Received 21 May 2022, accepted 8 June 2022, date of publication 13 June 2022, date of current version 7 July 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3182802

LayerLSH: Rebuilding Locality-Sensitive Hashing Indices by Exploring Density of Hash Values

JIWEN DING, ZHUOJIN LIU, YANFENG ZHANG, (Member, IEEE), SHUFENG GONG, AND GE YU, (Senior Member, IEEE)
School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China

Corresponding author: Yanfeng Zhang (zhangyf@mail.neu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 62072082, in part by the Key Research and Development Program of Liaoning Province under Grant 2020JH2/10100037, and in part by the Fundamental Research Funds for the Central Universities under Grant N2216015.

ABSTRACT Locality-sensitive hashing (LSH) has attracted extensive research efforts for approximate nearest neighbors (NN) search. However, most of these LSH-based index structures fail to take data distribution into account. They perform well in a uniform data distribution setting but exhibit unstable performance when the data are skewed. As known, most real life data are skewed, which makes LSH suffer. In this paper, we observe that the skewness of hash values resulted from skewed data is a potential reason for performance degradation. To address this problem, we propose to rebuild LSH indices by exploring the density of hash values. The hash values in dense/sparse ranges are carefully reorganized using a multi-layered structure, so that more efforts are put into indexing the dense hash values. We further discuss the benefit in distributed computing. Extensive experiments are conducted to show the effectiveness and efficiency of the reconstructed LSH indices.

INDEX TERMS LSH, nearest neighbors search, multi-layered structure, data skewness.

I. INTRODUCTION

The nearest neighbors (NN) search problem aims to find objects that are close to the given query, which is the basic and important problem in a wide range of applications [1]–[5]. Due to the difficulty in finding an efficient method for exact NN search in high-dimensional space, many researchers have focused on approximate nearest neighbors search as an alternative approach. Locality sensitive hashing (LSH) [6] is known as one of the most promising indexing methods for approximate nearest neighbors search, where the locality sensitive hash function has the property that points that are closer to each other have a higher probability of colliding than points that are farther apart. To improve the hashing effect, m randomly chosen LSH functions are utilized together to generate a compound hash key for each object o. In compensation for the loss of candidate points because of importing compound hash key, a large number l of hash tables are constructed.

Many LSH variants have been proposed in recent years. E2LSH [7] is a typical table-based LSH which proposes a hash function used in Euclidean space. The data points with the same hash value or the same concatenating hash values are placed in the same bucket, which implies that they are close to each other. The approximate NN search is achieved by returning the data in the same bucket as which the query falls in.

However, most of these LSH index structures fail to take data distribution into account. They perform well in a uniform data distribution setting, but exhibit instable performance when the data are skewed. As known, most real life data are skewed, which makes LSH suffer from poor search quality. Based on our observation, the skewed data distribution leads to skewed hash values and, as a result, leads to a skewed index structure. This is the potential reason for the performance degradation.

The Euclidean-based LSH function [7] projects high-dimensional data points to a real number line and partitions the line into fixed-length intervals. As a result, the hash values exhibit skewed distribution as long as the original data are skewed. The points with the same hash values are assigned to the same bucket, so the bucket sizes vary greatly. Figure 1a shows the skewed bucket size distribution for the KDD dataset (Table 1). Intuitively, LSH-based kNN search

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Asif Naeem.

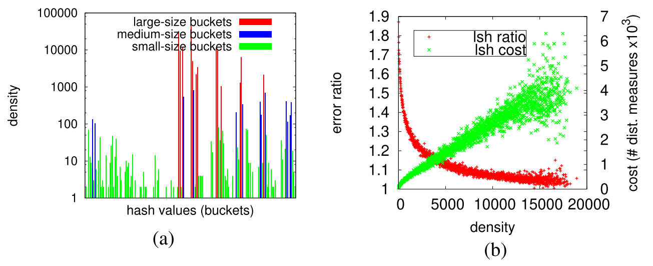


FIGURE 1. **(a) The density of hash values for KDD dataset (in log scale). Each bar represents the number of points in a bucket. (b) Density of hash values vs. query accuracy and query cost (KDD). Smaller error ratio indicates higher accuracy, which is defined in Equation (12).**

displays higher accuracy for dense queries[1] while lower accuracy for sparse queries. This is because the distances from query q to its kNNs are small (or large) in a dense (or sparse) region, so that they are likely (or unlikely) to be hashed to the same bucket. On the other hand, due to the large bucket size, it displays higher cost (evaluated by the number of distance measurements) for dense queries than sparse queries. This phenomenon is illustrated in Figure 1b. If we randomly return k points in the bucket rather than returning the exactly k nearest points, it may lead to much lower accuracy. Specifically, for the kNN applications with a small k (e.g., top 1 query at extreme), if we randomly return 1 point in the dense bucket, the error problem will be enlarged and will significantly impact the user experience.

In this paper, we propose to rebuild LSH index structures by exploring the density of hash values. The hash values in dense ranges are rehashed to make them distributed more evenly, so as to reduce the query cost. The hash values in sparse ranges are merged to be returned together during query processing, so as to improve the search quality. Therefore, the rebuilt LSH indices become more targeted in terms of data distribution, and a multi-layered structure is constructed. Comparing with the simple rehashing method, the multi-layered approach will still guarantee the search quality by carefully choosing the number of groups and hash functions, which is a nice property for applications with restrict accuracy requirement.

A. DIFFERENCE TO DATA SENSITIVE HASHING

The recently proposed data sensitive hashing, e.g., DSH [8] and selective hashing [9], also leverages data distributions. DSH [8] chooses the most suitable hashing family by learning data distributions. Selective hashing [9] creates multiple LSH indices with different granularities (i.e., radii) and locates each point only in one suitable LSH index according to data densities. These data sensitive hashing techniques learn the appropriate hash families from the data, and accordingly have the ability to create relatively balanced indexing structures. Our approach is orthogonal to them since we rely on the density of hash values and directly rebuild the existing index structures. Moreover, our rebuilding scheme can also be used as a postprocessing step for these data sensitive hashing techniques to further improve performance. We will rebuild DSH index to illustrate the possibility.

[1]A query falling in a dense bucket/range is referred to as a dense query, and vice versa.

B. CONTRIBUTIONS

We list our contributions as follows.

- We rebuild the basic LSH structure and design **LayerLSH** (Section III). The points in overloaded bucket are recursively rehashed to multiple groups of smaller buckets, forming a multi-layered index structure. Thus, the query is more efficient since a less number of more accurate NN candidates are returned. Further, by carefully choosing the new set of rehashing LSH parameters, the collision probability can be guaranteed. We also propose a stream processing approach to adapt streaming data.
- We demonstrate the benefits of our approach in distributed computing (Section IV). We also present a use case on supporting distributed all-pairs computation, i.e., point density evaluation.
- We conduct extensive experiments on real datasets to verify the effectiveness and efficiency of the proposed multi-layered structures. LayerLSH can reach the same search quality as LSH with only 5%-20% query cost. LayerLSH also exhibits much better performance on distributed point density approximation (Section V).

We survey the related work in Section VI. Finally, we conclude our work in Section VII.

II. PRELIMINARIES

A. PROBLEM SETTING

The problem of nearest neighbors search refers to finding objects that are similar to the query object. The typical kNN search problem is formally defined as follows.

Definition 1 (kNN): Given an object q, a dataset O and an integer k ($k < |O|$), the kNN query returns a set of k objects from O denoted as KNN(q, O), such that $\forall o \in$ KNN(q, O), $\forall o' \in O -$ KNN(q, O), $|q, o| \leq |q, o'|$, where $|\cdot, \cdot|$ denotes the distance between two objects.

In this paper, we focus on answering **approximate** kNN queries for high-dimensional data in the Euclidean space. That is, we aim to find k objects whose distances are within a small factor $(1+\epsilon)$ of the exact k-nearest neighbors' distances and minimize ϵ. Our goal is to design an indexing scheme for approximate kNN queries with both high search quality and high efficiency.

B. LOCALITY-SENSITIVE HASHING

The Locality-Sensitive Hashing (LSH) function has the property that points that are closer to each other have a higher probability of colliding than points that are farther apart [6]. Let O be the dataset of n data objects in d-dimensional Euclidean space $\mathbb{R}^d$ and let $||o_1, o_2||$ denote the Euclidean distance between two objects o_1 and o_2, $o_1, o_2 \in O$. LSH is formally defined as follows.

Definition 2 (Locality Sensitive Hashing): Given a distance r, an approximation ratio c and two probability values $P1$ and $P2$, a hash function $h : \mathbb{R}^d \rightarrow U$ is called $(r, cr, P1, P2)$-sensitive if for any $o_1, o_2 \in O$

- If $||o_1, o_2|| \leq r$ then $\Pr[h(o_1) = h(o_2)] \geq P1$,
- If $||o_1, o_2|| > cr$ then $\Pr[h(o_1) = h(o_2)] \leq P2$,

We pick $c > 1$ and $P1 \geq P2$. With these choices, nearby objects (i.e. those within distance r) have a greater chance of being hashed to the same value than points that are far apart, i.e. those at a distance greater than cr away.

The commonly used LSH family for Euclidean distance consists of LSH functions in the following form [7]:

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

where a is a d-dimensional random vector, each entry of which is chosen independently from standard Gaussian distribution $\mathcal{N}(0, 1)$ [10], b is a real number chosen from $[0, w]$, and w is also a real number representing the partition width of the LSH function.

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 randomly chosen hash function h, denoted as $p(s, w)$, can be computed as follows [7].

$$\begin{aligned} p(s, w) &= Pr[h(o_1) = h(o_2)] \\ &= \int_0^w \frac{1}{s} f_2(\frac{t}{s})(1 - \frac{t}{w})dt \\ &= 1 - 2norm(-\frac{w}{s}) - \frac{2s}{\sqrt{2\pi} w}(1 - e^{-\frac{w^2}{2s^2}}), \end{aligned} \tag{2}$$

where $f_2(x)$ is the density function of a Gaussian distribution [7], i.e., $f_2(x) = \frac{2}{\sqrt{2\pi}} e^{-\frac{w^2}{2s^2}}$, and $norm(\cdot)$ represents the cumulative distribution function for a random variable that is distributed as Gaussian distribution. The collision probability $p(s, w)$ decreases monotonically when s increases but grows monotonically when w rises.

