, c)-NN problem.

The following lemma guarantees that for specific parameters, m can be chosen to ensure that $\mathcal{P}_1$ and $\mathcal{P}_2$ hold with a constant probability.

LEMMA 1. *Given a collision threshold percentage $p_2 < \alpha < p_1$, where p_1 and p_2 are the same as defined in Equation (3), a false positive percentage $0 < \beta < 1$, and an error probability $0 < \delta < \frac{1}{2}$, if the base cardinality m, satisfies:*

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

we have $Pr[\mathcal{P}_1] \geq 1-\delta$ and $Pr[\mathcal{P}_2] > \frac{1}{2}$.

PROOF. The proof is in the Appendix A. □

3.3.1 *Parameters for C2LSH*

We now discuss the setting of the parameters for Lemma 1. We set $w = 1$, and with a given approximation ratio c, we can then compute p_1 and p_2 respectively by $p_1 = p(1)$ and $p_1 = p(c)$, where $p(s)$ is the collision probability function as defined in Section 2. We set $\delta = 0.01$. At this moment, we have three parameters β, α and m left for setting. We manually set $\beta = v/n$, where v is a positive integer that is much smaller than the database cardinality n, and thus we have $0 < \beta < 1$. Let $m_1 = \lceil \frac{1}{2(p_1-\alpha)^2}\ln\frac{1}{\delta}\rceil$ and $m_2 = \lceil \frac{1}{2(\alpha-p_2)^2}\ln\frac{2}{\beta}\rceil$, and by letting $m_1 = m_2$, we can decide α by the following formula:

$$\alpha = \frac{zp_1+p_2}{1+z}, \text{where } z = \sqrt{\frac{\ln\frac{2}{\beta}}{\ln\frac{1}{\delta}}}.$$

In fact, with $z > 0$ and $p_1 > p_2$, we have

$$p_2 = \frac{zp_2+p_2}{1+z} < \alpha = \frac{zp_1+p_2}{1+z} < p_1 = \frac{zp_1+p_1}{1+z}.$$

Therefore, α decided in the above approach satisfies the requirement $p_2 < \alpha < p_1$ of Lemma 1.

Replacing α in m_1 by $\frac{zp_1+p_2}{1+z}$, we can then decide m:

$$m = \lceil \frac{\ln\frac{1}{\delta}}{2(p_1-p_2)^2}(1+z)^2\rceil.$$

Since $z > 0$, it is easy to know that m monotonically increases with z and equivalently, m monotonically decreases with $\beta = v/n$. Intuitively, if v is too small, m will become too large so that we have to maintain many more hash tables and for processing each query C2LSH has to visit many more hash tables. On the other hand, if v is too big, the cost to maintain $\mathcal{P}_2$ will be too high since we should find and check at least βn candidates before we finally return a result. In this paper, we set $v = 100$ which seems to be a good trade-off and hence we have $\beta = 100/n$.

3.4 C2LSH for c-Approximate NN Search

As mentioned at the beginning of this section, C2LSH only materializes the m hash tables $\{T_1, \ldots, T_m\}$ corresponding to the m $(1, c, p_1, p_2)$-sensitive functions $\{h_1, \ldots, h_m\}$ in the initial base $\mathcal{B}_1$. Note that each T_i $(1 \leq i \leq m)$ is a sorted list of level-1 buckets. We can use those $\{T_1, \ldots, T_m\}$ to directly support $(1, c)$-NN search; and exploits virtual rehashing to use the same set of T_i's to support (R, c)-NN search at other radii $\{c, c^2, \ldots\}$. In this manner, we can do c-approximate NN Search without presuming a "magic" radius r_m for building hash tables. In the following, we first discuss the details of virtual rehashing, and then give the k-NN algorithms of C2LSH for c-approximate NN Search.

3.4.1 *Virtual Rehashing*

Given a query object q, there may not exist any data object within the ball centered at q with the radius $R = 1$. In this case, C2LSH does not return any data object as q's c-approximate NN. Then, C2LSH enlarges the search radius gradually, which simulates the search of E2LSH at $R = c, c^2, \ldots$.

Virtual rehashing first enlarges the search radius from $R = 1$ to $R = c$, and exploits m (c, c^2, p_1, p_2)-sensitive functions $H^c(\cdot) = \lfloor\frac{h(\cdot)}{c}\rfloor$ where the $h(\cdot)$'s are $(1, c, p_1, p_2)$-sensitive functions as defined in Equation (3) in the initial base $\mathcal{B}_1$. According to Observation 3 in Section 3.2, locating the level-c bucket $H^c(q) = \lfloor\frac{h(q)}{c}\rfloor$ is equivalent to locating c level-1 buckets in the m hash tables $\{T_1, \ldots, T_m\}$, whose level-1 *bid*s satisfy the following inequality with $r = c$:

$$\lfloor\frac{h(q)}{r}\rfloor * r \leq bid \leq \lfloor\frac{h(q)}{r}\rfloor * r + r - 1. \quad (4)$$

If necessary, we can similarly do virtual rehashing at subsequent radii $R = c^2, c^3, \ldots$, until we find query result for q.

For example, as shown in Figure 1, let approximation ratio $c = 3$, when $R = 1$, we assume the *bid* of the bucket $h(q)$ for some fixed $\vec{a}$ and b^* is 4. When R is enlarged to be 3, the level-3 bucket $H^3(q) = \lfloor\frac{h(q)}{3}\rfloor$ consists of 3 level-1 buckets whose *bid*s are respectively 3, 4, and 5, since these 3 *bid*s satisfy $\lfloor\frac{4}{3}\rfloor * 3 \leq bid \leq \lfloor\frac{4}{3}\rfloor * 3 + 3 - 1$, i.e., $3 \leq bid \leq 5$. And similarly when $R = 9$, the level-9 bucket $H^9(q)$ consists of 9 level-1 buckets whose *bid*s satisfy $0 \leq bid \leq 8$.

Avoiding Duplicate Collision Counting. A useful property of virtual rehashing for collision counting is that when we check a level-cr bucket t, we can skip the checking of the level-r bucket that is covered by t, since the bucket has been checked in previous round of searching at radius $R = r$. The formal proof will be given by Lemma 2 in Section 4.

3.4.2 *Nearest Neighbor Algorithm*

Given a k-NN query q, we traverse the m hash tables T_i with $1 \leq i \leq m$ starting from the level-1 bucket $H_i^1(q)$ with the radius $R = 1$. A candidate list C is used to store the frequent data objects encountered during the traversal, and is initialized to be an empty set. We first locate $H_i^1(q)$ in each hash table T_i. Then, we count the collision numbers for the data objects that appear in at least one of the m buckets $H_i^1(q)$'s, and add those frequent objects to C. At level-R (i.e., the current radius is R), if all the level-1 buckets that are covered by the m level-R buckets $H_i^R(q)$ have been traversed, but the number of frequent objects in C is still not big enough, we enlarge the radius R to cR and do collision counting over the level-cR buckets $H_i^{cR}(q)$. This process is repeated until enough candidates are found, and finally the top k NN objects in C are returned.

