DLSH: A Distribution-aware LSH Scheme for Approximate Nearest Neighbor Query in Cloud Computing

Yuanyuan Sun
Huazhong University of Science and Technology, China
sunyuanyuan@hust.edu.cn

Yu Hua
(Corresponding Author)
Huazhong University of Science and Technology, China
csyhua@hust.edu.cn

Xue Liu
McGill University, Canada
xueliu@cs.mcgill.ca

Shunde Cao
Huazhong University of Science and Technology, China
csd@hust.edu.cn

Pengfei Zuo
Huazhong University of Science and Technology, China
pfzuo@hust.edu.cn

ABSTRACT

Cloud computing needs to process and analyze massive high-dimensional data in a real-time manner. Approximate queries in cloud computing systems can provide timely queried results with acceptable accuracy, thus alleviating the consumption of a large amount of resources. Locality Sensitive Hashing (LSH) is able to maintain the data locality and support approximate queries. However, due to randomly choosing hash functions, LSH has to use too many functions to guarantee the query accuracy. The extra computation and storage overheads exacerbate the real performance of LSH. In order to reduce the overheads and deliver high performance, we propose a distribution-aware scheme, called DLSH, to offer cost-effective approximate nearest neighbor query service for cloud computing. The idea of DLSH is to leverage the principal components of the data distribution as the projection vectors of hash functions in LSH, further quantify the weight of each hash function and adjust the interval value in each hash table. We then refine the queried result set based on the hit frequency to significantly decrease the time overhead of distance computation. Extensive experiments in a large-scale cloud computing testbed demonstrate significant improvements in terms of multiple system performance metrics. We have released the source code of DLSH for public use.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09...$15.00
https://doi.org/10.1145/3127479.3127485

CCS CONCEPTS

• **Information systems** → **Nearest-neighbor search**; *Information retrieval query processing*; • **Theory of computation** → *Bloom filters and hashing*; • **Computer systems organization** → Cloud computing;

KEYWORDS

Approximate nearest neighbor query, locality sensitive hashing, storage systems, cloud computing

ACM Reference Format:
Yuanyuan Sun, Yu Hua, Xue Liu, Shunde Cao, and Pengfei Zuo. 2017. DLSH: A Distribution-aware LSH Scheme for Approximate Nearest Neighbor Query in Cloud Computing. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 14 pages.
https://doi.org/10.1145/3127479.3127485

1 INTRODUCTION

In cloud computing systems, massive high-dimensional data need to be fast processed and well analyzed. According to the report of International Data Corporation (IDC) in 2014, the data we create and copy is doubling in size every two years from now until 2020. By then, the size of data will reach 44 ZettaBytes [40], which needs to be handled in a real-time manner. The popular use of mobile devices accelerates the production of large amounts of data. There exist 1.51 billion mobile active users on Facebook in March 2016, with an increase of 21% year-over-year [2]. Moreover, industrial companies have already been dealing with petabytes-scale data everyday [1, 2].

Cloud computing systems consume a large amount of system resources, such as computation, storage and networks, to support query-related requests. However, it is still challenging to return accurate queried results in a real-time manner [13, 18, 22, 41]. In order to address this challenge, many researchers focus on query services in the cloud computing communities, such as search tree for multiple-set membership testing [47], lock-free logarithmic search tree based on MDList [49], query authentication and correction on outsourcing data [39], photo search for preserving privacy [50], real-time search for files [46], keyword search over encrypted

data [8, 19, 35, 36], distributed query processing in RFID-enabled supply chains [21], pattern matching string search on encrypted cloud systems [42] and automating model search for machine learning [34]. However, these schemes suffer from space inefficiency and high-complexity hierarchical addressing, thus failing to support real-time queries.

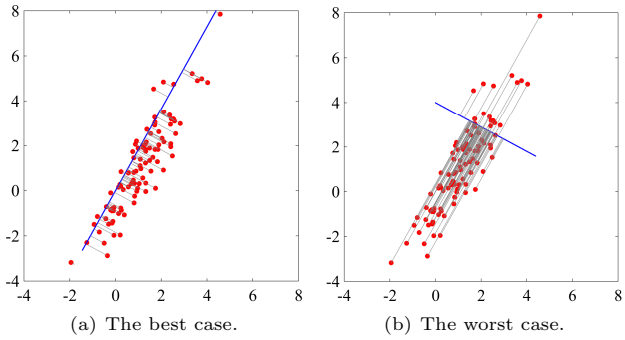
Obtaining clear and accurate results for query requests is time-consuming and not easy-to-use to users, who usually fail to provide accurate descriptions for query requests. Hence, approximate near neighbor (ANN) query service has received many attentions in practical applications due to the real-time property. Locality Sensitive Hashing (LSH) [15] and its variants [6, 7, 12, 14, 17, 24, 30] have been widely used to support ANN query due to the simplicity of hash computation and the maintenance of data locality. The basic idea of LSH is to hash similar points into the same bucket with a high probability, as well as hash different points into the same bucket with a low probability. For basic LSH and its variants, hash functions project data points over randomly chosen directions, which are independent of data distributions. In order to maintain the accuracy of approximate queries, multiple hash tables have to be utilized, resulting in space-inefficiency.

The essential reason of the space-inefficiency of LSH-based queries is that the LSH functions are unaware of data distributions. The "one-size-fit-all" methodology fails to efficiently meet the needs of real-world applications, thus having to use multiple hash tables to maintain data locality and guarantee the query accuracy. For example, entropy-based LSH [29] uses hundreds of hash tables to achieve good search quality. Multi-probe LSH [24] uses derived probing sequences to probe multiple hash buckets, resulting in partly reducing space overhead. Figure 1 shows two projected results of the identical 100 data samples following a Gaussian distribution, with different projection vectors in the 2-dimensional space. Our design goal is to differentiate the aggregated data in a suitable direction, which faithfully exhibits the data locality as well as efficiently decreases the hash collisions. As shown in Figure 1(a), the projection vector in the best case is efficient to identify correlated data. However, as shown in Figure 1(b), it is difficult to accurately obtain them, since almost all data have been projected into one small area.

Existing efforts improve the performance of hash-based schemes by using data-dependent hash functions, such as spectral hashing [43], shift-invariant kernel hashing [32], complementary hashing [45], isotropic hashing [16], hashing with decision trees [20], circulant binary embedding [48] and supervised discrete hashing [33]. Due to overlooking the data locality, these hashing schemes fail to provide real-time query services.

To deliver high performance and support real-time ANN queries, we need to deal with three main challenges.