The locality-preserving property of LSH allows us to partition the set of objects based on their hash values. If two points o_1 and o_2 are hashed to the same bucket, o_1 and o_2 are close to each other with certain confidence. However, it is possible that two distant points happen to be hashed to the same bucket according to Equation (1). To reduce such *false positives*, a group of m hash functions $G(\cdot) = \{h_1(\cdot), h_2(\cdot), \ldots, h_m(\cdot)\}$ are employed. That is, only points sharing all the m hash values are placed in the same bucket. Thus, each object o is labeled with a compound hash key $G(o) = \{h_1(o), h_2(o), \ldots, h_m(o)\}$, which is considered as the bucket key. The probability that two objects collide is reduced as shown in Equation (3).

$$\begin{aligned} Pr[G(o_1) = G(o_2)] &= \prod_{i=1}^{m} Pr[h_i(o_1) = h_i(o_2)] \\ &= p(s, w)^m \end{aligned} \tag{3}$$

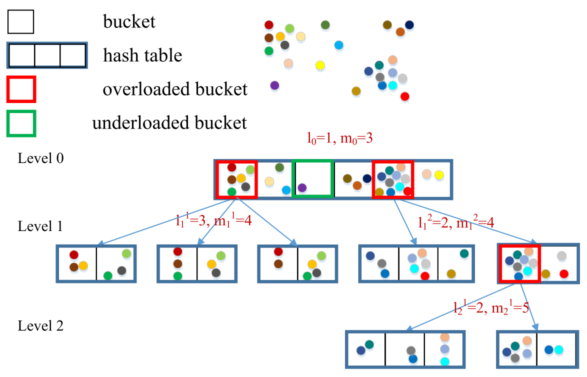


FIGURE 2. An illustrative example of LayerLSH structure (a bucket with more than 5 objects is considered as an overloaded bucket).

However, the probability $p(s, w)^m$ may be very small when m is large, which may lead to a large number of *false negatives*. In order to reduce the loss of false negatives, multiple hash tables are used. That is, a set of l hash groups $\{G_1(\cdot), G_2(\cdot), \ldots, G_l(\cdot)\}$ are employed and l hash tables are constructed (i.e., each object has l copies in l hash tables), hoping that the close points collide at least on one hash table. The final collision probability P is shown in Equation (4).

$$\begin{aligned} P &= 1 - \prod_{i=1}^{l} \left\{1 - Pr[G_i(o_1) = G_i(o_2)]\right\} \\ &= 1 - [1 - p(s, w)^m]^l \end{aligned} \tag{4}$$

III. LayerLSH: REBUILD BASIC LSH

As illustrated in Section I, the query falling in dense buckets tends to result in high cost, while the query falling in sparse buckets tends to result in low accuracy. Our idea is to split the dense buckets and merge the sparse buckets, which is simple but empirically shown to be effective (Section V). Suppose we have a set of 2-D data objects distributed as shown in the top of Figure 2. They are hashed to different buckets in two hash tables. Some of the buckets are lightly loaded, while some are heavily loaded. LayerLSH will rehash the objects residing in an overloaded bucket to a new set of hash tables, such that the overloaded bucket is rehashed into multiple groups of smaller buckets where each group corresponds to a new hash table. The overloaded buckets are rehashed recursively until no overloaded one exists. At meanwhile, the underloaded bucket will not be further processed but only be marked. When a query falls into the underloaded bucket, the query algorithm will simply expand the search scope and search the "nearby" buckets to improve the accuracy.

It is notable that since limiting bucket size will reduce the accuracy from probability theory's point of view. The objects in the overloaded bucket should be copied to more than one hash tables to compensate for the reduced accuracy. This is for sustaining the expected accuracy P as depicted in Equation (4), which will be described in detail in Section III-A. Since the overloaded buckets are rehashed recursively, multiple layers of LSH tables are constructed.

The root level (level 0) of the LayerLSH is exactly the same as the original LSH. The hash tables in higher levels are the new generated LSH tables for the rehashed buckets. Figure 2 shows an illustrative example of the ***multi-layered*** tree-like structure.

A. BUILDING LayerLSH

We rebuild the original hash tables in terms of two factors, the user specified expected recall and precision. Let KNN(q, O) denote the set of kNNs of q. Given a query q and a set of objects O, an approximate kNN query algorithm returns a set of candidates $\mathcal{C}$. We have the **recall** $\alpha = \frac{|\mathcal{C} \cap \text{KNN}(q,O)|}{|\text{KNN}(q,O)|}$, which implies the accuracy, and the **precision** $\beta = \frac{|\mathcal{C} \cap \text{KNN}(q,O)|}{|\mathcal{C}|}$, which implies the efficiency. Given α and β, we study the lower/upper bound size of each bucket.

Proposition 1 (Bucket Size Constraints): When using LSH with l hash tables to answer kNN query, with an expected recall α and an expected precision β, the bucket size S of each hash table has the following constraints:

$$k \cdot (1 - \sqrt[l]{1-\alpha}) \leq S \leq \frac{k}{\beta \cdot l}. \tag{5}$$

Proof: Suppose the LSH parameters are $\{l, m, w\}$. In each of the l hash table, the expected size of the bucket that q falls in is $\overline{S} = \sum_{o \in O} p(|q, o|, w)^m$ where $p(s, w)^m$ is defined in Equation (2) and (3). Let $s_{(q,k)}$ denote the distance from q to its kth NN. Then we have

$$\begin{aligned} \overline{S} = \sum_{o \in O} p(|q, o|, w)^m &\geq \sum_{o \in \text{KNN}(q,O)} p(|q, o|, w)^m \\ &\geq k \cdot p(s_{(q,k)}, w)^m \\ &\geq k \cdot (1 - \sqrt[l]{1-\alpha}), \end{aligned} \tag{6}$$

The first "$\geq$" is true because the candidates for summation is reduced from O to KNN(q, O). The second "$\geq$" is true because for any $|q, o| \leq s_{(q,k)}$ we have $p(|q, o|, w) \geq p(s_{(q,k)}, w)$ according to Equation (2). The third "$\geq$" is true because in order to achieve the overall recall α over l hash tables, the probability that q and any of its kNNs collide in a specific hash table should be no less than $1 - \sqrt[l]{1-\alpha}$ according to Equation (4). If we can make $p(s_{(q,k)}, w)^m \geq 1 - \sqrt[l]{1-\alpha}$ for q's kth NN, it is also true for any other kNNs. Thus, the bucket size S should satisfy $S \geq k \cdot (1 - \sqrt[l]{1-\alpha})$.

When all of q's kNNs are contained in the returned candidates set, i.e., KNN$(q, O) \subset \mathcal{C}$, $|\mathcal{C}|$ can be as large as $\frac{k}{\beta}$ to satisfy the expected precision, otherwise $|\mathcal{C}|$ has to be smaller. Thus, $\frac{k}{\beta}$ is the upper bound of $|\mathcal{C}|$. Since there are l hash tables, the candidates are retrieved from l buckets. Let us assume the candidates are collected evenly from l hash tables. Thus, the bucket size S should satisfy $S \leq \frac{k}{\beta \cdot l}$.

Note that, satisfying the above constraints does not necessarily guarantee the expected recall and precision but helps us identify the underloaded and overloaded buckets.

Given the constraints of the bucket size, we propose Algorithm 1 to recursively rehash the overloaded buckets to build LayerLSH index. The input includes the original

```
Algorithm 1: Building LayerLSH
  input  : LSH hash tables $HT = \{HT_1, \ldots, HT_l\}$,
           LSH parameters $\{l, m, w\}$, $k$, $R = \alpha$, $P = \beta$
  output: LayerLSH hash tables set
1 Function RecursSplit($HT$, $\{l, m, w\}$, $R$, $P$):
2     $R' = 1 - \sqrt[l]{1-R}$, $P' = P \cdot l$;
3     $T_l = k \cdot R'$, $T_u = \frac{k}{P'}$;
4     foreach $HT_i$ in $HT$ do
5         foreach bucket $b$ in $HT_i$ do
6             $S \leftarrow$ size of $b$;
7             if $S > T_u$ then
8                 $\{l_c, m_c, w_c\} \leftarrow$FindParam($\{l, m, w\}$, $S$,
                   $P'$);
9                 $HT_{child} \leftarrow$LSH($b$, $\{l_c, m_c, w_c\}$);
10                RecursSplit($HT_{child}$, $\{l_c, m_c, w_c\}$, $R'$, $P'$);
11            end
12            if $S < T_l$ then
13                mark $b$ as underloaded;
14            end
15        end
16    end
17 end
```

LSH tables (i.e., level-0 hash tables), the LSH parameters set $\{l, m, w\}$, the number of returned NNs k, the expected recall $R = \alpha$, and the expected precision $P = \beta$. With respect to each hash table, the recall is relaxed to $R' = 1 - \sqrt[l]{1-R}$, and the precision is tightened to $P' = P \cdot l$ (Line 2). Then, we have the lower bound size ($T_l = k \cdot R'$) and upper bound size ($T_u = \frac{k}{P'}$) for each bucket (Line 3). We check the size of each bucket of each hash table. For the overloaded bucket that contains more than T_u objects (Line 7), we first determine a new set of child LSH parameters (Line 8), based on which the objects in that bucket are rehashed into a new set of hash tables (Line 9). The overloaded buckets are rehashed by recursively invoking this process until no bucket is overloaded (Line 10). For the underloaded bucket that contains fewer than $\frac{k \cdot \alpha}{l}$ objects (Line 11), we mark it for future use in query processing (Line 12).