Intuitively, with each hash table T_i as a sorted list of level-1 buckets, we perform a round-robin scan over the m hash tables for one level-1 bucket at a time. In T_i, the scan starts from level-1 bucket $H_i^1(q)$, and always goes to the next "closest" level-1 bucket of $H_i^1(q)$. The next closest level-1 bucket of $H_i^1(q)$ can be chosen from the level-1 buckets covered by the level-R buckets $H_i^R(q)$'s. By doing collision counting over those "closest" buckets first, we expect that enough frequent objects can be found as soon as possible. This round-robin scan is similar to that used by the MedRank

Algorithm 1: k-NN

```
   Variable: $C$ - a candidate list; $P_l^i, P_r^i, P_s^i, P_e^i$ - pointers
             for traversing hash table $T_i$
1  $R := 1$, $C := \emptyset$;
2  while $TRUE$ do
3      if $|\{o | o \in C \wedge \|o, q\| \leq c \times R\}| \geq k$ then
4          return top $k$ NN objects in $C$;
5      end
6      for $1 \leq i \leq m$ do
7          if $R = 1$ then
8              $P_l^i, P_r^i, P_s^i, P_e^i \to H_i^1(q)$;  // Initialization
9              $next \to H_i^1(q)$;
10         else // $R > 1$
11             Alternately move $P_l^i$ one step "left" or move
               $P_r^i$ one step "right", providing $P_l^i \geq P_s^i$ or
               $P_r^i \leq P_e^i$;
12             Set $next$ to be updated $P_l^i$ or $P_r^i$;
13         end
14         Count collision for objects in the bucket pointed
           by $next$, and add frequent objects to $C$;
15         if $|C| \geq k + \beta n$ then
16             return top $k$ NN objects in $C$;
17         end
18     end
19     if we still have unchecked level-1 buckets then
20         go to Line 6;
21     end
22     for $1 \leq i \leq m$ do
23         $P_l^i := P_s^i$;   $P_r^i := P_e^i$;
24         $P_s^i \to$ the leftmost level-1 bucket of $H_i^{cR}(q)$;
25         $P_e^i \to$ the rightmost level-1 bucket of $H_i^{cR}(q)$;
26     end
27     $R := c \times R$;
28 end
```

algorithm [3]. The details of the k-NN algorithm is shown in Algorithm 1.

Terminating condition. The k-NN Algorithm terminates in two cases which are respectively supported by the properties $\mathcal{P}_1$ and $\mathcal{P}_2$ of Lemma 1:

C_1: At level-R, there exist at least k candidates whose distances to q are less than or equal to cR (referring to Line 3-5 in Algorithm 1).

C_2: When collision counting over all the level-R buckets is still ongoing, at least $k+\beta n$ candidates have been found (referring to Lines 15-17 in Algorithm 1).

Note that at level-R, we only check C_1 for termination at the very beginning. In this manner, we can avoid unnecessary collision counting. Intuitively, we can check C_1 for termination again at the end of level-R, specifically after collision counting over all the level-R buckets has been finished, and if C_1 is satisfied, then Algorithm 1 terminates and we have no need to enlarge R to cR. Logically, if C_1 is satisfied at the end of level-R, it is surely satisified at the beginning of level-cR. Therefore for simplicity, we only check C_1 for termination at the very beginning of each level.

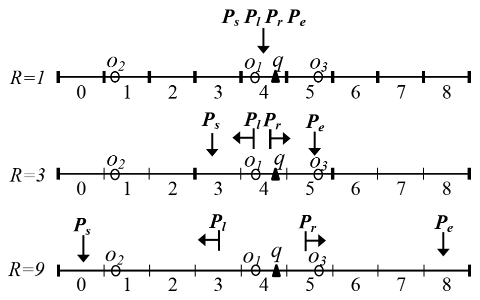


Figure 1: Query processing over a hash table at different radius R by Virtual Rehashing ($h(q) = 4$).

4. THEORETICAL ANALYSIS

In this section, we discuss the theoretical support of virtual rehashing, the bound on approximation ratio for 1-NN search, the query and space complexities, and the setting of collision threshold.

4.1 Theory of Virtual Rehashing

First, we list a simple observation of Floor function $\lfloor \cdot \rfloor$ [5].

Observation 4. *$\lfloor \frac{\lfloor x \rfloor}{v} \rfloor = \lfloor \frac{x}{v} \rfloor$, where x is a real number and v is a positive integer.*

Using this observation, we now give the proof of Observation 2 mentioned in Section 3.2.

Proof. From Observation 4, we have $H^R(o) = \lfloor \frac{h(o)}{R} \rfloor = \lfloor \frac{\lfloor \frac{\vec{a}\cdot\vec{o}+b^*}{w} \rfloor}{R} \rfloor = \lfloor \frac{\vec{a}\cdot\vec{o}+b^*}{Rw} \rfloor$. Since b^* is uniformly drawn from $[0, c^{\lceil \log_c td \rceil} w^2]$, $b^*/(c^{\lceil \log_c td \rceil} w)$ is uniformly distributed in $[0, w]$. Hence,

$$H^R(o) = \lfloor \frac{\vec{a}\cdot\vec{o} + \frac{b^*}{c^{\lceil \log_c td \rceil} w}(c^{\lceil \log_c td \rceil} w)}{Rw} \rfloor$$
$$= \lfloor \frac{\frac{\vec{a}\cdot\vec{o}}{R} + \frac{b c^{\lceil \log_c td \rceil} w}{R}}{w} \rfloor = \lfloor \frac{\frac{\vec{a}\cdot\vec{o}}{R} + bxw}{w} \rfloor,$$

where $b = \frac{b^*}{c^{\lceil \log_c td \rceil} w}$ is uniformly distributed in $[0, w]$, $x = \frac{c^{\lceil \log_c td \rceil}}{R} \geq 1$ and x is an integer.

Let $\vec{o'} = \frac{\vec{o}}{R}$, $H^R(o) = \lfloor \frac{\vec{a}\cdot\vec{o'}+bwx}{w} \rfloor = h'(o')$. By Observation 1, $h'(o')$ is $(1, c, p_1, p_2)$-sensitive. Since the distance between o_1 and o_2 is R times the distance between the corresponding o'_1 and o'_2, $H^R(o)$ is (R, cR, p_1, p_2)-sensitive. $\square$

We now give Lemma 2 which are made use of to avoid duplicate collision counting in the process of virtual rehashing.