- **Low Accuracy**. The basic LSH and its variants [12, 14, 17, 24, 30] choose the projection vectors of hash functions without taking the data distribution into account. For uniformly-distributed data points, they are hashed into buckets with equal probability, and thus points in buckets are uniform. However, in practice, the distribution of descriptors of data in cloud computing applications is far from being uniform [31, 33]. Hence, unevenly distributed points are hashed by randomly-chosen directions, which results in the aggregation of many irrelevant points and decreases the accuracy of query operations.
- **Space Inefficiency**. Projection vectors of LSH functions are independent of data distributions. Conventional LSH-based schemes heavily depend on using many hash tables to guarantee the accuracy of queries. The heavy memory consumption of hash tables becomes the performance bottleneck of basic LSH and its variants.
- **High Query Latency**. Because of random projection vectors, many irrelevant points are probed and stored in the result set of a query. Hence, the result set contains too many candidates for the following distance computation, which is time-consuming and high query latency for cloud users.



(a) The best case. (b) The worst case.

Figure 1: An example of projection difference in 2-d space.

In order to address the challenges, a Distribution-aware Locality Sensitive Hashing (DLSH) scheme is proposed for cloud computing to guarantee query accuracy, improve the space efficiency and reduce query latency. The idea behind DLSH is to first explore and exploit the data distribution, which is further used to judiciously select hash functions, and then refine the queried result set. Specifically, in order to maintain the data locality and reduce the space overhead, we leverage Principal Component Analysis (PCA) [3] into the Distribution-aware LSH. The basic idea of PCA is to maintain the maximum standard deviation of projection vectors in datasets, as well as reduce the dimensionality of data.

By using the principal components of the data distribution as the projection vectors of hash functions, we hence select, rather than randomly choose, hash functions, thus reducing the number of hash functions used and space overhead. Unfortunately, conventional PCA fails to offer successive projections and eliminate all irrelevant points of queries due to the orthogonality constraints, thus not meeting the needs of approximate queries in cloud computing. Hence, we put

forward a refinement method of queried result set based on traditional PCA by utilizing the corresponding eigenvalues of principal components to quantify the weight of each hash function, adjusting the interval value in each hash table and then recording the hit frequency of candidates to downsize the result set, resulting in significantly decreasing the time overhead of following distance computation. Specifically, this paper has made the following contributions.

- **Improving the Accuracy of ANN Queries**. We take the data distribution into account and propose a distribution-aware LSH scheme, called DLSH, in cloud computing systems. DLSH takes advantage of principal components of data distribution, to construct hash functions, rather than randomly-chosen projection vectors. In order to significantly reduce the PCA computation overhead, we compute the covariance matrix of a dataset in an offline manner to obtain the projection vectors, since we obtained the dataset in advance. Unlike conventional PCA, we further quantify the weight of each hash function and adjust the interval value in each hash table to refine the queried result set. This scheme can significantly improve the accuracy of ANN queries.
- **Improving the Space Efficiency and Decreasing Query Latency**. Through taking the data distribution into account, a small number of hash tables is enough to guarantee the query accuracy, which improves space efficiency. In query process, DLSH takes advantage of the hit frequency of candidates to further reduce the number of candidate results, which can eliminate a lot of irrelevant points. The refinement process cuts down the time overhead of distance computation, which decreases the query latency.
- **Practical Implementation**. We have implemented the DLSH prototype that is compared with the basic E2LSH [4, 9] and Principal Component Hashing (PCH) [25], in a large-scale cloud computing testbed. We utilize three widely used datasets to examine the practical performance of the proposed DLSH. We further propose a new metric for ANN query, named weighted recall, to differentiate the contribution of each queried ANN for query performance. The results demonstrate significant performance improvements in terms of accuracy and time overhead of ANN queries and space overhead of hash tables.

The rest of this paper in organized as follows. Section 2 presents the research background. Section 3 shows the DLSH design and practical operations. Section 4 illustrates the performance evaluation, and Section 5 presents the related work. Finally, we conclude our paper in Section 6.

2 BACKGROUND

In this section, we present the research background of locality sensitive hashing and principal components analysis for approximate nearest neighbor query.

2.1 (R, c)-Nearest Neighbor Query

Suppose A is a set of data points in d-dimensional space and points a, $b \in A$. $D(a, b)$ is the distance between a and b.

Definition 2.1. **(R, c)-NN Query**. Given parameters R, c and a query point q, for $\forall\ a \in A$, if $D(q, a) \leq cR$, Point a is defined as an ANN of Point q, where $c\ (> 1)$ is called approximation ratio and R is a tolerable distance threshold.

Points a and q with d-dimensional attributes can be represented as vectors $\overrightarrow{a_d}$ and $\overrightarrow{q_d}$, respectively. If their distance $D(a, q)$ is smaller than a pre-defined constant R, Point a is called an approximate nearest neighbor (ANN) of Point q. All ANN points of the query point q in a particular set compose the set of query result. The distance can be represented in many forms, such as the well-known Euclidean and Manhattan distance.

2.2 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) introduced by Indyk and Motvani in [15] has the property that maps close-by points into the same hash bucket in hash tables with a higher probability than distant ones. LSH is an efficient approximate algorithm for high-dimension similarity search. In order to support ANN query, given a query point q, we first hash it into buckets in multiple hash tables, and then gather all points in those hit buckets by ranking them in increasing order of their distances to the query point q. We eventually select the first k points for a query request, which is called kNN query. In each function of LSH family, similar points have a higher chance of colliding into the same hash bucket than that are far apart. Formally, the LSH family can be defined as follows [5]:

Definition 2.2. **LSH Function Family**. Let U be a set of hash values, and S be the set of n points, $S \subset R^d$. $H = \{h : S \to U\}$ is the hash function family. For $\forall p, q \in S$,

- If $D(p, q) \leq R$, then $P_{rH}[h(p) = h(q)] \geq P_1$.
- If $D(p, q) > cR$, then $P_{rH}[h(p) = h(q)] \leq P_2$.

where $c > 1$ and $P_1 > P_2$ for ANN query. Hence, the LSH family is called $(R, cR, P_1, P_2) - sensitive$.

The hash functions in H can be defined as:

$$h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{\omega} \rfloor, \tag{1}$$

where $\lfloor \cdot \rfloor$ is floor function, a is a d-dimensional random vector following a p-stable distribution, and b is a real number chosen uniformly from the range $[0, \omega)$, in which ω is a large constant.

The hash function projects high-dimensional points onto the vector a and maps the inner products into hash buckets with the interval ω. The parameter b can be regarded as the offset.