In Algorithm 1, we refer to the to-be-rehashed bucket as *parent bucket* and the new LSH for this bucket as *child LSH*. The core of bucket rehashing is to choose a proper new set of child LSH parameters, such that the bucket size is reduced (for efficiency) but at the same time the probability that a query and its kNNs collide in the same bucket is not reduced (for accuracy). We fix the LSH width parameter w (We will explain the reason later). Let $\{l_p, m_p, w\}$ denote the set of parent LSH parameters, and $\{l_c, m_c, w\}$ denote the set of child LSH parameters. We use the following propositions to guide the selection of child LSH parameters.

Proposition 2 (For Accuracy): Let $s_{(q,k)}$ denote the distance from a query q to its kth NN. Suppose we can find a r^* such that $r^* \geq s_{(q,k)}$ for any q. In order to guarantee the expected recall, the child LSH parameters $\{l_c, m_c\}$ should be chosen to satisfy:

$$1-(1-p^{m_p+m_c})^{l_c}=p^{m_p}, \tag{7}$$

where $p=p(r^*, w)$ is defined in Equation (2).

Proof: Given the definition of r^*, we have $\forall q, r^* \geq s_{(q,k)}$ and further have $p(r^*, w)^{m_p} \leq p(s_{(q,k)}, w)^{m_p}$. That is, the probability that any query q and its kth NN collide in the same bucket is no less than $p(r^*, w)^{m_p}$. If the probability $p(r^*, w)^{m_p}$ could be sustained after bucket rehashing, the probability that q and any of its kNNs fall in the same bucket will not reduce, so that the expected recall α is guaranteed. Accordingly, in terms of Equation (4), the child LSH parameters $\{l_c, m_c, w\}$ should be chosen such that $1-[1-p(r^*, w)^{m_p+m_c}]^{l_c} \geq p(r^*, w)^{m_p}$. Further, since all the objects in child hash tables are originated from the parent bucket, the collision probability in child hash tables will never be greater than $p(r^*, w)^{m_p}$. Therefore, we aim to make $1-[1-p(r^*, w)^{m_p+m_c}]^{l_c} = p(r^*, w)^{m_p}$.

In practice, r^* can be approximately estimated by sampling. A number of sample objects are randomly selected to calculate their exact kNNs. The median of their kth NNs' distances is used to estimate r^*.

Proposition 3 (For Efficiency): Let S denote the size of the overloaded bucket that q falls in. In order to approximately satisfy the precision constraint, the child LSH parameter m_c and l_c should be chosen as follows:

$$m_c = \left\lceil log_p\left(\frac{k}{S\cdot\beta\cdot l_p}\right)\right\rceil,$$
$$l_c \leq \left\lfloor \frac{k}{S_c^*\cdot\beta\cdot l_p}\right\rfloor, \tag{8}$$

where $p = p(r^*, w)$ is defined in Equation (2), β is the expected precision, and S_c^* is the biggest bucket size among all child hash tables.

Proof: The expected size of the bucket that q falls in is $\overline{S} = \sum_{o\in O} p(|q, o|, w)^{m_p}$. From this equation, we learn that the actual bucket size relates to two factors, 1) the number of "close" objects to q and 2) the probability that these "close" objects fall in q's bucket (determined by m_p and w). The more "close" objects to q, the more likely S is larger. The larger m_p is, the more likely S is smaller. Let $X = \{o : |q, o| \leq r^*\}$ denote the set of objects within a range of r^*, which are considered as "close" objects. In other words, $|X|$ implies q's density. Then we assume that the bucket size S is proportional to $|X|$, i.e., $S \propto |X|$. On the other hand, to simplify the analysis and obtain an approximate answer, we assume that the probability for all objects in X falling in q's bucket is proportional to $p(r^*, w)^{m_p}$ that is fixed for all objects in X, i.e., $S \propto p^{m_p}$ where $p = p(r^*, w)$. Thus, we have $S \propto |X|\cdot p^{m_p}$.

In order to satisfy the efficiency constraint $\frac{k}{S\cdot l_p} \geq \beta$, the bucket size S should be reduced to less than $\frac{k}{\beta\cdot l_p}$. Since $S \propto |X|\cdot p^{m_p}$, $|X|\cdot p^{m_p}$ should be correspondingly reduced to $|X|\cdot p^{m_p+m_c}$. Therefore, we should choose m_c to satisfy the following equation:

$$\frac{S}{\frac{k}{\beta\cdot l_p}} = \frac{|X|\cdot p^{m_p}}{|X|\cdot p^{m_p+m_c}}. \tag{9}$$

Since m_c should be an integer, we choose the result of the ceiling function, i.e., $m_c = \lceil log_p(\frac{k}{S\cdot\beta\cdot l_p})\rceil$ to sustain the inequality of precision constraint.

On the other hand, after rehashing we also need to limit the number of child hash tables in order to satisfy the efficiency constraint. Suppose $S_{(q,i)}$ is the size of a query q's bucket (or multiple child buckets if it points to child hash tables) in child hash table i, we need to make sure $\sum_{i=1}^{l_c} S_{(q,i)} \leq \frac{k}{\beta\cdot l_p}$. Further, suppose S_c^* is the biggest bucket size among all child hash tables, we have $\sum_{i=1}^{l_c} S_{(q,i)} \leq l_c * S_c^*$. Thus, by making $l_c * S_c^* \leq \frac{k}{\beta\cdot l_p}$, i.e., $l_c \leq \lfloor\frac{k}{S_c^*\cdot\beta\cdot l_p}\rfloor$ (since l_c should be an integer), we can satisfy the efficiency requirement.

By combining Proposition 2 and Proposition 3, we can obtain the available child LSH parameters m_c and l_c, which are used during bucket rehashing (Line 8 in Algorithm 1). However, there probably is no solution since the recall and precision requirements cannot be satisfied at the same time (i.e., l_c's lower bound is greater than its upper bound). Furthermore, S_c^* in Equation (8) is even unknown before bucket rehashing. In such a case, we first choose m_c according to Equation (8), ignore the upper bound of l_c, and generate enough more child hash tables to sustain accuracy constraint according to Equation (7). We will further satisfy the efficiency constraint during the query processing.

In the original LSH, it is required to set $\{l, m, w\}$, while in LayerLSH we use the expected recall R and the expected precision P in place of $\{l, m, w\}$. This is because R and P should be more user-friendly since the effectiveness of real LSH applications is usually evaluated by the expected recall and precision. In addition, the reason why we perform the analysis by fixing w is explained as follows. Given a query q, for any point, its probability (shown in Equation 2) to collide with q depends on its distance to q (i.e., s) and the partition width w. If adjusting w is allowed, regarding a particular point, its probability to collide with q would change after rehashing. The new probability $p(s, w)$ depends on a variable s since s is variant for different points. That means, the analysis in Proposition 2 or 3 should also consider s, the distance from a point to a query. Obviously, s is unknown in prior since query is unknown in prior. This will bring big challenges in analyzing accuracy and efficiency. Therefore, we propose to fix w.

B. QUERY PROCESSING

There are two kinds of buckets in LayerLSH, which should be differentiated during query processing. One kind that contains similar data objects, which are referred to as *data buckets*. Another kind contains the pointers to child hash tables, which are referred to as *pointer buckets*.

To answer a kNN query, we use Algorithm 2 to retrieve the candidates set from multi-layered hash tables. We first read

Algorithm 2: Query Processing in LayerLSH

```
input : query $q$, LSH tables $HT = \{HT_1, \dots, HT_l\}$, $k$,
        $R = \alpha$, $P = \beta$
output: $k$NN candidates set $\mathcal{C}$
Function LayerLSHQuery ($q$, $HT$, $R$, $P$, $\mathcal{C}$):
  $\{l, m, w\} \leftarrow$ load LSH parameters for $HT$;
  $R' = 1 - \sqrt[l]{1-R}$, $P' = P \cdot l$;
  $T_l = k \cdot R'$, $T_u = \frac{k}{P'}$;
  foreach $HT_i$ in $HT$ do
    $b \leftarrow$ locate bucket with hash key $\{h_1(q), \dots, h_m(q)\}$;
    if $b$ is a pointer bucket then
      $HT_{child} \leftarrow$ locate the child hash tables $b$ points to;
      LayerLSHQuery ($q$, $HT_{child}$, $R'$, $P'$, $\mathcal{C}$);
    end
    else if $b$ is a data bucket then
      if $b$ is underloaded then
        $B \leftarrow b$;
        while $|B| < T_l$ do
          $b'$ find "nearby" bucket from $b$;
          $B \leftarrow B \cup b'$;
        end
        $\mathcal{C} \leftarrow \mathcal{C} \cup B$;
      end
      else
        if $R$ is primary then $\mathcal{C} \leftarrow \mathcal{C} \cup b$;
        else if $P$ is primary then $\mathcal{C} \leftarrow \mathcal{C} \cup b'$ s.t. $|b'| \le T_u$;
        else $\mathcal{C} \leftarrow \mathcal{C} \cup b'$ s.t. $|b'| \le \frac{T_u + \frac{\sum_i^l |b_{(q,i)}|}{l}}{2}$;
      end
    end
  end
end
```

the LayerLSH parameters $\{l, m, w\}$ from the input LayerLSH tables (Line 2). With respect to each hash table, the recall is relaxed to R', and the precision is tightened to P' (Line 3). Then, we have the lower bound size ($T_l = k \cdot R'$) and upper bound size ($T_u = \frac{k}{P'}$) for each bucket (Line 4). Given a query q, we first compute its compound hash keys and project q to the bucket in a particular hash table (Line 6). If the positioned bucket is a pointer bucket, the query q is rehashed in multiple child hash tables along with the bucket-based R' and P' (Line 8), and the query algorithm is invoked recursively (Line 9).