Lemma 2. *For any query q, the level-R bucket identified by $H^R(q)$ is always contained by the level-cR bucket identified by $H^{cR}(q)$.*

Proof. According to Observation 3, for any query q, the level-R bucket identified by $H^R(q)$ consists of R consecutive level-1 buckets with the ID in the range of $[bid_R, bid_R + R - 1]$ where $bid_R = \lfloor \frac{h(q)}{R} \rfloor * R$. Similarly, the level-$cR$ bucket $H^{cR}(q)$ consists of cR consecutive level-1 buckets with the ID in the range of $[bid_{cR}, bid_{cR} + cR - 1]$ where $bid_{cR} = \lfloor \frac{h(q)}{cR} \rfloor * cR$. Hence, to establish Lemma 2, we need to prove that $bid_R \geq bid_{cR}$ and $bid_R + R - 1 \leq bid_{cR} + cR - 1$.

It is easy to know that $h(q) = \lfloor \frac{h(q)}{R} \rfloor * R + x = \lfloor \frac{h(q)}{cR} \rfloor * cR + y$, where x and y are respectively the integers in the range of $[0, R-1]$ and $[0, cR-1]$. By Observation 4,

$$\begin{aligned}
&\lfloor\frac{h(q)}{cR}\rfloor = \lfloor\frac{\lfloor\frac{h(q)}{R}\rfloor}{c}\rfloor \leq \frac{\lfloor\frac{h(q)}{R}\rfloor}{c}\\
\Longrightarrow & y - x = \left(\frac{\lfloor\frac{h(q)}{R}\rfloor}{c} - \lfloor\frac{h(q)}{cR}\rfloor\right) * cR \geq 0\\
\Longrightarrow & bid_R - bid_{cR} = h(q) - x - (h(q) - y) = y - x \geq 0.
\end{aligned}$$

On the other hand,

$$\begin{aligned}
&(bid_{cR} + cR - 1) - (bid_R + R - 1)\\
=&\lfloor\frac{h(q)}{cR}\rfloor * cR + cR - (\lfloor\frac{h(q)}{R}\rfloor * R + R)\\
=&[(\lfloor\frac{h(q)}{cR}\rfloor + 1) * c - (\lfloor\frac{h(q)}{R}\rfloor + 1)] * R\\
=&[(\lfloor\frac{\lfloor\frac{h(q)}{R}\rfloor}{c}\rfloor * c + c - 1 + 1) - (\lfloor\frac{h(q)}{R}\rfloor + 1)] * R\\
\geq&[(\lfloor\frac{h(q)}{R}\rfloor + 1) - (\lfloor\frac{h(q)}{R}\rfloor + 1)] * R = 0.
\end{aligned}$$

□

4.2 Bound on Approximation Ratio

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

THEOREM 1. *Algorithm 1 returns a c^2-approximate NN with at least constant probability.*

PROOF. Let o^* be the real NN of query q, and $r^* = \|o^*, q\|$. Let R be the smallest power of c bounding r^* such that $c^i < r^* \leq c^{i+1} = R$, where i is an integer. In other words, $B(q, \frac{R}{c})$ is empty but $o^* \in B(q, R)$. Then, we have $R < cr^*$ and $cR < c^2r^*$.

Assume the properties $\mathcal{P}_1$ and $\mathcal{P}_2$ of Lemma 1 hold at the same time, Algorithm 1 terminates in either the C_1 case or in the C_2 case as described in Section 3.4.

Suppose Algorithm 1 terminates in case C_1 at the beginning of level-R or even smaller level, it is guaranteed that Algorithm 1 returns a data object o_1 belonging to $B(q, cR)$. Hence, the distance of o_1 to q is at most c^2r^*, i.e., $\|o_1, q\| \leq cR < c^2r^*$. Otherwise, the situation that Algorithm 1 terminates in case C_1 must happen at the beginning of level-cR. This is because $\mathcal{P}_1$ of Lemma 1 guarantees there must exist an object $o \in B(q, R)$ among the candidates identified at level-R. When Algorithm 1 returns an object o_2, o_2 is as least as good as o. The distance of o_2 to q is then at most c^2r^*, i.e., $\|o_2, q\| \leq \|o, q\| \leq R < cr^* < c^2r^*$.

If Algorithm 1 terminates in case C_2, since $\mathcal{P}_2$ of Lemma 1 is true, there are no more than βn false positives. Thus, when Algorithm 1 returns the top 1 NN object o, we can assure that $o \in B(q, cR)$ and $\|o, q\| \leq cR < c^2r^*$.

Hence, in either case, the object returned by Algorithm 1 has a distance to q at most c^2r^*. From Lemma 1, the properties $\mathcal{P}_1$ and $\mathcal{P}_2$ hold at the same time with at least constant probability. Therefore, Theorem 1 is established. □

4.3 Query Time and Space Complexities

As mentioned in Section 3.3, we set $\beta = 100/n$ for C2LSH where n is the cardinality of the database D, and hence we have $m = O(\log n)$.

The query time of C2LSH consists of three parts. The first part is the time of locating the $m = O(\log n)$ buckets of the query object, and it is $md = O(d \log n)$ where d is the dimensionality of data objects. The second part is the time of collision counting. Obviously, we have to do collision counting for at most n objects over each hash table. Hence, the time of collision counting over m hash tables is $O(n \log n)$. The third part is the time of computing real distances for at most $k + \beta n$ candidates. So the query time of this part is $(k + \beta n)d = O(d)$. Therefore, the total query time is $O(d \log n + n \log n)$.

The space consumption of C2LSH consists of both the space consumption of dataset and the space of $m = O(\log n)$ hash tables. For the space of dataset, the space consumption is $O(dn)$. The space of m hash tables is $O(n \log n)$ because in each hash table, there are n data object ID's. Therefore, the total space consumption of C2LSH is $O(dn + n \log n)$.

4.4 Collision Threshold vs Candidate Criteria

For a data object o, if its distance to a query object q is larger than R but less than cR, i.e., $o \notin B(q, R)$ but $o \in B(q, cR)$, we call it a level-cR-only object. By $\mathcal{P}_1$ of Lemma 1, at level-cR, a level-cR-only object will be frequent and hence taken as a candidate for q with probability at least $1 - \delta$. If we can identify some level-cR-only objects as candidates at level-R, we may be able to speed up C2LSH. In fact, it is possible to identify both level-cR-only and level-c^2R-only objects as candidates at level-R by using a smaller candidate criteria C_t to replace the collision threshold $l = \alpha m$ in Lemma 1. The C_t is chosen so as to ensure that with a constant probability $\mathcal{P}_1$ of Lemma 1 still holds.