Random projections are leveraged to generate the family of hash functions. Intuitively, the close-by points in the high-dimensional space will also be close-by in all random projections. However, the probability is very small while two close points in high-dimensional space are close in all directions of

projection space, if enough number of projections are used. Hence, we need to enlarge the gap between P_1 and P_2. In practice, a concatenation of LSH functions is leveraged to generate a hash value for each point in R^d. Formally,

Definition 2.3. **Concatenation of LSH Functions**. $G = \{g : R^d \to U^k\}$ is a concatenation of LSH functions, in which $\forall g_i \in G$ consists of a sequence of k hash functions randomly chosen from H. Namely,

$$g_i(p) = (h_{i1}(p), h_{i2}(p), \ldots, h_{ik}(p)), \tag{2}$$

where k is the number of functions in each concatenation and $h_{i1}, h_{i2}, \ldots, h_{ik}$ are randomly chosen from the LSH Family H.

We leverage the concatenation of LSH functions to construct hash tables. Hence,

- If $D(p,q) \leq R$, then $P_{rH}[g(p) = g(q)] \geq P_1^k$.
- If $D(p,q) > cR$, then $P_{rH}[g(p) = g(q)] \leq P_2^k$.

The expected number of points in S that collide with Point q in a hash table but are far from cR in space is less than $P_2^k * |S|$. Moreover, the recall for one hash table is P_1^k, which is not large in real-world applications [9, 11]. In order to increase the overall recall, l concatenations of LSH functions are applied simultaneously to construct l hash tables. Thus, if $D(p,q) \leq cR$, the possibility that Point p collides with Point q in at least one hash table is at least $1 - (1 - P_1^k)^l$, which is close to 1.

Figure 2 shows an example to illustrate the LSH scheme. LSH has the property that hashes near neighbor points (q and a) into the same or adjacent bucket in hash tables with a high probability, and hashes distant points (q and b) into different buckets.

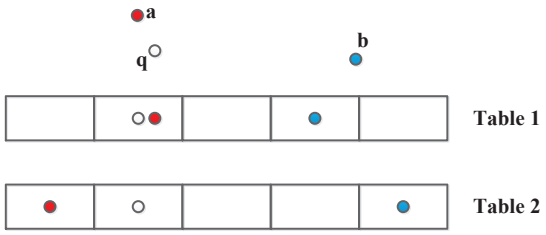


Figure 2: An example of the LSH scheme.

When the hash values of a query q are $g_i(q)(1 \leq i \leq l)$, we should gather the candidate points that are collide with q in $\cup_{i=1}^{l} B_{ig_i(q)}$, which have at least one hash value $g_i(q)$. However, the union of $B_{ig_i(q)}$ contains too many candidates for time-consuming distance computation.

LSH has been successfully used in approximate nearest neighbor queries in high-dimensional vector space. In practice, we need to configure two main parameters in LSH for approximate queries, namely, l, the number of hash tables, and k, the number of hash functions in each hash table. Although there exists the theoretical guarantee for the recall in the conventional LSH, it is inefficient to space and time overheads due to the use of multiple hash tables with multiple hash functions [15]. Moreover, the projections of LSH functions are chosen randomly which are independent of data distributions, resulting in a high probability of bad cases like Figure 1(b). We have to utilize multiple hash tables to guarantee the query accuracy, which is space inefficient. We hence take into account the principal components of data distributions to decrease space overhead.

2.3 Principal Component Analysis

Principal component analysis [3, 25, 51] is a widely-used technique of analyzing and simplifying datasets in multivariate statistical analysis. PCA is usually used for dimension reduction, feature extraction, and image coding and enhancement, in which a number of related variables are transformed into a set of uncorrelated variables. The basic idea of PCA is to reduce the dimensionality of data, as well as maintain the maximum standard deviation of projection vectors in datasets. PCA is calculated through eigenvalue decomposition of a data covariance matrix, then gets the eigenvectors and eigenvalues, usually after mean centering and normalizing the data matrix for each attribute [3].

Considering a data set X contains n vectors, where each vector is composed of d variables:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1d} \\ x_{21} & x_{22} & \ldots & x_{2d} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{nd} \end{bmatrix}. \tag{3}$$

To obtain principal components, we calculate the means of the variables. The vector of the means is:

$$\overline{\mathbf{X}} = \begin{bmatrix} \overline{x}_1 \\ \overline{x}_2 \\ \vdots \\ \overline{x}_d \end{bmatrix}, \tag{4}$$

and the covariance matrix is $S = \frac{1}{n-1} X^T X$, namely,

$$\mathbf{S} = \begin{bmatrix} S_1^2 & S_{12} & \ldots & S_{1m} \\ S_{12} & S_2^2 & \ldots & S_{2m} \\ \vdots & \vdots & \ldots & \ldots \\ S_{1m} & S_{2m} & \ldots & S_m^2 \end{bmatrix}, \tag{5}$$

where S_i^2 is the variance and the covariance is:

$$S_{ij} = \frac{n \sum_{k=1}^{n} x_{ik} x_{jk} - \sum_{k=1}^{n} x_{ik} \sum_{k=1}^{n} x_{jk}}{[n(n-1)]}. \tag{6}$$

The eigenvalues N and the eigenvectors V can be obtained from the covariance matrix S, and vectors in V are arranged by the decreasing order of corresponding eigenvalues in N. We usually selected first several eigenvectors in V as the principal components V' of data set X. We use a new dataset Y to represent X in V', which is calculated as $Y = XV'$. In particular, the variances of the data in first several eigenvectors carry much more information, which greatly reflects the query performance. We can only use the first several eigenvectors of the data distribution as the projection vectors in LSH and construct the hash tables, resulting in reducing space overhead.

3 DESIGN AND IMPLEMENTATIONS

In this section, in order to address the challenges of time and space inefficiencies in LSH-based schemes, we present a cost-effective hashing scheme, i.e., Distribution-aware LSH (DLSH). In practice, we utilize the principal components of the data distribution to allocate the projection vectors in LSH, which meets the needs of approximate queries with a few hash tables and is thus space saving. In order to enhance the precision of ANN query and decrease the overhead of distance computation, we quantify the weight of k hash functions and adjust the interval value in each hash table, and then record the hit frequency of candidates to decrease the size of ANN query results without any loss of accuracy.

First of all, we summarize the main notations used in Table 1.

Table 1: Variables and descriptions.

Notation	Description
c	approximation ratio
S	the set of n points
$D()$	the distance between two points
k	the number of functions in each concatenation
l	the number of hash tables
$C(q)$	query result set of q
$C'(q)$	refined result set of q
$Col(p)$	collision number of p
m	collision threshold
α	collision threshold in percentage
β	$false\ positive$ probability
δ	error probability

3.1 Distribution-aware Locality Sensitive Hashing

In the conventional LSH, the parameters of hash functions are determined through random-chosen algorithms that overlook the data distribution [24, 30]. In order to guarantee the accuracy, we have to use numerous hash tables and corresponding hash computation, thus causing substantial space and time overheads.