If the positioned bucket is a data bucket and this bucket is underloaded (Line 11), we will expand the search scope and search the "nearby" buckets whose compound hash keys are slightly different. This searching scope is expanded to more and more buckets as soon as enough objects (T_l) are returned (Line 12-16). The idea of merging "nearby" sparse buckets is similar to multi-probe LSH [11]. Given the property of LSH, if an object is close to a query q but not hashed to the same bucket, it is likely to be in a bucket that is "close by" (i.e., the hash keys of the two buckets only differ slightly). LayerLSH also designates the "close by" buckets by applying a *hash perturbation vector* $\Delta = \{\delta_1, \delta_2, \dots, \delta_m\}$ (e.g., $\{+1, 0, \dots, 0\}$ or $\{0, -1, \dots, 0\}$) on the original compound hash key $G(q) = \{h_1(q), h_2(q), \dots, h_m(q)\}$ and obtains the nearby bucket $G(q) + \Delta$.

If the data bucket is not underloaded, the objects in that bucket are conditionally put into the candidates set $\mathcal{C}$ (Line 17). Recall that, it is possible that the specified recall and precision are in conflict with each other. It is required to return all candidates from all hash tables in order to satisfy the recall requirement, but also required to return at most T_u candidates from only a few hash tables to satisfy the precision requirement. LayerLSH will let users specify one primary choice, the expected recall α or the expected precision β. Then the query processing algorithm will correspondingly include all the objects in bucket to satisfy recall requirement (Line 18) or limit the number of returned candidates to satisfy precision requirement (Line 19). If both or none of recall and precision is primarily selected, LayerLSH balances these two factors and return at most $\frac{T_u + \frac{\sum_i^l |b_{(q,i)}|}{l}}{2}$ candidates, where T_u is for precision requirement and $\frac{\sum_i^l |b_{(q,i)}|}{l}$ is for recall requirement (where $b_{(q,i)}$ is q's bucket in the ith child hash table).

It is noticeable that the query might expand to more and more buckets as it goes deeper in the LayerLSH tree. However, the large number of checked buckets does not necessarily lead to large number of candidates since the checked buckets are much smaller. With regard to the dense buckets, LayerLSH narrows the search scope, as a result the search is more efficient. Rather than using a large number of hash tables to achieve high search quality, we can achieve the same search quality with a smaller number of level-0 hash tables. More hash tables are only created for the dense buckets. The hashing is more targeted in terms of data distribution.

C. STREAM DATA PROCESSING

To deal with dynamic data, LayerLSH needs to support continuous point insertions and deletions. A naive implementation could be letting LayerLSH check the bucket size after each insertion or deletion. However, this may result in too many unnecessary bucket splits. For instance, an insertion to a nearly full bucket followed by multiple deletions may result in unnecessary bucket split. To alleviate this problem, we introduce the *time window* concept and buffer these insertions/deletions within a time window range. In a time window, if the bucket size does not exceed a predefined *maximum tolerance* $(1 + \epsilon_m) \cdot T_u$, it will not be split, where $\epsilon_m > 0$ and T_u is the original bucket size upper bound. Otherwise, it will still be split. The maximum tolerance of bucket size implies the effect of buffering. The bigger the ϵ_m is, the more

Algorithm 3: Stream Insertions Processing in LayerLSH

```
input  : newly arrived point $p$, LayerLSH tables
         $HT = \{HT_1, \ldots, HT_l\}$, maximum tolerance
         param $\epsilon_m$, caching tolerance param $\epsilon_c$,
         time window $W$
output: updated LayerLSH tables $HT$

while true do
    blockread($p$);
    $t_c$ ← obtain current time;
    if $t_c - t_l \geq W$ then
        split the buckets in $HT$ whose sizes are
          bigger than the relaxed upper bound
          $(1 + \epsilon_c) \cdot T_u$;
        $t_l$ ← obtain current time;
    end
    status←StreamInsert ($p$, $HT$);
    if status == overloaded then
        $HT_{child}$ ← split the overloaded bucket and
          create child hash tables;
        Let $b$ point to $HT_{child}$;
    end
end

Function StreamInsert (p, HT):
    foreach $HT_i$ in $HT$ do
        $b$ ← locate bucket with hash key
          $\{h_1(p), \ldots, h_m(p)\}$;
        if b is a pointer bucket then
            $HT_{child}$ ← locate the child hash tables $b$
              points to;
            status←StreamInsert ($p$, $HT_{child}$);
            if status == overloaded then
                resplit $HT_{child}$ based on the updated
                  bucket size;
            end
        end
        else if b is a data bucket then
            if $|b| > (1 + \epsilon_m) \cdot T_u$ then
                return overloaded;
            end
        end
    end
    return success;
end
```

insertions can be buffered. Note that, if the overloaded bucket is already in a child hash tables, we will resplit the child hash tables based on the newly updated bucket size instead of recursively splitting the overloaded bucket. This is because, if the recursive split is used, the continuous insertions might result in very deep search path in terms of LayerLSH's tree-like structure, which could degrade the query performance.

At the end of each time window, all the buckets are evaluated to be determined whether they should be split by comparing their sizes to a *caching tolerance* $(1 + \epsilon_c) \cdot T_u$, where $0 < \epsilon_c < \epsilon_m$. We introduce the caching tolerance for avoiding unnecessary bucket splits after each time window. By introducing the time window based buffering, a large number of unnecessary bucket splits can be avoided so that the processing throughput is expected to be higher, at the expanse that the query cost can be increased due to the delayed bucket splits. The tradeoff between processing throughput and query cost can be adjusted by tuning the maximum tolerance parameter ϵ_m and caching tolerance parameter ϵ_c.

D. INDEX MAINTENANCE

In addition, LayerLSH can be implemented as a disk-based index for maintaining large data sets. Since the basic structure in LayerLSH is tree-like, it is straightforward to store the index using a tree structure. The internal nodes storing the child LSH parameters as well as the pointers to child buckets are maintained in an *index file*, and the leaf nodes storing the data points are maintained in a *data file*. Note that, a leaf node that stores a large bucket is written to multiple file blocks, and multiple leaf nodes storing multiple small buckets are written to a single file block to save space. Similar buckets are stored continuously in a file block to support "nearby" bucket search.

When answering queries, the internal nodes maintained in index file are loaded into memory for fast access, or part of them for large index. After the data buckets are positioned, the file blocks storing the candidate buckets are loaded into memory for distance measurements, followed by returning the approximate kNNs. To support insertion of a point, we first locate the file blocks that store the hashed buckets and then append the point data. Note that, request of a new file block might be needed if the returned file block is full. To support deletion of a point, we first locate the file blocks and label this point indicating its invalidation. A periodical recycling process is executed offline to recycle the file blocks where no valid data is contained.

IV. DISTRIBUTED IMPLEMENTATION

A. DISTRIBUTED LSH

The approach of supporting distributed NN search with both LSH and LayerLSH is straightforward. We use Hadoop MapReduce to implement distributed LayerLSH. The *map*() function invocation on a point o_i computes l hash keys $G_1(o_i), G_2(o_i), \ldots, G_l(o_i)$ to obtain the intermediate key-value pair of point oi, $\langle G_1(o_i), o_i\rangle, \langle G_2(o_i), o_i\rangle, \ldots, \langle G_l(o_i), o_i\rangle$. According to these intermediate key-value pairs, we can have l different partition results (see Section II-B), i.e., l Hash tables. We then send the buckets in these hash tables to the corresponding reducers, so that each *reduce*() will receive a subset of points (i.e. a bucket) under a specific LSH partition layout. Next, we judge whether the size of the bucket exceeds the threshold. If so, at the reducer, we recursively split until the predefined conditions are met. After the map/reduce operations, we can obtain the LayerLSH index. During the query phase, for each query point, we retrieve from multiple distributed reducers to obtain its l approximate KNN candidate sets. We then aggregate these candidate sets in a second MapReduce job, and finally select the top k as its nearest k neighbors at the single reducer.

Suppose we have n workers. A bucket with key bk is assigned to worker $\frac{H(bk)}{n}$ where $H(\cdot)$ is any hash function that maps a bucket key to an integer. As a query q arrives, its multiple hash values $G_1(q), G_2(q), \ldots, G_l(q)$ corresponding to multiple hash tables are first calculated. The query is then sent to multiple workers $\frac{H(G_j(q))}{n}$, $1 \leq j \leq l$ for local computation. The candidates obtained by local NN search are refined before being merged as a global candidate list, where the refining method could be extracting only the top k nearest neighbors. Due to the skewed data distribution, hot spots might exist in distributed LSH, while LayerLSH has the advantage of alleviating hot spot contention.

B. ALL-PAIRS COMPUTATION

All-pairs computation is a common preprocessing step in many applications, e.g., retrieving similarity matrix for learning data correlations [12], pruning distant neighbors for abstracting a graph structure [13], evaluating implicit properties for each data point [14], and so on. All-pairs computation is known as a computation intensive task, which requires N^2 distance measurements. This is extremely costly for large volume and high dimensional data. Since the all-pairs computation is often performed only based on the nearest neighbors in these applications, LSH is an ideal approximation method to optimize all-pairs computation. Furthermore, using distributed machines can further speedup the computation intensive task. The LSH buckets are distributed among multiple workers, where the all-pairs computation is performed locally within each bucket.

However, distributed LSH-based all-pairs computation suffers from the drawback of skewed bucket size distribution. The workers with dense buckets can be the stragglers, which can significantly slow down the whole process. Fortunately, LayerLSH can alleviate this impact by bounding the bucket size, while at the same time guaranteeing the accuracy. Moreover, we merge similar small buckets in order to not only improve the accuracy but also reduce the number of distributed tasks.

C. CASE STUDY: POINT DENSITY EVALUATION

We take point density evaluation as a use case for illustration. A point p_i's density ρ_i is defined as the number of neighbors within a radius R, i.e., $\rho_i = |\{p_j|\forall j, |p_i, p_j| \leq R\}|$. In this problem, the computation of ρ_i only depends on its nearest neighbors with distance to p_i less than R. Suppose the approximated density is $\hat{\rho}_i$. By using LSH, the probability $\Pr(\hat{\rho}_i = \rho_i)$ can be studied as follows.