At level-r, the collision probability that a data object o collides with a query object q is $p(\frac{\|o,q\|}{r})$, where the collision probability function $p(\cdot)$ is as follows according to Section 2:

$$p(s) = 1 - 2norm(-w/s) - \frac{2}{\sqrt{2\pi}w/s}(1 - e^{-\frac{w^2}{2s^2}}).$$

For a level-c^2r-only object o_{c^2r}, where r is some integer power of c, let $s = \|o_{c^2r}, q\|$, then $cr < s \leq c^2r$. The collision probabilities of o_{c^2r} at level-c^2r, level-cr and level-r are given by $p(\frac{s}{c^2r})$, $p(\frac{s}{cr})$ and $p(\frac{s}{r})$ respectively. By $\mathcal{P}_1$ of Lemma 1, the collision number $\#collision(o_{c^2r})$ exceeds l at level-c^2r with probability at least $1-\delta$. Let A denote the event "o_{c^2r} collides with q at level-c^2r" and B denote the event "o_{c^2r} collides with q at level-S", where S is an integer power of c and $S \leq c^2r$. By Lemma 2, we know that $Pr[A|B] = 1$. The probability that o_{c^2r} collides with q at level-S given that o_{c^2r} collides with q at level-c^2r is given by

$$Pr[B|A] = \frac{Pr[A|B] \cdot Pr[B]}{Pr[A]} = \frac{Pr[B]}{Pr[A]}.$$

Hence, the expected collision number of o_{c^2r} at level-cr, denoted by $E_{cr}(o_{c^2r})$, is at least $\frac{p(\frac{s}{cr})}{p(\frac{s}{c^2r})}l$; and the expected collision number of o_{c^2r} at level-r, denoted by $E_r(o_{c^2r})$, is at least $\frac{p(\frac{s}{r})}{p(\frac{s}{c^2r})}l$. Furthermore, by running Matlab, it is easy to know that $\frac{p(\frac{s}{cr})}{p(\frac{s}{c^2r})} \geq \frac{p(c)}{p(1)}$ and $\frac{p(\frac{s}{r})}{p(\frac{s}{c^2r})} \geq \frac{p(c^2)}{p(1)}$, where $cr < s \leq c^2r$ for both $c = 2$ and $c = 3$. In other words, $E_{cr}(o_{c^2r}) \geq \frac{p(c)}{p(1)}l$ and $E_r(o_{c^2r}) \geq \frac{p(c^2)}{p(1)}l$.

From above analyses, we set $C_t = \frac{p(c^2)}{p(1)}l = \frac{p(c^2)}{p(1)}\alpha m$. Since for object o_{c^2r}, the collision probability at level-r is $p(\frac{s}{r}) \geq \frac{p(c^2)}{p(1)}p(\frac{s}{c^2r}) \geq \frac{p(c^2)}{p(1)}p(1) > \frac{p(c^2)}{p(1)}\alpha$, where $s = \|o_{c^2r}, q\|$, we can construct a new $\mathcal{P}_1$ to replace the old $\mathcal{P}_1$ of Lemma 1 as follows:

At level-r, for level-S object o_S, where S is a power of c and $S \leq c^2 r$ (specifically $S = c^2 r$ or $S = cr$), for given parameters: collision threshold percentage $\frac{p(c^2)}{p(1)}\alpha$, base cardinality m and false positive percentage β, the error probability δ_S satisfies:

$$Pr[\#collsion(o_S) \geq C_t] \geq 1 - exp(-2(p(\frac{S}{r}) - \frac{p(c^2)}{p(1)}\alpha)^2 m) = 1 - \delta_S,$$

where α, β and m are the same as defined in Lemma 1.

PROOF. The proof is the same as that of $\Pr[\mathcal{P}_1] \geq 1 - \delta$ in Lemma 1 in Appendix A. $\square$

Therefore, $\delta_S = exp(-2(p(\frac{S}{r}) - \frac{p(c^2)}{p(1)}\alpha)^2 m) \leq \delta_{c^2 r} = exp(-2(p(c^2) - \frac{p(c^2)}{p(1)}\alpha)^2 m) = \delta^{(\frac{p(c^2)}{p(1)})^2}$, which is a constant for given c, $p(c^2)$, $p(1)$ and δ. Thus, the new $\mathcal{P}_1$ holds with at least constant probability.

When Algorithm 1 runs with candidate criteria C_t, intuitively we can expect objects closer to q can be found earlier with higher probability. Thus, those objects should appear in the first $k + \beta n$ candidates with higher probability. In fact, at level-r, the expected collision numbers of a level-r-only object o_r, a level-cr-only object o_{cr} and a level-c^2r-only object o_{c^2r}, i.e., $E_r[o_r]$, $E_r[o_{cr}]$ and $E_r[o_{c^2r}]$, satisfy:

$$C_t = \frac{p(c^2)}{p(1)} l \leq E_r[o_{c^2r}] < \frac{p(c)}{p(1)} l \leq E_r[o_{cr}] < l \leq E_r[o_r].$$

5. EXPERIMENTS

In this section, we study the performance of C2LSH using both synthetic and real datasets. Comparisons with the state-of-the-art LSH based algorithms for external memory, i.e., LSB-tree and LSB-forest, are also conducted.

5.1 Datasets and Queries

In our experiments, we use four real data sets: *Mnist*[2], *Color*[3], *Audio*[4], and *LabelMe*[5]. A synthetic dataset called *RandInt* is also used. When the dimension values are real numbers, we normalize them to integers by proper scaling.

Mnist. The Mnist dataset contains 60,000 handwritten pictures. Each picture has 28 × 28 pixels with each pixel corresponding to an integer in the range of $[0, 255]$. Hence, every data object (i.e., a picture) has 784 dimensions. Since many pixels take zero-values, we follow Tao et al. [13] and take the top 50 dimensions with the largest variance to construct a dataset of 60,000 50-dimensional objects. In addition, there is a test set of 10,000 data objects, from which we randomly choose 50 data objects to form a query set, and apply the same dimensionality reduction for each query object.

Color. The Color dataset contains 68,040 32-dimensional data objects, which are the color histograms of images in the Corel collection [9]. The dimension values are real numbers with at most 6 decimal digits, and hence we scale them by multiplying 10^6. We randomly choose 50 data objects to form a query set and remove them from the dataset. As a result, in our experiments, the size of Color dataset is 67,990.

Audio. The Audio dataset contains 54,387 192-dimensional data objects. It is extracted from the LDC SWITCHBOARD-1 collection, which contains 2,400 two-sided telephone conversations among 543 speakers from all areas of the United States. We normalize dimension values to be integers in the range of $[0, 100, 000]$ and randomly pick 50 data objects from the dataset to form a query set. Therefore, the size of Audio is 54,337.

LabelMe. The LabelMe dataset contains 181,093 images with annotations provided by CSAIL Laboratory of MIT. We obtain the GIST feature of each image and generate a corresponding 512-dimensional data object. The dimension values are normalized to be integers in the range of $[0, 58, 104]$. We randomly pick 50 data objects as a query set. Hence, the size of LabelMe is 181,043.