Our idea is to first utilize the principal components of the data distribution to replace the random projection vectors in LSH. In practice, we use the principal components of the data distribution to guide the hash selection. PCA decreases the dimensionality of data, and keeps the maximum standard deviation of projection vectors in datasets. Moreover, the computation overhead of PCA is proportional to the size of the dataset. We hence choose a feature dataset from the original dataset to guarantee time efficiency. Our experiments show that when the size of the feature dataset is larger than 5000, the obtained eigenvectors V has little impact on projection of the original dataset. This is also confirmed by existing observations [51]. However, the simple use of PCA fails to improve the overall performance due to orthogonality constraints, which introduces too many irrelevant points into queried result set. We hence put forward a refinement method to meet the needs of space and time efficiencies. The proposed refinement method consists of three-step operations, i.e., weight quantization, interval adjustment and frequency recordation. First, for constructing the distribution-aware LSH index, we randomly choose a feature dataset to represent the original dataset to reduce the computation overhead due to the use of offline PCA computation. Second, we obtain the eigenvalues and eigenvectors based on the covariance matrix of dataset. Finally, we quantify the weight of hash functions and adjust the interval value in each hash table to construct hash tables, and then compute hash values of items to insert. Algorithm 1 illustrates the construction algorithm of Distribution-aware LSH structure.

Algorithm 1 The DLSH construction algorithm

Input: Dataset X
1. Randomly choose a feature dataset X'.
2. Compute the means of variables based on Equation 4.
3. Obtain the covariance matrix based on Equation 6.
4. Get the eigenvalues N and the eigenvectors V, and choose the first $t(= kl)$ eigenvectors $V' = [v_1, v_2, \ldots, v_t]$.
5. The l hash tables are constructed as follows:
5.1 Compute the weight of k hash functions for each hash table based on the corresponding eigenvectors.
5.2 Adjust ω in LSH of i hash tables according to the LSH in $i-1$ hash tables $(1 < i \leq l)$.
5.3 Compute hash values Hash(x) for each $x \in X$:
5.4 Insert x to a bucket according to Hash(x).
Output: Distribution-aware LSH Index

3.2 The Refinement Method

3.2.1 Weight Quantization. Theoretical analysis proves the efficiency and efficacy of LSH for hashing close points into the same bucket [15]. However, due to the use of too many hash tables, the conventional LSH consumes high space overhead, which is much larger than the limited-size fast memory. Frequent access to the low-speed storage devices (e.g., hard disks) suffers from the slow query performance. The conventional random projection based LSH requires a random weight value for each hash function, and then computes the weighted sum as the hash value of each point in one hash table. Referring to Equation 2, for $\forall p \in S$, the hash value in one hash table is:

$$g_i(p) = a_{i1} * h_{i1}(p) + a_{i2} * h_{i2}(p) + \ldots + a_{ik} * h_{ik}(p), \quad (7)$$

where the weight a_{ij} is a constant randomly chosen from the range $[0, 1)$ following a p-stable distribution, and $1 \leq i \leq l$.

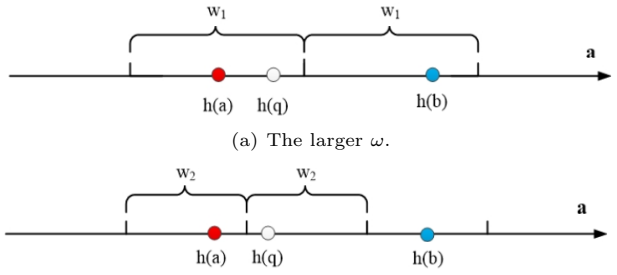
In Figure 1, we know that the projection vector in the Left is better than that of the Right, because the former differentiates the aggregated data, and meets the needs of approximate queries, namely, projecting close points into same bucket with a high probability and hashing distant points into same bucket with a low probability. Intuitively, a larger weight value should be allocated for the hash function with the better projection vector, for obtaining better query performance. There is no guarantee for query performance with random allocation of weight value in the conventional LSH.

Hence in Distribution-aware LSH, we utilize several eigenvectors as projection vectors, and so the corresponding eigenvalues can demonstrate the contribution of each direction better. Suppose that a dataset S and $N = [n_1, n_2, \ldots, n_k]$ is the set of k eigenvalues, then the weight of each hash function is:

$$a_i = \frac{n_i}{\sum_{i=1}^{k} n_i} (1 \leq i \leq k). \tag{8}$$

Hence in each hash table, the hash value of Point p is $g(p) = \sum_{i=1}^{k} a_i h_i(p)$ for each $p \in S$.

3.2.2 Interval adjustment. The interval value ω in LSH is a constant which is usually overlooked in real-world cloud computing applications. However, our experiments demonstrate that the overall performance of query systems is significantly different when using different values of ω. The parameter ω exhibits the granularity of hash collisions. A large value of ω increases the collision probability between similar points, but may hash distant points into the same bucket, which affects the query accuracy, as shown in Figure 3(a). Meanwhile, a small value of ω decreases the collision probability between far points, but possibly causes similar points to be hashed into different buckets as shown in Figure 3(b), thus exacerbating the query recall, although more hash tables are used.



(a) The larger ω.

(b) The smaller ω.

Figure 3: The example of interval difference.

In our DLSH design, we adjust the value of interval ω to improve the query performance. In each hash table, we utilize k hash functions, and the values of ω are identical, which are smaller than that in the last hash table. The projection vectors in DLSH are selected from eigenvectors V in sequence. The first principal component (eigenvector) has the largest possible variance, and others decrease in order. Principal components with the larger variance have the property that hash distant points into different buckets with a larger probability. In order to keep the property in principal components with smaller variance, we need to adjust and decrease the values of ω. In practice, the value of ω in each hash table is half of that in last hash table. Namely, for the ith $(i = 1, 2, \ldots, l)$ hash table, the value of ω is:

$$\omega_i = \frac{\omega_0}{2^i}, \tag{9}$$

where ω_0 is the initial default value. We manually set $\omega_0 = 4.0$ following the principle in E2LSH [9] and LSH-Forest [38].

3.2.3 Frequency Recordation. In approximate query applications, two similar points collide in all hash tables with very small probability. When the hash values of a query q are $g_i(q)(1 \leq i \leq l)$, we collect the candidate results containing at least one hash value $g_i(q)$. Hence, the result set for query q becomes the union of $B_{ig_i(q)}$:

$$C(q) = \cup_{i=1}^{l} B_{ig_i(q)}. \tag{10}$$

The result set $C(q)$ contains all candidates for the query q, and almost all near neighbors are hit with large probability. However, $C(q)$ includes too many irrelevant points that collide few times with Point q, which incurs high overhead of distance computation and the decrease of query precision.