Lemma 1: Given a point p_i and an LSH function $h(p_i) = \lfloor \frac{a \cdot p_i + b}{w} \rfloor$, the probability that p_i and its nearest neighbors set $\{p_j|\forall j, |p_i, p_j| \leq R\}$ are hashed to the same bucket is:

$$Pr\big[h(p_i) = h(p_j)|\forall j, |p_i, p_j| \leq R\big] \geq 1 - \frac{4R}{\sqrt{2\pi}w}. \quad (10)$$

Proof: Let us consider a number line, where each point is a real number. $y_i = a \cdot p_i + b$ is a point on the number line. By floor dividing w, the number line is divided into a sequence of w-width slots. According to the LSH function, all the points in the same w-width slot share the the same hash key. The points that are close to p_i are all hashed to the positions close to y_i on the number line. The position of y_i is important. If y_i is close to the center of the slot, it is more likely that all R-length neighbors of p_i are in the same slot.

According to the definition of p-stable distribution [7], given a d-dimensional random vector a each entry of which is chosen independently from a standard gaussian distribution $\mathcal{N}(0, 1)$, for two points p_i and p_j, the distance between their projections $|a \cdot p_i - a \cdot p_j|$ (here $|\cdot|$ means the absolute value) is distributed as $|p_i, p_j| \cdot x$, where x is the **absolute value** of a standard gaussian random variable. Therefore, for any p_j where $|p_i, p_j| < R$, we have $\max_j |y_i - y_j| = \max_j |a \cdot p_i - a \cdot p_j| < R \cdot x$.

Moreover, $y_i = a \cdot p_i + b$ is uniformly distributed in a certain slot. To ensure that y_i and all its R-length neighbors are in the same slot, y_i has to be located in the interval of $[\alpha w + Rx, (\alpha + 1)w - Rx)$ for some α. The probability that y_i resides in such an interval is $\frac{w-2Rx}{w} = 1 - \frac{2Rx}{w}$. It is worth noting that the random variable a for mapping the query y_i is the same as the random variable for mapping all its R-length neighbors. The probability density function of the absolute value of the standard gaussian distribution is $f_p(x) = \frac{2e^{-x^2/2}}{\sqrt{2\pi}}$, where $x \geq 0$. Therefore, the probability becomes $1 - \frac{2Rx}{w} = \int_0^{\infty}(1 - \frac{2Rx}{w})f_p(x)dx$, and a further calculation shows that the probability is $1 - \frac{4R}{\sqrt{2\pi}w}$.

By applying the LSH properties described in Equation (3) and Equation (4), we have the following theorem.

Theorem 1: With l groups of m hash functions, the probability is finally enlarged as

$$Pr[\hat{\rho}_i = \rho_i] \geq 1 - \left[1 - \left(1 - \frac{4R}{\sqrt{2\pi}w}\right)^m\right]^l. \quad (11)$$

Proof: After applying l groups of m hash functions, we will obtain l $\hat{\rho}_i^g$ values ($1 \leq g \leq l$). According to the definition of ρ_i, we have $\hat{\rho}_i^g \leq \max_g \hat{\rho}_i^g \leq \hat{\rho}_i$. If $\max_g \hat{\rho}_i^g \neq \rho_i$, then $\hat{\rho}_i^g \neq \rho_i$ for all $1 \leq g \leq l$.

From Lemma 1, under a single hash function the probability that p_i and all its R-length neighbors are hashed to the same bucket is at least $1 - \frac{4R}{\sqrt{2\pi}w}$. With a group of m LSH functions $G = (h_1, h_2, \ldots, h_m)$ applied on each point, only points sharing all the m hash values are placed in the same partition. Suppose $\hat{\rho}_i^g$ is the approximated density value for a specific hash function group $G_g(p_i)$. Due to the fact that

TABLE 1. Datasets used in the experiments.

data	instances	dim	size
KDD	145,751	75	33MB
Forest	581,012	55	77MB
Color	68,040	32	10MB
Audio	54,387	192	36MB
Mnist	60,000	256	41MB

each LSH function is independently and randomly selected, we have:

$$\begin{aligned}\Pr[\hat{\rho}_i^g = \rho_i] &= \Pr\big[G_g(p_i) = G_g(p_j)|\forall j, ||p_i, p_j|| \leq R\big]\\ &= \prod_{t=1}^{m} \Pr\big[h_t(p_i) = h_t(p_j)|\forall j, ||p_i, p_j|| \leq R\big]\\ &\geq \left(1 - \frac{4R}{\sqrt{2\pi} w}\right)^m\end{aligned}$$

Further, since the l groups of hash functions $G_g(1 \leq g \leq l)$ is independently and randomly generated, we have the following:

$$\begin{aligned}\Pr[\hat{\rho}_i = \rho_i] &= 1 - \prod_{g=1}^{l}\left(1 - \Pr\left[\hat{\rho}_i^g = \rho_i\right]\right)\\ &\geq 1 - \left[1 - \left(1 - \frac{4R}{\sqrt{2\pi} w}\right)^m\right]^l\end{aligned}$$

Therefore, users are allowed to specify an expected accuracy in density approximation. However, the unbalanced buckets allocation brings troubles in distributed computing. As shown in Section V-F), one or two stragglers significantly slow down the whole process. LayerLSH rehashes the overloaded buckets to alleviate this problem. Meanwhile, the theoretical accuracy can be guaranteed by choosing child LSH parameters in terms of Proposition 2.

V. EXPERIMENTS

The experiments were performed on a Ubuntu system equipped with one Intel(R) Xeon(R) 2.60GHz CPU, 32GB of memory.

Datasets and Queries: We evaluate our approach using five real datasets, including **KDD**,[2] **Forest**,[3] **Color**,[4] **Audio**,[5] and **Mnist**.[6] Properties of these five datasets are summarized in Table 1. We also generate five sets of queries from each dataset. We first evaluate the density of each point, which is the number of neighbors in a given radius, then extract the top 2% highest density points as **dense queries**, the top 2% lowest density points as **sparse queries**, and the randomly sampled 2% points as **random queries**.

Query Accuracy: Query accuracy is measured by *error ratio*. Given a query q, let $o_1^*, o_2^*, \ldots, o_k^*$ be the kNNs with respect to q. The approximated kNNs are $o_1, o_2, \ldots, o_k$. The approximate error ratio is computed as

$$ratio(q) = \frac{1}{k}\sum_{i=1}^{k}\frac{||q, o_i||}{||q, o_i^*||}. \tag{12}$$

So a small ratio implies high query accuracy. An average of the error ratios from all queries is used for evaluation.

Query Cost: We evaluate the query cost in terms of the number of candidates to be checked for distance measurement.

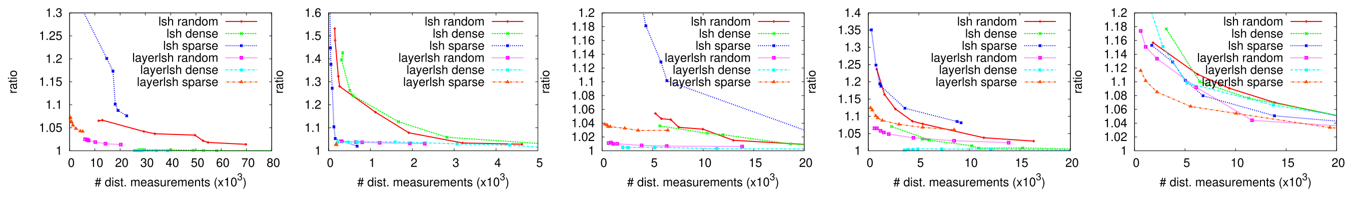
Parameters Setting: The original LSH indices for these datasets are first built with parameters $l = 3, m = 3$ (The effect of different l and m will be shown in Sec. V-D). w is set to satisfy a predefined accuracy which is different for different experiments. The number of returned NNs k is set to 20 unless particularly mentioned. r^* is estimated by sampling 1% data points as discussed in Section III-A, which differs with respect to k and dataset. The LayerLSH parameters are set as $\alpha = 0.9, \beta = 0.005$ unless particularly mentioned. And none of these two parameters are primarily chosen, so that the algorithm will perform query processing by "averaging" these two requirements as discussed in Section III-B.

A. OVERALL PERFORMANCE

Generally speaking, more query cost will result in high accuracy and vice versa. It does not make too much sense if comparing query cost or query accuracy independently, so we will show the query cost result and accuracy result in the same figure. We are trying to answer the question, which method results in the highest accuracy with the same query cost, or which method requires the lowest cost to achieve the same accuracy. We first build multiple LSH indices for these datasets by using different accuracy requirement parameters. By using different LSH indices, we expect to obtain various (cost, accuracy) pairs when answering kNN queries, which correspond to the multiple points in an (x, y)-plot. We then reconstruct the LSH indices and build their layered versions. By tuning the expected recall α and expected precision β, we can also create multiple layered indices with various (query cost, query accuracy) pairs.