RandInt. We use synthetic datasets to study the influences of dimensionality and dataset size. We first fix the dataset size to be $10K$, vary the dimensionality from 100 to 2,000, and generate a set of datasets called *RI_n10K*. We then fix the dimensionality to be 1,000, vary the size among $\{10K, 20K, 40K, 80K, 160K\}$, and generate another set of datasets called *RI_d1000*. The dimension values are integers randomly and uniformly chosen from $[0, 10, 000]$. For each synthetic dataset, we also randomly generate a query set with 50 data objects.

5.2 Evaluation Metrics

We adopt three metrics to evaluate a c-approximate NN search method in the experiments.

Query Efficiency. Since c-approximate NN search is I/O intensive, we evaluate the query efficiency in terms of I/O cost. The I/O cost consists of two parts: the cost for finding candidates and the cost for distance computation in the original space R^d.

Query Accuracy. We adopt the same metric used in [13] to measure the quality of query results. Specifically, for a query object q, denote the k-NN query results returned by a method as $o_1, \ldots, o_k$, which are sorted in nondecreasing order of their distances to q. Let $o_1^*, \ldots, o_k^*$ be the actual k-NN objects of q and they are also sorted in the same way. Then, the *rank-i approximation ratio* is defined as

$$R_i(q) = \frac{\|o_i, q\|}{\|o_i^*, q\|},$$

where $i = 1, \ldots, k$ and $\| \cdot \|$ denotes the Euclidean Distance. The overall ratio is defined as $\frac{1}{k}\sum_{i=1}^{k} R_i(q)$. The more closely the overall ratio approaches 1, the more accurate the reults are, and when it equals to 1, the results are exact. Given a query set Q, we use the mean of the overall ratios of all the query objects in Q as the final measurement for query accuracy. This mean is called the *average overall ratio*. For simplicity, we may just call it *ratio*.

Space Consumption. The space consumption is the size of index file.

5.3 Parameter Settings of C2LSH

In this section, we discuss the performance of C2LSH when different criteria l or C_t are adopted for determining candidates. The approximation ratio c is set to be 2 or 3. We manually set the interval size $w = 1$, false positive percent-

[2]`http://yann.lecun.com/exdb/mnist/`
[3]`http://kdd.ics.uci.edu/databases/CorelFeatures/`
[4]`http://www.cs.princeton.edu/cass/audio.tar.gz`
[5]`http://labelme.csail.mit.edu/instructions.html`

Table 1: Performance of C2LSH for 1-NN query

C2LSH			Color	Mnist	Audio	LabelMe
$c=2$	m		390	386	383	419
	l	I/O	2245	2813	3479	5068
		Ratio	1.00	1.01	1.00	1.03
	C_t	I/O	697	937	1080	1440
		Ratio	1.00	1.00	1.02	1.07
$c=3$	m		208	206	205	224
	l	I/O	1069	1428	1541	2337
		Ratio	1.02	1.01	1.01	1.02
	C_t	I/O	185	187	187	232
		Ratio	1.06	1.13	1.13	1.19

age $\beta = 100/n$, error probability $\delta = 0.01$. The remaining parameters including p_1, p_2, collision threshold percentage α, and base cardinality m are calculated based on the analyses in section 3.3. Then, l or C_t can be computed based on the above settings.

5.3.1 *Approximation Ratio* c

Table 1 shows the performances of C2LSH on four real datasets for 1-NN query. We can see that the base cardinality m of case $c = 2$ is larger than that of case $c = 3$. The average overall ratio of case $c = 2$ is very close to 1, which means the 1-NN results returned in case $c = 2$ are very close to the real NNs. The ratio of case $c = 3$ is a bit larger than that of case $c = 2$, yet it is still very good. Notably, the I/O cost of case $c = 3$ is only about one half and $\frac{1}{6}$ of those of case $c = 2$ using l and C_t respectively. Therefore, it is good to use $c = 3$ to trade a little accuracy for a much higher query efficiency.

5.3.2 *Candidate Criteria* C_t

From Table 1, the l version of C2LSH is more accurate than the C_t version. The l version theoretically guarantees that the returned object is at most c^2-approximate of the real NN with at least constant probability, while the C_t version does not have such a guarantee. However, to achieve the theoretical guarantee, the l version pays higher I/O cost. As discussed in Section 4.4, the C_t version can look ahead for at most two levels. For instance, for an object $o \in B(q, R)$, it will be returned at level-R in the l version but the C_t version can return it at level-$\frac{R}{c^2}$ by chance. Since a level-R bucket logically consists of c^2 level-$\frac{R}{c^2}$ buckets, the I/O cost for counting a level-R bucket can be almost c^2 times the cost for counting a level-$\frac{R}{c^2}$ one. Such difference in I/O cost can be observed from Table 1. Actually, although the C_t version can not provide a theoretical guarantee, its accuracy is practically high and its efficiency is very satisfactory. Therefore, in the following experiments, we set $c = 3$ and adopt the C_t criterion for C2LSH.

5.4 Comparisons on Synthetic datasets

We use *RL_n10K* and *RL_d1000* datasets to study the influences of dimensionality and dataset size. Figure 2 shows the I/O cost and average overall ratio for 50-NN queries, and Table 2 shows the space consumption.

We need a pre-specified page size B for constructing an LSB-tree, and, with different dimensionality, the page size required is different. Hence in the experiments on *RL_n10K* as shown in Figures 2(a) and 2(b), B is set to be 4KB when d varies from 100 to 300, 8KB when d varies from 400 to 600, 16KB when d varies from 700 to 1000, and 32KB when d is 2000. Our C2LSH method takes the same B setting in each corresponding experiment. From Figure 2(a), for a fixed B with the dimensionality d varying in a certain range, as d increases, the I/O cost of LSB-forest increases notably, and that of LSB-tree shows a similar trend. In contrast, the I/O cost of C2LSH is stable with d. The I/O cost of LSB-tree is lower than that of C2LSH, which is further lower than that of LSB-forest. From Figure 2(b), the average overall ratio of C2LSH is the best, and the ratio of LSB-forest is slightly better than that of LSB-tree.