In order to further reduce the size of the queried result set, we record the collision number for each candidate in $C(q)$ when performing the hash computation, i.e., how many times each candidate point collide with the query q in l hash tables. The *collision number* for $\forall p \in C(q)$ is:

$$Col(p) = \sum_{i=1}^{l} \neg(g_i(p) \bigoplus g_i(q)). \tag{11}$$

Intuitively, the probability that one is an ANN of the query q increases with its collision number. Namely, the point with greater collision number is the near neighbor of query q with a larger probability, while the point with smaller collision number is the near neighbor of query q with a smaller probability. In practice, the hash functions in the first hash table utilize the first few principal components with larger variance. The candidates with hash value $g_1(q)$ are approximate nearest neighbors of query q with larger possibility. These candidates are still maintained in $C'(q)$, which is the refined result set of query q. In other hash tables, when the collision numbers of candidates reach a threshold m, corresponding candidates are then added into $C'(q)$. If $Col(p) \geq m$, the point p is called to be $frequent$. Let α be the collision threshold in percentage, so $m = \alpha l$. During the query process, if there is a $frequent$ point whose distance to q is less than or equal to cR, we return TURE and the point; Otherwise, we return FALSE.

Lemma 3.1. *Let β and δ be the false positive and error probability, respectively. For $P_2 < \alpha < P_1$, $0 < \beta < 1$ and $0 < \delta < 1$, the number of hash tables l can be defined as [14]:*

$$l = \lceil \max(\frac{1}{2(p_1 - \alpha)^2} \ln \frac{1}{\delta}, \frac{1}{2(\alpha - p_2)^2} \ln \frac{2}{\beta}) \rceil. \tag{12}$$

3.3 Weighted Recall Evaluation

In order to take into account the contribution of each queried ANN for query performance, we propose a new metric for ANN query, called weighted recall. The weighted recall is defined as the fraction of relevant weighted instances that are retrieved. In an ANN query of Point q, we prefer to find nearer points, rather than farther ones, based on approximate degree. Nearer points to the query are more important than farther ones in ANN query systems. Moreover, the nearer points significantly contribute to the query performance, especially for the recall ratio. Hence, we describe relevant

instances of the query point with different weights. In our design, we explore and exploit the Euclidean distances between candidates and the query point to respectively set the weights of candidates. Based on the distances of candidates in $C'(q)$ from the query q, we allocate larger weights to nearer points. Suppose that a point $x_i \in C'(q)$, the corresponding weight is:

$$w_i = \frac{a}{D(q, x_i)^2}. \tag{13}$$

In fact, the candidate x_i may match the query q, in which the $D(q, x_i)^2$ is equal to 0, resulting in the infinite weight. In order to address this problem, we add an offset to the distance between each candidate and the query point, i.e.,

$$w_i = \frac{a}{D(q, x_i)^2 + b}, \tag{14}$$

where a and b are two constants.

Suppose that $X : \{x_1, x_2, \ldots, x_n\}$ is the exact-matching set, and $Y : \{y_1, y_2, \ldots, y_m\} \subset X$ is the ANN set. The weighted recall ratio is the sum of ANN weights divided by that of exact-matching weights, namely:

$$recall = \frac{\sum_{i=1}^{m} w_i}{\sum_{i=1}^{n} w_i}. \tag{15}$$

3.4 Parameter Settings

According to the Equation 12, the accuracy of DLSH is controlled by the error probability δ and the false positive percentage β, which are constants set by users. Specially, δ determines the successful rate of all LSH-based methods for ANN query. In this paper, we set $\delta = \frac{1}{e}$ following the principle in QALSH [14]. Intuitively, DLSH checks more frequent items with a bigger β, and hence achieves a better search quality but higher costs in terms of distance computation. Like [10, 14], we set $\beta = 100/n$ to restrict computation overhead, where $n = |S|$.

We now consider the number of hash tables l, collision threshold percentage α and collision threshold m. Referring to Equation 12 of Lemma 3.1, let $l_1 = \lceil \frac{1}{2(p_1-\alpha)^2} \ln \frac{1}{\delta} \rceil$, and $l_2 = \lceil \frac{1}{2(\alpha-p_2)^2} \ln \frac{2}{\beta} \rceil$, then l is equal to the larger one of l_1 and l_2, namely, $l = \max(l_1, l_2)$. Since $p_2 < \alpha < p_1$, l_1 increases monotonically with α and l_2 decreases monotonically with α. In practice, l should be as small as possible for space saving. Hence, we have the smallest l when $l_1 = l_2$. Then, α can be determined by:

$$\alpha = \frac{\mu \cdot p_1 + p_2}{1 + \mu}, \ where \ \mu = \sqrt{\frac{\ln \frac{2}{\beta}}{\ln \frac{1}{\delta}}}. \tag{16}$$

Replacing α in l_1 by Equation 16, we have:

$$l = \lceil \frac{(\sqrt{\ln \frac{2}{\beta}} + \sqrt{\ln \frac{1}{\delta}})^2}{2(p_1 - p_2)^2} \rceil. \tag{17}$$

After having the values of l and α, we set the collision threshold m as follows:

$$m = \lceil \alpha l \rceil. \tag{18}$$

3.5 Practical Operations

We describe practical operations of DLSH to support item insertion, ANN query and item deletion.

3.5.1 Insertion. The insertion operation needs to place items in hashed buckets for ANN query. Algorithm 2 illustrates the insertion algorithm for item x. We denote $B[*]$ to be the item in the bucket. In each hash table, we compute k hash values for item x, and then obtain the final hash value according to the weights of hash functions. Hence, the item x is inserted into the list of the bucket. We repeat the above operation until item x is inserted into all hash tables.

Algorithm 2 Insert(Item x)

```
1: i := 1
2: j := 1
3: while i ≤ l do
4:    while j ≤ k do
5:       h_j(x) = ⌊(a'·x+b)/ω_i⌋
6:    end while/* a' is the corresponding eigenvector of each
      hash function*/
7:    g_i(x) = Σ_{j=1}^{k} a_j h_j(x)
8:    x → B[g_i(x)]
9: end while/* a_j is the corresponding weight value of each
   hash function */
10: Return
```

3.5.2 ANN Query. The ANN query operation needs to traverse all hashed buckets of the item to be queried and obtain approximate nearest neighbors. Algorithm 3 shows the details of query operation. We first lookup the hashed bucket in the first hash table, and store the list into a result set. From the second hash table, we record the hit times of candidates in all hashed buckets of query item x. If the number of hit times is larger than a pre-defined $Threshold$, we store that item in the result set. We need to execute exact-matching operation on the result set of a query point x to eliminate irrelevant points and obtain the approximate nearest neighbors.

3.5.3 Deletion. In the deletion operation, we need to lookup the item to be deleted and then remove it from the bucket of all hash tables. Algorithm 4 illustrates the algorithm of deleting an item x from DLSH. We first lookup the item to be deleted in hash tables according to its hash values, and then delete it. The deletion operation is to remove an existing item in hash tables.