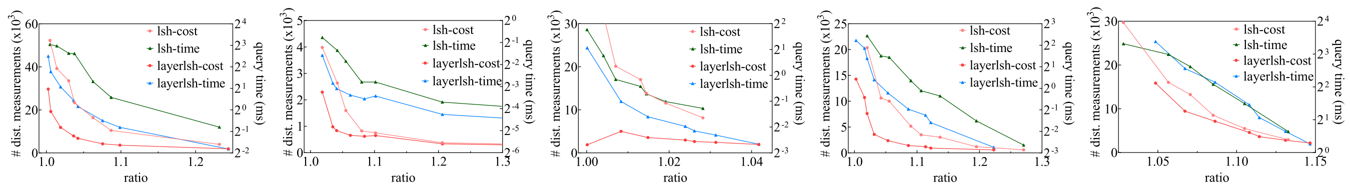
We show the results of LSH and LayerLSH when answering different types of queries in Figure 3. All the results are obtained by averaging 3 trials. The error ratio is lower with more query cost as expected. We can see that the query accuracy (reflected by error ratio) and the query cost (reflected by the number of candidates or distance measurements) vary a lot for different types of queries. With respect to the sparse queries, less accurate kNNs are returned and less number of distance measurements is required. With respect to the dense queries, more accurate kNNs are returned and more query cost is required. This is true for both LSH and LayerLSH. Figure 3 also shows the comparison results between LSH and LayerLSH. To consider both query accuracy and query cost, a curve that corresponds to an LSH index and a specific query type exhibits better performance when it

[2]http://www.kdd.org/kdd-cup/view/kdd-cup-2004/Data
[3]http://archive.ics.uci.edu/ml/datasets/Covertype
[4]http://kdd.ics.uci.edu/databases/CorelFeatures/
[5]http://www.cs.princeton.edu/cass/audio.tar.gz
[6]http://yann.lecun.com/exdb/mnist/



(a) KDD (b) Forest (c) Color (d) Audio (e) Mnist

FIGURE 3. **LSH vs. LayerLSH.**



(a) KDD (b) Forest (c) Color (d) Audio (e) Mnist

FIGURE 4. **Average cost and runtime comparison.**

is close to the bottom left corner. We can see that, for all types of queries, LayerLSH requires much less query cost (say 5%-20% of that of LSH) to achieve the same error ratio.

In addition, to further verify the superiority of our proposed method on average query performance, we show the average query time and query cost of LSH and LayerLSH in Figure 4. We vary the parameters to have multiple <ratio, query cost> pairs and <ratio, query time> pairs for LSH and LayerLSH, and draw four curves, each corresponding to LSH-cost, LSH-time, LayerLSH-cost, or LayerLSH-time. A curve exhibits better performance when it is close to the bottom left corner. As can be seen from these figures, LayerLSH can always achieve the same error rate with less time and cost than LSH.

B. SPACE CONSUMPTION

The space consumption is the size of the index file which is used to store the index. The space consumption of the basic LSH index and the LayerLSH index are listed in Table 2, where $l = 3, m = 3, \alpha = 0.9$ for both LSH and LayerLSH. Since the dense buckets are rehashed in extra hash tables and more copies of dense buckets exist, LayerLSH needs more space to store the extra indices. In LSH, using fewer hash tables is supposed to take up less space, but this is not true for LayerLSH. This is because that more overloaded buckets and much denser buckets could exist when using a small number of LSH tables.

In LayerLSH, the space consumption highly depends on the expected precision parameter β, which indicates the threshold for overloaded bucket. The bigger the β is, the smaller bucket is preferred. Thus, a larger number of small buckets and more child hash tables are expected to be created, so that more space for indexing these buckets and hash tables are required. As shown in Table 2, the index size is not increased too much which is acceptable. As will be seen later in Section V-D, we can achieve good enough accuracy and efficiency when setting $\beta = 0.5\%$

TABLE 2. **Space consumption for LSH and LayerLSH.**

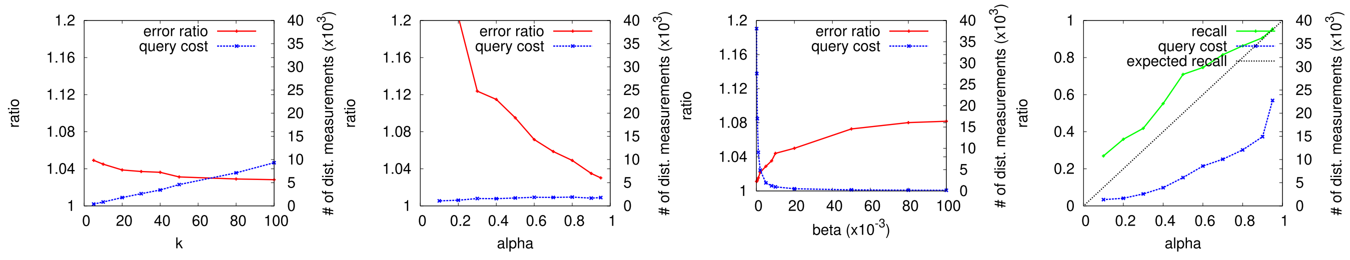
Dataset	LSH	LayerLSH $\beta = 0.1\%$	LayerLSH $\beta = 0.2\%$	LayerLSH $\beta = 0.5\%$
KDD	133MB	141MB	156MB	241MB
Forest	391MB	398MB	400MB	404MB
Color	27MB	29MB	39MB	70MB
Audio	126MB	128MB	130MB	144MB
Mnist	185MB	189MB	196MB	227MB

TABLE 3. **Rebuilding time for LayerLSH.**

Dataset	LSH	LayerLSH $\beta = 0.1\%$	LayerLSH $\beta = 0.2\%$	LayerLSH $\beta = 0.5\%$
KDD	1.35s	1.41s	4.1s	15.9s
Forest	3.67s	0.09s	0.1s	1.02s
Color	0.46s	0.43s	1.04s	3.21s
Audio	1.47s	0.5s	0.86s	2.35s
Mnist	2.2s	3.3s	6.62s	25.2s

C. REBUILDING TIME

Rebuilding LSH indices is not free. We need extra time for building LayerLSH index. We measure the rebuilding time for LayerLSH in this experiment. The amount of rebuilding effort is highly related to the parameter β, which indicates the threshold for overloaded bucket. A smaller β implies that higher query cost is tolerable, so that it is possible that only a small number of highly overloaded buckets are rehashed. In contrast, a bigger β implies that a large number of buckets are probably recognized as overloaded buckets and great efforts can be put on rehashing. Accordingly, the rebuilding time increases as β is increased.



(a) Effect of k (b) Effect of α (c) Effect of β (d) Effect of α on recall

FIGURE 5. Effect of parameters in LayerLSH (KDD).

We show the rebuilding time when $\beta = 0.1\%, 0.2\%, 0.5\%$ in Table 3. The LSH building time includes the data loading time and the index building time. The LayerLSH rebuilding time includes the LSH index loading time and the index rebuilding time. As can be seen, comparing to the original LSH indexing time, LayerLSH needs reasonable more time for rebuilding index for most datasets when β is small. It is expected to require longer rebuilding time when β is bigger. But as shown in the next experiment, by setting $\beta = 0.5\%$ the accuracy and efficiency can be both good enough.

TABLE 4. Effect of different *l* and *m*.

		$m=5$		$m=6$		$m=8$	
l		lsh	layerlsh	lsh	layerlsh	lsh	layerlsh
3	r	1.157	1.125	1.297	1.231	1.759	1.593
	c	1706	1578	690	683	232	206
5	r	1.093	1.087	1.138	1.114	1.573	1.379
	c	2900	1994	1501	1482	472	427
8	r	1.061	1.058	1.095	1.093	1.322	1.251
	c	3760	3433	2479	2081	603	548
10	r	1.041	1.038	1.069	1.062	1.291	1.182
	c	5873	4719	2634	2564	732	701

D. PARAMETER STUDIES

In this study, we investigate the parameters that potentially affect the performance of LayerLSH. These parameters include k, the expected recall α which implies the accuracy, and the expected precision β which implies the efficiency. The experiments are launched on KDD dataset, In order to comparably show the effects of these parameters, we show the error ratio and query cost in the same scale in Figure 5a, Figure 5b, and Figure 5c. Figure 5d shows recall rather than error ratio.

We first study the effect of k by varying k from 5 to 100 and fixing $\alpha = 0.9, \beta = 0.005$. Figure 5a shows the results. In general, the error ratio drops slightly when k increases, and the query cost increases when k increases. This is under expectation since we fix the expected precision $\beta = 0.005$ and the query cost will increase as k increases.

We next study the effect of α by varying α from 0.1 to 0.95 and fixing $k = 20, \beta = 0.005$. Figure 5b shows the results. The error ratio drops significantly, and the query cost increases slightly when α increases.

We also study the effect of β by varying β from 0.0001 to 0.1 and fixing $k = 20, \alpha = 0.9$. Figure 5c shows the results. The query cost drops dramatically when β increases. At the same time, the error ratio also increases as expected. We can learn that it is not suggested to set β too small when expecting a lower error ratio, since it is not worth due to the significant query cost.

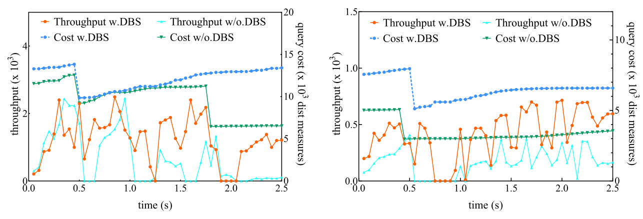
In addition, α is known as the user-specified expected recall. In order to see its effect on the real recall, we set α as the primary parameter (see Section III-B) and measure the recall rates when varying α. As shown in Figure 5d, LayerLSH can successfully guarantee the expected recall but at the expense of higher query cost.

In addition, we evaluate the effect of different l and m when comparing LSH and LayerLSH on the KDD dataset. The error ratio (abbrv. r) and query cost (abbrv. c) results are shown in Table 4. We can see that LayerLSH constantly outperforms LSH on both query accuracy and query cost when varying l and m. After bucket split, a large number of sparse buckets come up, which will trigger more nearby bucket search operations. This helps reduce the error ratio a lot.

E. HANDLING STREAM DATA

To illustrate LayerLSH's ability for handling stream data, we prepare a sequence of points from the KDD and Forest datasets and process these points one by one. The size of the initial dataset is randomly selected from the entire original dataset, whose size is 25% of that of the entire dataset. The maximum tolerance parameter $\epsilon_m = 0.6$ and the caching tolerance parameter $\epsilon_c = 0.2$. The time window W is set as 0.5s. We measure the processing throughput every 50ms time unit. Note that, the overloaded bucket will be split during this process, and the throughput could be affected. At the same time, we use another thread to query a specific point's kNNs after each insertion and record the number of returned candidates for distance measurements, which is considered as query cost.