Table 2: Space consumption on RandInt

(a) Space consumption vs. dimensionality d ($n = 10$K)

d	100	400	800	1000	2000
m	176	176	176	176	176
C2LSH	20MB	21MB	21MB	22MB	24MB
LSB-tree	4MB	38MB	37MB	66MB	132MB
L	32	45	45	50	50
LSB-forest	130MB	1.66GB	1.64GB	3.22GB	6.43GB

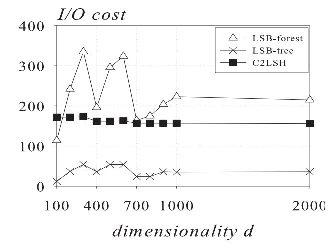
(b) Space consumption vs. dataset size n ($d = 1000$)

n	10K	20K	40K	80K	160K
m	176	188	200	211	222
C2LSH	22MB	44MB	90MB	181MB	353MB
LSB-tree	66MB	133MB	267MB	535MB	1.0GB
L	50	70	99	140	198
LSB-forest	3.2GB	9.1GB	25.8GB	73.2GB	206.8GB

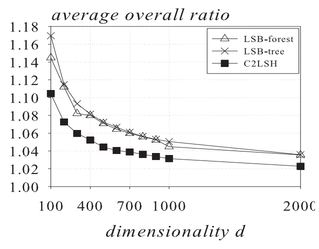
Figures 2(c) and 2(d) illustrate the trends of I/O cost and average overall ratio of different methods on the *RL_d1000* datasets. As the dataset size n increases, the I/O cost of LSB-forest increases very fast. The reason is that, the number of trees L in LSB-forest is $\sqrt{dn/B}$ which increases with n, and accessing more trees leads to bigger I/O cost. On the other hand, for a fixed n, L also gets larger as d increases. That's why the I/O cost of LSB-forest increases as d increases in Figure 2(a). In contrast, although the cardinality base m of C2LSH also increases with n, the I/O cost remains stable and lower than that of LSB-forest. The I/O cost of LSB-tree is still the lowest. Figure 2(d) shows that the accuracy of C2LSH is the best among the three methods.

Table 2(a) shows the space consumption with respect to d. As d increases from 100 to 2000, the space consumed by C2LSH remains small and only shows a slight increase, i.e., from 20MB to 24MB. In contrast, the spaces consumed by LSB-tree and LSB-forest respectively increase by about 30 and 50 times. When d equals to 2000, the space consumption of C2LSH is two magnitudes smaller than that of LSB-forest, i.e., 24MB vs. 6.43GB, and is one fifth of the space consumed by LSB-tree. The space consumption of C2LSH is more influenced by the dataset size n, which however is still significantly smaller than those of LSB-tree and LSB-forest, as shown in Table 2(b). On the largest dataset with n equal to 160K, the space consumed by C2LSH is 353MB while the spaces consumed by LSB-tree and LSB-forest are respectively 1.0GB and 206.8GB. In fact, the space consumption of C2LSH (for index file) is $O(n \log n)$, while that of LSB-tree is $O(\frac{dn}{B})$ and that of LSB-forest is $O((\frac{dn}{B})^{\frac{3}{2}})$. When d is large, the space consumption of LSB-tree and LSB-forest increases significantly while that of C2LSH is not affected by d.

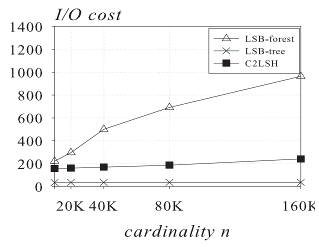
From this set of experiments, we can see C2LSH outperforms LSB-forest in terms of all the three metrics, i.e., query efficiency, query accuracy and space consumption, especially on the datasets with high dimensions. LSB-tree always has the lowest I/O cost. However, its accuracy is lower than that of C2LSH and its space consumption is larger than that of C2LSH on high dimensional datasets.



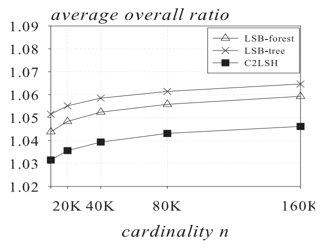
(a) I/O Cost vs. d



(b) Ratio vs. d

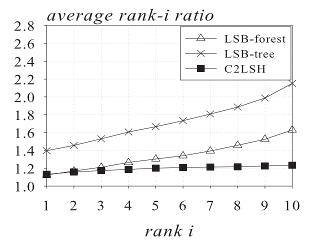


(c) I/O Cost vs. n

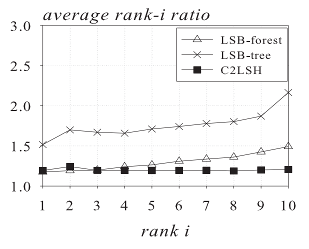


(d) Ratio vs. n

Figure 2: Performance on RandInt



(a) Audio



(b) LabelMe

Figure 4: Rank-i Ratio on Audio and LabelMe

5.5 Comparisons on Real Datasets

In this set of experiments, we use four real datasets to compare the performance of C2LSH, LSB-tree and LSB-forest. On each dataset, we conduct a series of k-NN searches by varying k in $\{1, 10, 20, 30, \ldots, 100\}$.

5.5.1 Comparison on High Dimensional Datasets

We first study the performance on two high-dimensional datasets, i.e., Audio with 192 dimensions and LabelMe with 512 dimensions.

Figures 3(a) and 3(c) respectively show the I/O cost on Audio and LabelMe, where the page sizes are respectively set to be 4KB for Audio and 8KB for LabelMe. On both datasets, the I/O cost of LSB-tree is the lowest, and the I/O cost of C2LSH is less than that of LSB-forest. As shown in Figures 3(b) and 3(d), the overall ratios of both C2LSH and LSB-forest are much smaller than that of LSB-tree. On LabelMe, C2LSH is more accurate than LSB-forest. While on Audio, the ratio of C2LSH is smaller than that of LSB-forest when $k \leq 70$. Note that the ratio of C2LSH increases with k while that of LSB-forest does not. The reason is that, the C_t version of C2LSH usually stops at terminating condition C_2 when $k+\beta n$ candidates have been found. Since $\beta = \frac{100}{n}$, C2LSH will return top k objects out of $k+\beta n = k+100$ candidates. For instance, for 1-NN search, C2LSH picks a top 1 result out of 1+100 candidates, while for 100-NN search, it returns top 100 results out of 100+100 candidates. Because relatively more number of candidates have been checked for picking one result when k is smaller, the results of C2LSH are more accurate.

We also show the Rank-i Approximation Ratio of 10-NN search on Audio and LabelMe in Figures 4(a) and 4(b). We can see that the quality of the objects returned by C2LSH is well maintained at every rank. In contrast, the quality of the rank-i objects returned by both LSB-forest and LSB-tree decrease apparently as i increases.

5.5.2 Comparison on Low Dimensional Datasets

In this section, we show the experiment results on two low-dimensional datasets, i.e., Color with 32 dimensions and Minst with 50 dimensions.

According to Figures 5(a) and 5(c), both LSB-tree and LSB-forest outperform C2LSH in terms of I/O cost. The reason is that, on low-dimensional datasets, the number of LSB-trees needed for constructing an LSB-forest is small. Moreover, the Z-order value for each data object is also short so that a leaf page can store more Z-order values. However, the number of hash tables m of C2LSH does not depend on dimensionality d, but depends on n. For example, m equals to 205 for the 192-dimensional Audio dataset, and equals to 208 for the 32-dimensional Color dataset, because the size of Audio is 54,337 and the size of Color is 67,990. Because the size of Color is large, the I/O cost of C2LSH increases. On the other hand, C2LSH outperforms both LSB-tree and LSB-forest in terms of query accuracy, as shown in Figure 5(b) and 5(d).