4 PERFORMANCE EVALUATION

In this section, we evaluate the performance of the proposed DLSH scheme by implementing a prototype in a large-scale cloud computing system.

4.1 Experimental Setup

The server used in our experiments runs on the Linux 2.6.18 environment and is equipped with an Intel 2.8GHz 16-core

Algorithm 3 ANN_Query(Item x)

```
1: Result = φ
2: counts ← {}
3: i := 1
4: Result := B[g_1(x)]
5: i++
6: while i ≤ l do
7:    for each item x ∈ B[g_i(x)] do
8:       if x ∈ items(counts) then
9:          Col[x] ← Col[x] + 1
10:      else
11:         Col[x] ← 1
12:      end if
13:   end for
14: end while
15: for each item x ∈ items(counts) do
16:   if Col[x] ≥ Threshold then
17:      Result ← Result ∪ {x}
18:   end if
19: end for
20: Return Result
```

Algorithm 4 Delete(Item x)

```
1: i := 1
2: while i ≤ l do
3:    if (B[g_i(x)] == x) then
4:       Delete x from B[g_i(x)]
5:    end if
6:    i++
7: end while
8: Return
```

CPU, a 16GB DRAM and 500GB hard disk. We implemented all functional components of DLSH scheme in the user space.

We examine the performance of our proposed DLSH by using three widely used datasets, which are distributed non-uniformly [45, 51]. The rationale comes from high dimensions and acceptable approximate results in these datasets for evaluation.

- **Object Recognition Benchmark Images**. The acknowledged object recognition benchmark images of University of Kentucky [28] consists of $10,200$ images in total. In this image set, 4 images compose a group. We utilize SIFT [26] to obtain 128-dimensional local feature descriptors of these images. A subset of $1k$ data points is sampled as the query set.
- **LabelMe Dataset**. We utilize $20K$ 512-dimensional Gist features extracted from the images of LabelMe dataset in our experiments. The first $1k$ features are used for querying. This dataset has been widely used as a benchmark dataset for evaluating object recognition methods [43].
- **Random Image Dataset**. 1 million 128-dimensional SIFT descriptors extracted from random images [23] are used in our experiments. We randomly choose $1k$ points from the dataset as the testing queries.

The evaluation metrics include the weighted recall ratio, precision ratio and F-Measure of ANN query operation, as well as mean time overhead and space overhead of the whole system. Specifically, the weighted recall ratio is defined as the fraction of the number of retrieved positive weighted instances to the total number of weighted instances which are positive in the dataset in ANN query. The precision ratio is defined as the fraction of the retrieved positive instances to the total number retrieved after completing lookup operations. The F-Measure is the combination of recall and precision, which is better to reflect the overall query performance among several schemes. The mean time overhead is the mean execution times per $1k$ ANN query requests in all schemes. The above metrics have been widely used in evaluating high-dimensional and approximate query results [45].

The performance of DLSH is associated with its parameter settings, especially the metric R that manages the measure of approximate nearest neighbors. Because of the uncertainties and probabilistic properties of LSH [15, 24], it is a severe task to identify an optimal R value for an ANN query. Hence, we utilize the popular and well-recognized sampling mechanism to obtain appropriate R values for our experiments, which was proposed in the conventional LSH study [9] and has been widely used in practical applications [5, 37]. We determine the suitable R values of approximate distances for three datasets to be 300, 400, 500 and 600 to appropriately and quantitatively represent the correlation.

We show advantages of DLSH scheme over conventional E2LSH [4, 9] as the baseline, and PCH [25], which is the combination of PCA and LSH. E2LSH randomly chooses a sample query set and a sample dataset from the testing dataset to compute the optimal parameters, i.e., the number of hash tables l and hash functions k in each hash table, based on the minimum processing time. After the computation based on three used datasets, the optimal case is 378 hash tables with 4 hash functions respectively. According to Equation 17, the appropriate value of hash tables in DLSH is $l = 5$. In the following figures, the pairs of number behind E2LSH, PCH, and DLSH are the numbers of hash functions in each table and the numbers of hash tables of the structure respectively. The capacity of each table is uniform. For example, $PCH(4, 5)$ indicates that the structure contains 5 hash tables, each of which has 4 hash functions.

4.2 Experimental Results

4.2.1 Weighted Recall Ratio. In an ANN query, the relevant items differ in their distances to the query point. Hence we take advantage of the weighted recall to evaluate the query performance, which we introduced in Section 3.3. Figure 4 illustrates the weighted recall ratio of query operations by using the dataset of benchmark images. We observe that the average weighted recall ratio of DLSH is 95%, which is slightly lower than the percentage of 98% in the conventional E2LSH when consuming 378 hash tables. Compared with PCH, our proposed DLSH obtains on average 5% recall ratio

degradations. E2LSH projects points with random projection vectors that are unaware of data distributions. Multiple hash tables have to be constructed to find as many similar points as possible, which is space-inefficient. PCH leverages principal components of data distributions to improve performance on E2LSH. Unlike it, DLSH further refines the queried result set based on the hit frequency to significantly decrease the time overhead of distance computation, with a little acceptable performance degradation.

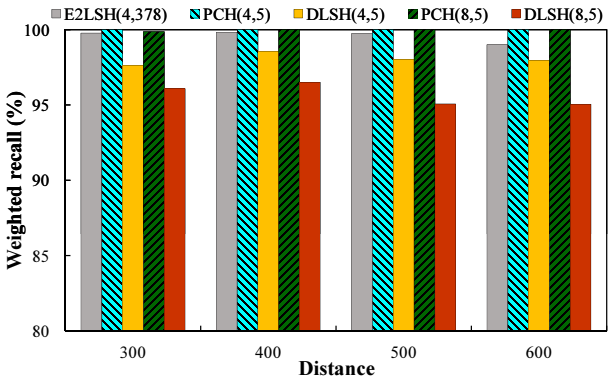


Figure 4: The weighted recall ratio of DLSH using the dataset of benchmark images.

Figure 5 shows the weighted recall ratio when using the LabelMe dataset. We observe that compared with E2LSH and PCH, our proposed DLSH only suffers on average 4% recall performance losses, while the average recall ratio of DLSH is 95% in LabelMe dataset, and 99% in E2LSH and PCH.

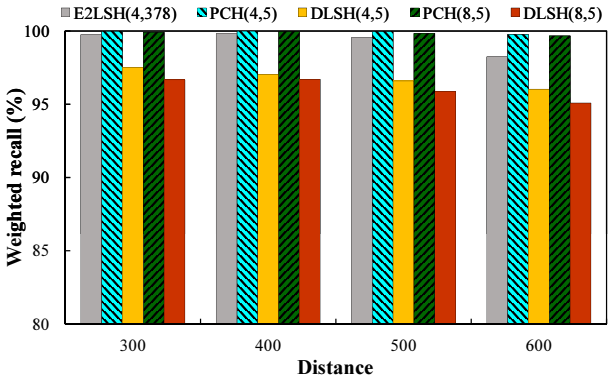


Figure 5: The weighted recall ratio of DLSH using the LabelMe dataset.