The processing throughput and query cost results are shown in Figure 6, where Throughput w.DBS and Cost w.DBS are the throughput and query cost recorded. We can see that the processing throughput is reduced drastically after



(a) KDD (b) Forest

FIGURE 6. The processing throughput and query cost for a sequence of insertions.

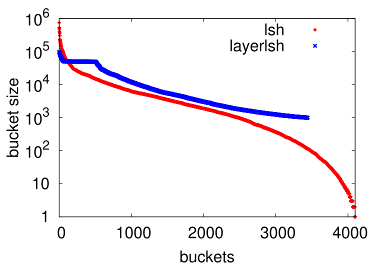
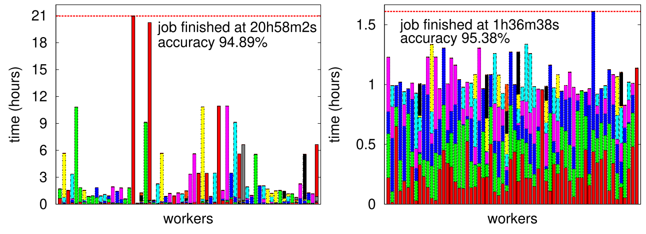


FIGURE 7. The distribution of bucket size in LSH and LayerLSH (BigCross).



(a) LSH (b) LayerLSH

FIGURE 8. The runtime of point density evaluation using Hadoop (BigCross).

each time window (0.5s) since many overloaded buckets will be split at that time. Note that, the throughput may also drop within a time window since some buckets might be seriously overloaded (i.e., the bucket size is larger than $(1 + \epsilon_m) \cdot T_u$) and cannot wait for the time window to end. The query cost for a specific point will be continuously increased until some of its host buckets are split. From this figure, this happens after the first time window (0.5s). It is probably because one of the query's host buckets is split. In addition, the throughput should be affected by ϵ_m and ϵ_c. We run a series of experiments and see that the throughput is increased when increasing ϵ_m. For $\epsilon_m = \{0.2, 0.4, 0.6, 0.8, 1\}$, the average throughput results on KDD dataset are $\{1.25, 1.49, 1.66, 1.77, 2.28\} \times 10^3$ pts/s.

To verify the effect of delay bucket split optimization, we turn off this optimization and show the results (i.e., Throughput w/o.DBS and Cost w/o.DBS) for comparison in Figure 6. As can be seen from these two figures, the query cost without delay bucket split optimization is reduced, but the throughput is getting lower more significantly, even 0 throughput during a few periods, e.g., 0.5-0.65 second period for KDD dataset.

F. DISTRIBUTED ALL-PAIRS COMPUTATION

In Section IV, we introduce a use case of distributed all-pairs computation, i.e., point density evaluation. We conduct experiments to show the benefit of our proposed multi-layered LSH structure in distributed computing. The experiment is performed in a large distributed cluster which contains 64 m1.medium Amazon EC2 instances. Each instance is equipped with 1 vCPU, 3.75GB memory, and 410GB disk. We utilize LSH and LayerLSH to partition the BigCross dataset, which contains 11,620,300 instances with 57 attributes for each, and compute the approximate point densities by using Hadoop MapReduce. The MapReduce implementation involves two jobs. The first performs LSH partition and all-pairs computation locally. The second job aggregates the local results. The LSH parameters are set as $l = 3, m = 3$. The expected accuracy is set to 0.95 such that w can be computed based on Equation (4), since the radius for evaluating point density is given. In LayerLSH, the bucket size limit is set as 10000 rather than being set according to precision rate, since this is not a kNN query application. In addition, similar sparse buckets are merged to further improve accuracy and to reduce number of partitions. The lower bound of bucket size is set as 1000. The number of reduce tasks is set as 256.

By applying LSH and LayerLSH, the bucket size distributions are depicted in Figure 7. The dense buckets are rehashed and the sparse buckets are merged in LayerLSH, so that the skewed buckets are balanced. As known, workload balance is crucial for distributed computing, which could bring significant performance gain especially in a very large scale distributed environment.

The runtime of reduce tasks are shown in Figure 8. Each color bar represents a reduce task, which performs local all-pairs computation. Each worker is assigned with multiple reduce tasks. The skewed distribution of bucket sizes leads to the skewed runtime of reduce tasks. We can see that the runtime of reduce tasks is seriously skewed when using LSH. While the runtime of reduce tasks in LayerLSH is more balanced. Accordingly, LSH-based point density evaluation requires much longer runtime than LayerLSH-based approach (20h58m2s vs. 1h36m38s). Since the rehashing strategy of LayerLSH does not reduce accuracy and the sparse buckets are merged, the accuracy is even higher by using LayerLSH.[7]

VI. RELATED WORK

A. LSH VARIANTS

The LSH functions based on Euclidean space are proposed by Datar et at. [7]. Since then, a large number of LSH variants were proposed for improving accuracy and reducing I/O cost,

[7]Because we cannot obtain the exact density values of all points in a reasonable time, we only show the sum of density values. In terms of point density's definition, the larger density should be more accurate.

including table-based LSH such as multi-probe LSH [11], entropy LSH [15], C2LSH [16], and tree-based LSH such as LSB-tree [17], LSH Forest [18], SK-LSH [19]. Besides them, numerous excellent works are proposed in recent years. LazyLSH [20] is able to answer approximate NN queries for multiple l_p metrics. It uses a single base index to support the computations in multiple l_p spaces. QALSH [21] introduces a novel concept of query-aware bucket partition which uses a given query as the anchor for bucket partition. SRS [22] requires only a single tiny index to answer approximate NN queries with theoretical guarantees. I-LSH [23] proposes an I/O efficient random hash based method, which obtains a good trade-off between search accuracy and I/O efficiency by using an incremental, rather than exponentially expanding, search strategy. SL-ALSH and S2-ALSH [24] support efficient approximate nearest neighbor search for multiple weighted distance functions with respect to the l_2 distance. Each of them introduce an asymmetric LSH family on top of E2LSH [7], and the asymmetric LSH families allow them to process each nearest neighbor query flexibly according to the weight vector attached to the query. PM-LSH [25] uses PM-Trees to index the data and improve the query processing time by using a tunable confidence interval to offer a higher accuracy of the results. Instead of using one-dimensional projections, R2LSH [26] uses two-dimensional projections and indexes the data by using B+-Trees, in the query processing phase, R2LSH [26] uses a query-centric ball to search the neighboring areas of the query and saves I/O costs. VHP [27] introduces the concept of virtual hypersphere partitioning and combines B-tree index to improve the search efficiency of unbounded and irregular space. LCCS-LSH [28] proposes a novel LSH scheme based on the Longest Circular Co-Substring (LCCS) search framework, which supports c-ANNS with different distance metrics. The LCCS search framework can make data objects that are closer together have longer LCCS than the far-apart ones. It is worth to mention another work that is called *Layered LSH* [29], which aims to distribute the hash buckets such that the search is likely to be performed on the same physical machine (hence network efficiency). It has different goal from us though has similar name.

B. DISTRIBUTED LSH

A set of research works focus on design efficient distributed LSH indices for supporting large data sets [30]–[32]. Zhang *et al.* utilize the locality preserving property of z-values and perform z-value based partition join in MapReduce to approximate the kNN joins [33]. Haghani *et al.* propose mappings from the multi-dimensional LSH bucket space to the linearly ordered set of peers that jointly maintain the indexed data, so that buckets likely to hold similar data are stored on the same or neighboring peers in a P2P system [34]. Bahmani *et al.* propose a distributed Entropy LSH implementation and prove that it exponentially decreases the network cost, while maintaining a good load balance between different machines [29]. PLSH [35] is a parallel LSH that supports high-throughput streaming of new data, which exploits an insert-optimized hash table structure and efficient data expiration algorithm for streaming data.

C. DATA DEPENDENT HASHING

As discussed in Section I, the recently proposed data sensitive hashing, e.g., DSH [8], selective hashing [9], ANN softmax [36], leverage data distributions. However, rather than learning the optimal hash functions from the skewed data, our approach relies on postprocessing and leverages the density of hash values to reorganize the existing structures. It is orthogonal to the data sensitive hashing. HashFile [37] also proposes to recursively partition the dense buckets. However, we use a multi-layered structure to organize the points as a general strategy that also benefits the tree-like LSH indices, which differs from it. OSimJoin [38] also proposes a recursive partitioning strategy, but its main purpose is to use locality-sensitive hashing to minimize the number of I/O operations between external memory and internal memory. Our goal is to improve the performance of locality-sensitive hashing itself by exploring the density of hash values. Second, OSimJoin needs to rehash each bucket to split the problem into sub-problems that fit into internal memory, while LayerLSH only rehashes dense buckets for dealing with skewed data. Moreover, OSimJoin only uses one hash function (m=1) to rehash each bucket, while LayerLSH uses multiple hash functions to rehash a dense bucket multiple times (the number of rehash times is not fixed but dynamic according to the data distribution) to ensure accuracy. The NSH [39] and our work share the same intuition that the limited hash bits should be used to better distinguish nearby items instead of capturing the distances among far apart items. However, NSH aims to devise a new hashing mechanism to achieve this goal, while we propose to reconstruct the existing LSH index structures as a postprocessing step. Learning-based hashing [40]–[42] has recently attracted many research efforts, which uses machine learning techniques to learn hash functions from a specific dataset so that the nearest neighbor search result in the hash coding space is as close as possible to the search result in the original space. Our LayerLSH is also significantly different from LSH Forest [18]: (1) LSH forest contains a set of LSH trees with different $\{l, m\}$ parameters, and each one is a logical prefix tree for the set of all labels, with each leaf corresponding to a point. While our LayerLSH is maintained in a single tree, where each node is either a data bucket containing a set of points or a pointer bucket containing the pointers to child hash tables. (2) Different from the LSH forest which contains multiple trees each with different $\{l, m\}$ parameters, LayerLSH is a single tree where the child nodes with the same parent node are a set of LSH buckets with different $\{l, m\}$ parameters.