5.5.3 Space consumption

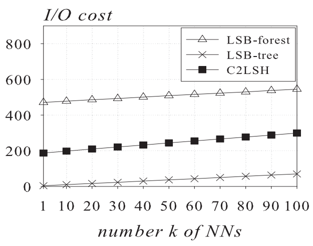
The space consumption of C2LSH, LSB-tree and LSB-forest on the four real datasets is listed in Table 3. C2LSH needs less space compared to LSB-forest, and its space consumption is even about two or three magnitude orders smaller than that of LSB-forest on the high dimensional datasets like Audio and LabelMe. LSB-tree consumes less space than C2LSH on low dimensional datasets, but much more space is needed by LSB-tree as the dimensionality increases.

Table 3: Space consumption on datasets

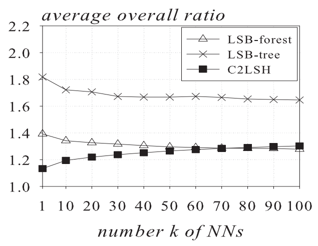
Dataset	Color	Mnist	Audio	LabelMe
m	208	206	205	224
C2LSH	159MB	53.7MB	122.5MB	373.6MB
LSB-tree	13.5MB	13.0MB	106MB	1.06GB
L	47	55	101	213
LSB-forest	633MB	714MB	10.5GB	224.8GB

5.6 Summary

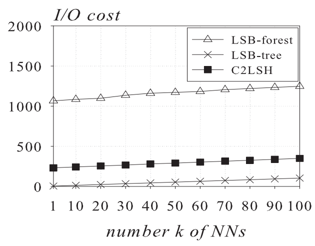
In summary, the performance of LSB-tree and LSB-forest is affected by the dimensionality d of datasets. They perform very well on the low dimensional datasets Color and Mnist. Although LSB-tree always performs well in terms of efficiency, the query quality is not so satisfying. On low dimensional datasets, C2LSH generally has a better average overall ratio than both LSB-tree and LSB-forest. Notably, the performance of C2LSH is not affected by d. On high dimensional datasets, C2LSH outperforms LSB-forest in terms of all three metrics. Specifically, the space consumption of C2LSH is two or three magnitude orders lower than that of LSB-forest on high dimensional datasets. Therefore, C2LSH



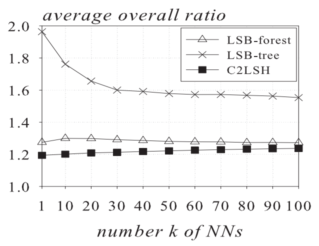
(a) Cost on Audio



(b) Ratio on Audio

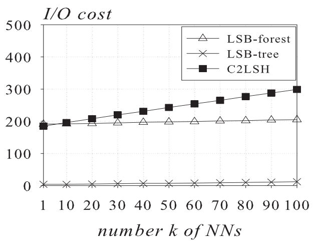


(c) Cost on LabelMe

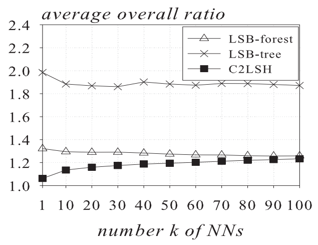


(d) Ratio on LabelMe

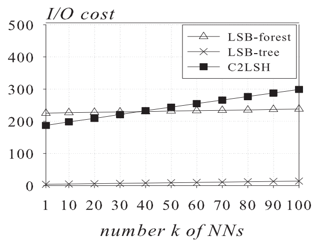
Figure 3: Performance on Audio and LabelMe



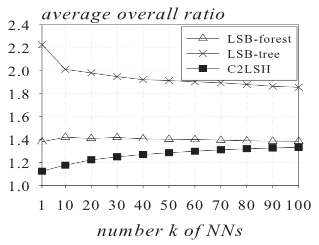
(a) Cost on Color



(b) Ratio on Color



(c) Cost on Mnist



(d) Ratio on Mnist

Figure 5: Performance on Color and Mnist

has a better overall performance than LSB-tree and LSB-forest, especially on high dimensional datasets. However, it should be noted that the LSB-forest has a theoretical guarantee that is different from our structure's guarantee. In particular, the LSB-forest ensures worst-case I/O cost sublinear to both n and d. Most of the overhead in LSB-forest is incurred due to this guarantee.

6. RELATED WORK

There is a large body of literature on the topic of NN search [1, 6, 9, 10, 13, 14, 15]. A good survey on techniques before year 2006 can be found in [12]. Recent work includes HashFile [15] and ATLAS [14]. The HashFile method is mainly designed for exact NN search in L_1 norm. The ATLAS method is a probabilistic algorithm for high dimensional similarity search over binary vectors with low similarity thresholds.

The LSH method is originally proposed by Indyk et al. for internal memory dataset in the Hamming space [8]. Later it is adapted for external memory use by Gionis et al. [4], and they propose to use a "magic radius" to reduce space consumption. The locality-sensitive hash functions based on p-stable distribution are proposed by Datar et al. [2]. The multi-probe LSH method, proposed by Lv et al. [11], not only checks the data objects that fall in the same bucket as the query object, but also checks the data objects falling in the "nearby" buckets. However, the multi-probe method still suffers from the need of building hash tables at different radiuses in order to achieve a theoretical gurantee of query quality.

The spirit of virtual rehashing is first proposed in the work on the LSB-tree/LSB-forest method[13]. Since the LSB-tree/LSB-forest method exploits compound hash functions, their virtual rehashing can be viewed as imposing multi-dimensional level-cR buckets over multi-dimensional level-R buckets. In contrast, we consider 1-dimensional buckets. Our k-NN Algorithm scans sorted lists of level-1 buckets in a round-robin way, which is similar to the MedRank method[3].

7. CONCLUSION

In this paper, we present the C2LSH method for c-approximate NN search. Our theoretical studies show that C2LSH can have a guarantee on query quality. Importantly, the performance of C2LSH is not affected by the dimensionality of data objects. The experimental studies based on synthetic datasets and four real datasets have shown that C2LSH outperforms the state of the art method LSB-forest in terms of all the three performance metrics: query efficiency, query accuracy and space consumption on high dimensional datasets. In addition, C2LSH can be straightforwardly implemented in relational database systems.