We examine the weighted recall ratio with the random image dataset as shown in Figure 6. Compared with E2LSH and PCH, our DLSH only decreases almost 2% and 4% recall ratio respectively, while the average weighted recall ratio of DLSH is 95% in random image dataset, and 97% in E2LSH as well as 99% in PCH.

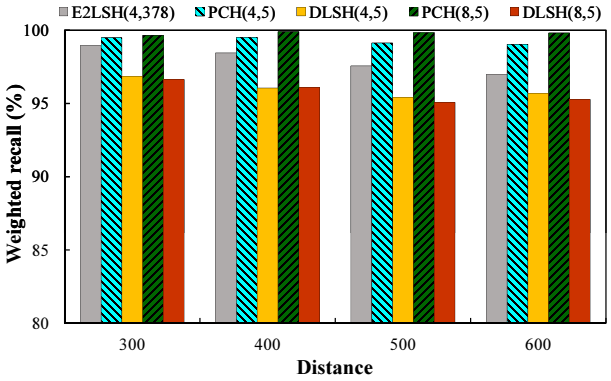


Figure 6: The weighted recall ratio of DLSH using the random image dataset.

4.2.2 Precision Ratio. We examine the precision ratio of E2LSH, PCH and DLSH by using the dataset of benchmark images as shown in Figure 7. Compared with E2LSH and PCH, our proposed DLSH significantly increases over 30% and 20% precision ratio when $k = 4$. DLSH takes into account of the principal components of data distributions, which project similar points into the same bucket with higher possibility than random projection vectors, as well as project distant points into different buckets. In order to handle the orthogonality constraints of PCA and reduce the queried result set, DLSH further adjusts the interval value in each hash table, takes advantage of the hit frequency of candidates to significantly decrease the number of irrelevant points, thus significantly improving the precision ratio.

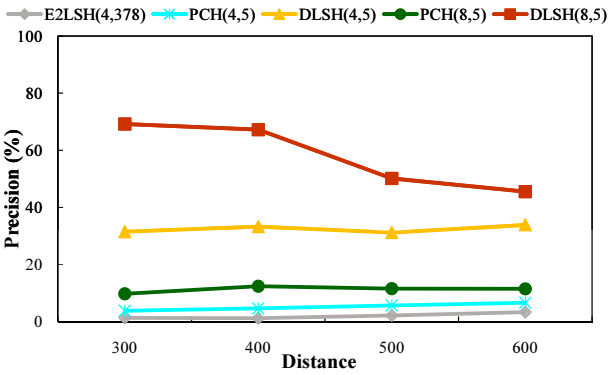


Figure 7: The precision ratio of DLSH using the dataset of benchmark images.

Figure 8 illustrates the precision ratio of query operations by using the LabelMe dataset. We observe that compared with E2LSH and PCH, our designed DLSH improves the precision over 35% and 25% when $k = 4$, respectively.

We examine the precision ratio with the random image dataset as shown in Figure 9. Compared with E2LSH and PCH, our designed DLSH obtains on average over 50% and 30% precision improvements.

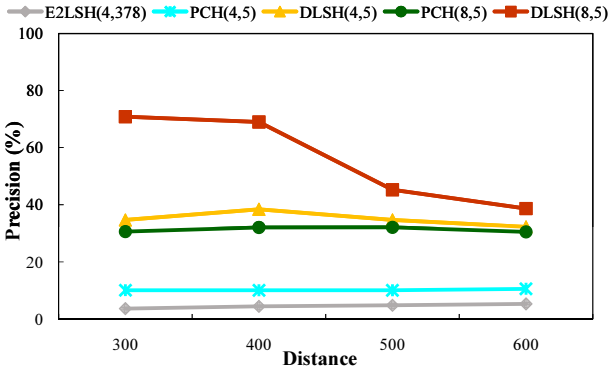


Figure 8: The precision of DLSH using the LabelMe dataset.

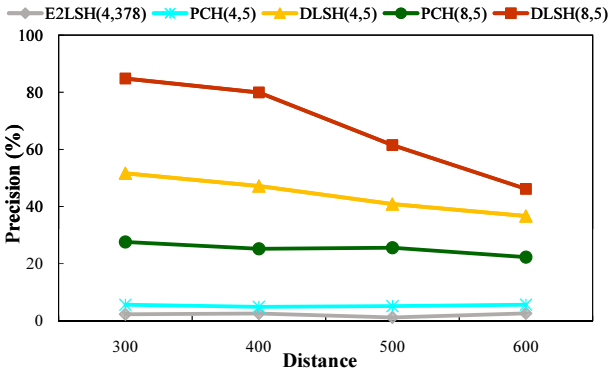


Figure 9: The precision of DLSH using the random image dataset.

4.2.3 F-Measure. The F-Measure is the combination of recall ratio and precision ratio, which comprehensively reflects the overall query performance of each scheme. It is defined as:

$$F - Measure = \frac{2 \times recall \times precision}{recall + precision}. \quad (19)$$

Figure 10 shows the F-Measure by using the dataset of benchmark images. We observe that DLSH improves the F-Measure more than 40% compared with E2LSH, as well as 35% when $k = 4$ and 60% when $k = 8$ compared with PCH. For the LabelMe dataset, compared with E2LSH and PCH, our proposed DLSH obtains on average 40% and 25% improvements when $k = 4$ respectively as shown in Figure 11.

Figure 12 illustrates the F-Measure with the random image dataset. We observe that DLSH obtains on average over 60% improvement compared with E2LSH, as well as 50% improvement when $k = 4$ and 40% when $k = 8$ compared with PCH. Hence, DLSH significantly improves the overall query performance than E2LSH and PCH.

4.2.4 Mean Insertion Time Overhead. Table 2 illustrates the mean times per 1k insertion operations of DLSH by using three datasets. DLSH and PCH achieve almost the same

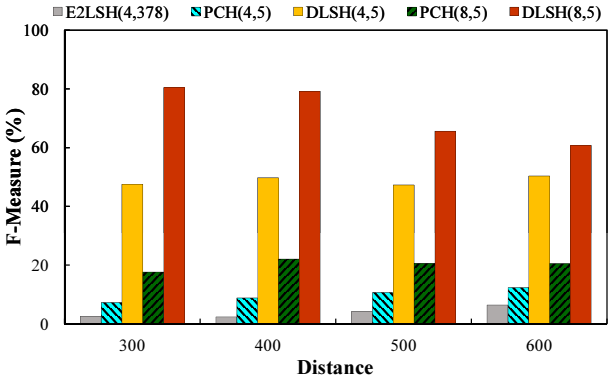


Figure 10: The F-Measure of DLSH using the dataset of benchmark images.