VII. CONCLUSION

In this paper, we present the layered version of LSH variants by exploring the density of hash values. The dense buckets are rehashed and the sparse buckets are merged in order to make the hashing more targeted in terms of data distribution. We also discuss the possibilities of rebuilding other LSH variants and demonstrate the benefit in distributed computing. The experiment results have shown their effectiveness and efficiency. Specifically, LayerLSH can reach the same search quality as LSH with only 5%-20% query cost.

REFERENCES

[1] Z. Ren, Y. Gu, C. Li, F. Li, and G. Yu, "GPU-based dynamic hyperspace hash with full concurrency," *Data Sci. Eng.*, vol. 6, no. 3, pp. 1–15, 2021.

[2] G. Du, L. Zhou, Y. Yang, K. Lü, and L. Wang, "Deep multiple auto-encoder-based multi-view clustering," *Data Sci. Eng.*, vol. 6, no. 3, pp. 1–16, 2021.

[3] Y. Zhang, S. Cheny, and Y. Ge, "Efficient distributed density peaks for clustering large data sets in MapReduce," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, 2017, pp. 67–68.

[4] A. Mondal, A. Kakkar, N. Padhariya, and M. Mohania, "Efficient indexing of top-k entities in systems of engagement with extensions for geo-tagged entities," *Data Sci. Eng.*, vol. 6, no. 4, pp. 411–433, Dec. 2021.

[5] P. Liu, M. Wang, J. Cui, and H. Li, "Top-k competitive location selection over moving objects," *Data Sci. Eng.*, vol. 6, no. 4, pp. 392–401, Dec. 2021.

[6] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proc. 13th Annu. ACM Symp. Theory Comput.*, 1998, pp. 604–613.

[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. 20th Annu. Symp. Comput. Geometry*, 2004, pp. 253–262.

[8] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi, "DSH: Data sensitive hashing for high-dimensional k-nnsearch," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2014, pp. 1127–1138.

[9] J. Gao, H. V. Jagadish, B. C. Ooi, and S. Wang, "Selective hashing: Closing the gap between radius search and k-NN search," in *Proc. 21st ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2015, pp. 349–358.

[10] V. M. Zolotarev, "One-dimensional stable distributions," in *Translations of Mathematical Monographs*, vol. 65. Providence, RI, USA: American Mathematical Society, 1986.

[11] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: Efficient indexing for high-dimensional similarity search," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 950–961.

[12] P. J. Phillips, P. J. Flynn, T. Scruggs, K. W. Bowyer, J. Chang, K. Hoffman, J. Marques, J. Min, and W. Worek, "Overview of the face recognition grand challenge," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 1, Jun. 2005, pp. 947–954.

[13] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, pp. 972—976, Feb. 2007.

[14] R. McConville, X. Cao, W. Liu, and P. Miller, "Accelerating large scale centroid-based clustering with locality sensitive hashing," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 649–660.

[15] R. Panigrahy, "Entropy based nearest neighbor search in high dimensions," in *Proc. 17th Annu. ACM-SIAM Symp. Discrete Algorithm*, 2006, pp. 1186–1195.

[16] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," in *Proc. Int. Conf. Manage. Data*, 2012, pp. 541–552.

[17] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Quality and efficiency in high dimensional nearest neighbor search," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2009, pp. 563–576.

[18] M. Bawa, T. Condie, and P. Ganesan, "LSH forest: Self-tuning indexes for similarity search," in *Proc. 14th Int. Conf. World Wide Web*, 2005, pp. 651–660.

[19] Y. Liu, J. Cui, Z. Huang, H. Li, and H. T. Shen, "SK-LSH: An efficient index structure for approximate nearest neighbor search," *Proc. VLDB Endowment*, vol. 7, no. 9, pp. 745–756, May 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p745-liu.pdf

[20] Y. Zheng, Q. Guo, A. K. H. Tung, and S. Wu, "LazyLSH: Approximate nearest neighbor search for multiple distance functions with a single index," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 2023–2037.

[21] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "query-aware locality-sensitive hashing for approximate nearest neighbor search," *Proc. VLDB Endowment*, vol. 9, no. 1, pp. 1–12, Sep. 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1-huang.pdf

[22] 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," *Proc. VLDB Endowment*, vol. 8, no. 1, pp. 1–12, Sep. 2014. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1-sun.pdf

[23] W. Liu, H. Wang, Y. Zhang, W. Wang, and L. Qin, "I-LSH: I/O efficient c-approximate nearest neighbor search in high-dimensional space," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Apr. 2019, pp. 1670–1673.

[24] Y. Lei, Q. Huang, M. Kankanhalli, and A. Tung, "Sublinear time nearest neighbor search over generalized weighted space," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 3773–3781.

[25] B. Zheng, X. Zhao, L. Weng, N. Q. V. Hung, H. Liu, and C. S. Jensen, "PM-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search," *Proc. VLDB Endowment*, vol. 13, no. 5, pp. 643–655, Jan. 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p643-zheng.pdf

[26] K. Lu and M. Kudo, "R2LSH: A nearest neighbor search scheme based on two-dimensional projected spaces," in *Proc. IEEE 36th Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 1045–1056.

[27] K. Lu, H. Wang, W. Wang, and M. Kudo, "VHP: Approximate nearest neighbor search via virtual hypersphere partitioning," *Proc. VLDB Endowment*, vol. 13, no. 9, pp. 1443–1455, May 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1443-lu.pdf

[28] Y. Lei, Q. Huang, M. Kankanhalli, and A. K. H. Tung, "Locality-sensitive hashing scheme based on longest circular co-substring," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 2589–2599.

[29] B. Bahmani, A. Goel, and R. Shinde, "Efficient distributed locality sensitive hashing," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2012, pp. 2174–2178.

[30] J. Wu, L. Shen, and L. Liu, "LSH-based distributed similarity indexing with load balancing in high-dimensional space," *J. Supercomput.*, vol. 76, no. 1, pp. 636–665, Jan. 2020.

[31] P. Zhang, H. Pan, Z. Li, P. Cui, R. Jia, P. He, Z. Zhang, G. Tyson, and G. Xie, "NetSHa: In-network acceleration of LSH-based distributed search," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 9, pp. 2213–2229, Sep. 2022.

[32] P. Zhang, H. Pan, Z. Li, P. He, Z. Zhang, G. Tyson, and G. Xie, "Accelerating LSH-based distributed search with in-network computation," in *Proc. IEEE Conf. Comput. Commun. (IEEE INFOCOM)*, May 2021, pp. 1–10.

[33] C. Zhang, F. Li, and J. Jestes, "Efficient parallel kNN joins for large data in MapReduce," in *Proc. 15th Int. Conf. Extending Database Technol.*, 2012, pp. 38–49.

[34] P. Haghani, S. Michel, and K. Aberer, "Distributed similarity search in high dimensions using locality sensitive hashing," in *Proc. 12th Int. Conf. Extending Database Technol. Adv. Database Technol.*, 2009, pp. 744–755.

[35] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey, "Streaming similarity search over one billion tweets using parallel locality-sensitive hashing," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1930–1941, Sep. 2013.

[36] K. Zhao, L. Song, Y. Zhang, P. Pan, Y. Xu, and R. Jin, "ANN softmax: Acceleration of extreme classification training," *Proc. VLDB Endowment*, vol. 15, no. 1, pp. 1–10, Sep. 2021. [Online]. Available: http://www.vldb.org/pvldb/vol15/p1-zhao.pdf

[37] D. Zhang, D. Agrawal, G. Chen, and A. K. H. Tung, "HashFile: An efficient index structure for multimedia data," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 1103–1114.

[38] R. Pagh, N. Pham, F. Silvestri, and M. Stöckel, "I/O-efficient similarity join," *Algorithmica*, vol. 78, no. 4, pp. 1263–1283, Aug. 2017.

[39] Y. Park, M. Cafarella, and B. Mozafari, "Neighbor-sensitive hashing," *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 144–155, Nov. 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p144-park.pdf

[40] J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen, "A survey on learning to hash," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 4, pp. 769–790, Apr. 2018.

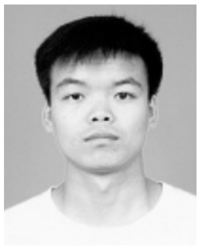
[41] Q. Tan, N. Liu, X. Zhao, H. Yang, J. Zhou, and X. Hu, "Learning to hash with graph neural networks for recommender systems," in *Proc. Web Conf.*, Apr. 2020, pp. 1988–1998.

[42] P. Li, L. Han, X. Tao, X. Zhang, C. Grecos, A. Plaza, and P. Ren, "Hashing nets for hashing: A quantized deep learning to hash framework for remote sensing image retrieval," *IEEE Trans. Geosci. Remote Sens.*, vol. 58, no. 10, pp. 7331–7345, Oct. 2020.

JIWEN DING received the bachelor's and master's degrees in computer science from Northeastern University, China, where he is currently pursuing the Ph.D. degree. His research interests include big data management and data mining.

ZHUOJIN LIU is currently pursuing the master's degree with Northeastern University, China. Her research interests include data mining, data management, and GPU acceleration.

YANFENG ZHANG (Member, IEEE) received the Ph.D. degree in computer science from Northeastern University, China, in 2012. He is currently an Associate Professor with Northeastern University. His research interests include distributed systems and big data processing. He has published many articles in the above areas. His paper in Socc 2011 was honored with the "Paper of Distinction."



SHUFENG GONG received the Ph.D. degree in computer science from Northeastern University, China. He is currently a Lecturer at Northeastern University. His research interests include distributed systems and graph processing.

GE YU (Senior Member, IEEE) received the Ph.D. degree in computer science from Kyushu University, Japan, in 1996. He is currently a Professor with Northeastern University, China. He has published more than 200 papers in refereed journals and conferences. His current research interests include distributed and parallel systems, cloud computing and big data management, and blockchain techniques and systems. He is a fellow of CCF and a member of ACM.