8. ACKNOWLEDGEMENTS

This work is partially supported by China NSF Grant 60970043 and Hong Kong RGC GRF project grant 617610. We would like to thank SIGMOD reviewers and especially the shepherd for giving us insightful comments. We thank Xingjia Ma, Qiang Huang, and Huaping Zhong for helping with the experiments.

9. REFERENCES

[1] J. L. Bentley. K-D trees for semi-dynamic point sets. In *Symposium on Computational Geometry*, 1990.

[2] M. Datar, P. Indyk, N. Immorlica, 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 *SIGMOD*, pages 301–312, 2003.

[4] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[5] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* 1994.

[6] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

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

[8] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[9] H. V. 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*, pages 364–397, 2005.

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

[11] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[12] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* 2006.

[13] 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, 2010.

[14] J. Zhai, Y. Lou, and J. Gehrke. Atlas: A probabilistic algorithm for high dimensional similarity search. In *SIGMOD*, pages 997–1008, 2011.

[15] D. Zhang, D. Agrawal, G. Chen, and A. K. H. Tung. Hashfile: An efficient index structure for multimedia data. In *ICDE*, pages 1103–1114, 2011.

APPENDIX

A. PROOF OF LEMMA 1

PROOF: First, we prove that $Pr[\mathcal{P}_1] \geq 1-\delta$.

For $\forall o \in B(q,R)$,

$$Pr[\#collision(o) \geq \alpha m] = \sum_{i=\alpha m}^{m} C_m^i p^i (1-p)^{m-i},$$

where $p = Pr[h_j(o) = h_j(q)] \geq p_1$, $j = 1, 2, \ldots, m$.

We define m Bernoulli random variables $X_i \sim B(m, 1-p)$ with $1 \leq i \leq m$. We let X_i equal 1 if o does not collide with q, i.e., $Pr[X_i = 1] = 1-p$. Let X_i equal 0 if o collides with q, i.e., $Pr[X_i = 0] = p$.

Since $E(X_i) = 1-p$, by linearity of expectation, we have $E(\overline{X}) = 1-p$ where $\overline{X} = \frac{\sum_{i=1}^m X_i}{m}$. From Hoeffding's Inequality [7], for any $t = p - \alpha > 0$,

$$Pr[\overline{X} - E(\overline{X}) \geq t] = Pr[\frac{1}{m}\sum_{i=1}^{m} X_i - (1-p) \geq t]$$
$$= Pr[\sum_{i=1}^{m} X_i \geq (1-\alpha)m] \leq \exp(-\frac{2(p-\alpha)^2 m^2}{\sum_{i=1}^{m}(1-0)^2})$$
$$= \exp(-2(p-\alpha)^2 m) \leq \exp(-2(p_1-\alpha)^2 m).$$

Since the event "$\#collision(o) \geq \alpha m$" is equivalent to the event "o misses the collision with q less than $(1-\alpha)m$ times",

$$Pr[\#collision(o) \geq \alpha m] = Pr[\sum_{i=1}^{m} X_i < (1-\alpha)m]$$
$$= 1 - Pr[\sum_{i=1}^{m} X_i \geq (1-\alpha)m] \geq 1 - \exp(-2(p_1-\alpha)^2 m).$$

Therefore, when $m = \lceil \max(\frac{1}{2(p_1-\alpha)^2}\ln\frac{1}{\delta}, \frac{1}{2(\alpha-p_2)^2}\ln\frac{2}{\beta}) \rceil$,

$$Pr[\mathcal{P}_1] = Pr[\#collision(o) \geq \alpha m] \geq 1-\delta > \frac{1}{2}.$$

Second, we show that $Pr[\mathcal{P}_2] > \frac{1}{2}$.

For any data object $o \notin B(q, cR)$,

$$Pr[\#collision(o) \geq \alpha m] = \sum_{i=\alpha m}^{k} C_m^i p^i (1-p)^{m-i},$$

where $p = Pr[h_j(o) = h_j(q)] \leq p_2 < \alpha$, $j = 1, \ldots, m$.

Let $f(p) = p^i(1-p)^{m-i}$ with $\alpha m \leq i \leq m$. We take the derivative of $f(p)$, which is $f'(p) = p^{i-1}(1-p)^{m-i-1}(i-pm)$. When $0 < p < \alpha < 1$, we have $f'(p) > 0$, which means that $f(p)$ is a *monotonic function* and it monotonically increases with p. Therefore, when $p \leq p_2 < \alpha$,

$$Pr[\#collision(o) \geq \alpha m] \leq \sum_{i=\alpha m}^{m} C_m^i p_2^i (1-p_2)^{m-i}.$$

Similarly, we define m Bernoulli random variables $Y_i \sim B(m, 1-p_2)$ with $1 \leq i \leq m$, where $Pr[Y_i = 1] = 1-p_2$ and $Pr[Y_i = 0] = p_2$. Since $E(Y_i) = 1-p_2$, $E(\overline{Y}) = 1-p_2$ where $\overline{Y} = \frac{\sum_{i=1}^m Y_i}{m}$. Based on Hoeffding's Inequality, for any $t = \alpha - p_2 > 0$, we have

$$Pr[\#collision(o) \geq \alpha m] \leq \sum_{i=\alpha m}^{m} C_m^i p_2^i (1-p_2)^{m-i}$$
$$= Pr[\sum_{i=1}^{m} Y_i < (1-\alpha)m] = Pr[\frac{1}{m}\sum_{i=1}^{m} Y_i < (1-p_2-t)]$$
$$= Pr[(1-p_2) - \frac{1}{m}\sum_{i=1}^{m} Y_i > t)] = Pr[E(\overline{Y}) - \overline{Y} > t]$$
$$(\text{There exists } \Delta t > 0 \text{ such that } E(\overline{Y}) - \overline{Y} \geq t + \Delta t.)$$
$$\leq \exp(-2(\alpha - p_2 + \Delta t)^2 m)$$
$$< \exp(-2(\alpha - p_2)^2 m).$$

We define $S = \{o | \#collision(o) \geq \alpha m \wedge o \notin B(q, cR)\}$ and have $|S| \leq n$. Hence, the expectation of the size of S satisfies $E(|S|) < n * \exp(-2(\alpha - p_2)^2 m)$.

From Markov's Inequality, we have

$$Pr[|S| \geq \beta n] \leq \frac{E[|S|]}{\beta n} < \frac{1}{\beta} * \exp(-2(\alpha - p_2)^2 m).$$

Therefore, when $m = \lceil \max(\frac{1}{2(p_1-\alpha)^2}\ln\frac{1}{\delta}, \frac{1}{2(\alpha-p_2)^2}\ln\frac{2}{\beta}) \rceil$,

$$Pr[|S| < \beta n] = 1 - Pr[|S| \geq \beta n]$$
$$> 1 - \frac{1}{\beta} * \exp(-2(\alpha - p_2)^2 m) > \frac{1}{2}.$$

Thus, Lemma 1 is established. □