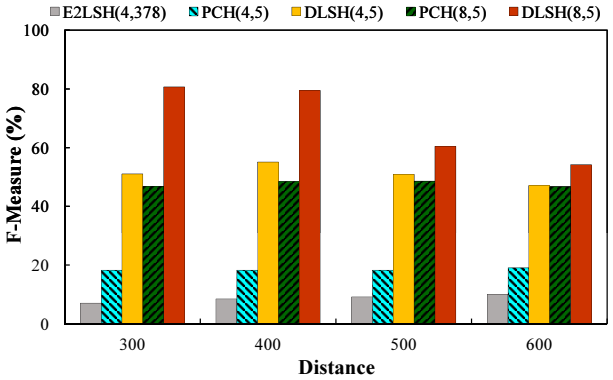


Figure 11: The F-Measure of DLSH using the LabelMe dataset.

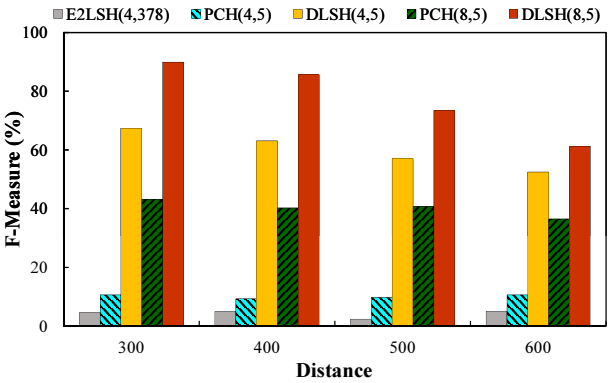


Figure 12: The F-Measure of DLSH using the random image dataset.

performance due to similar implementations of insertion operation. With the increase of distance threshold, DLSH consumes on average 18.5ms to perform 1k insertion operations. We observe that the insertion performance is tightly correlated with the hash computation efficiency, which has been significantly improved in our scheme. Hence, we obtain the

improvement upon insertion performance due to the reduction in the number of hash tables, which significantly reduces the overhead of hash computation.

Table 2: The mean times(ms) per 1k insertions.

Datasets	R=300	R=400	R=500	R=600
Benchmark Images	18.02	18.02	18.02	19.82
LabelMe	18.93	18.25	19.61	18.25
Random Image	18.86	18.19	18.19	18.86

4.2.5 Mean Query Time Overhead. We examine the mean time overheads per $1k$ ANN queries of E2LSH, PCH and our DLSH by using the dataset of benchmark images as shown in Figure 13. Compared with E2LSH and PCH, DLSH significantly reduces almost 80% and 50% time overheads. E2LSH hashes points with random projection vectors, which adds extra irrelevant points hashed into the same bucket. A large size of the result set is generated for distance computation, which is time consuming. PCH introduced principal components of the data distribution reduces the size of the ANN query result set, and saves much computation time. Based on PCH, our proposed DLSH handles the orthogonality constraints of PCA and further decreases the size of the queried result set based on hit frequency, result in obtaining time saving.

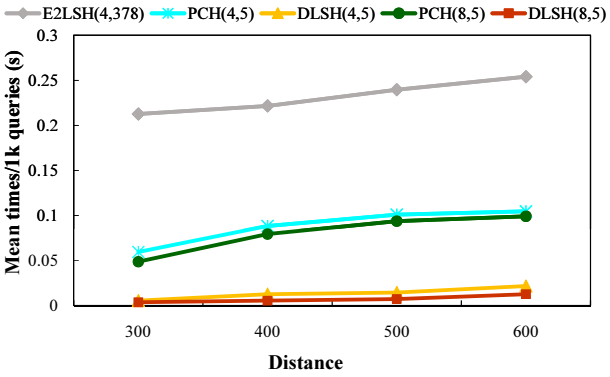


Figure 13: The mean times per group having 1000 queries of DLSH using the dataset of benchmark images.

Figure 14 illustrates the mean time overheads per group having $1k$ ANN queries of three schemes by using the LabelMe dataset. We observe that DLSH decreases over 85% time overhead compared with E2LSH, and almost 75% compared with PCH.

Figure 15 shows the mean time overheads in the random image dataset. Compared with E2LSH and PCH, our proposed DLSH decreases on average almost 90% and 80% time overheads. Hence, our DLSH optimizes the cloud computing system performance by decreasing the time overhead.

Both DLSH and PCH need to carry out the two-step operations, i.e., hash computation and distance computation. PCA has been used to generate the projection vectors before

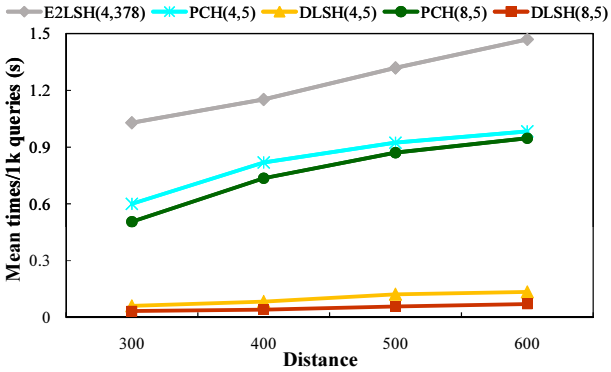


Figure 14: The mean times per group having 1000 queries of DLSH using the LabelMe dataset.

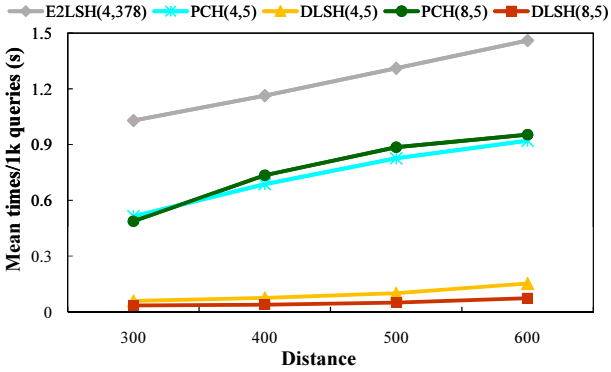


Figure 15: The mean times per group having 1000 queries of DLSH using the random image dataset.

hash computation. Unlike PCH, DLSH uses the proposed refinement method to decrease the data amount to be computed within the hash computation without extra overheads. This has been implemented via changing the computation parameters, i.e., the weight and the interval value ω of each LSH function. Table 3 illustrates the reduction ratio of data amount in distance computation. Compared with PCH, DLSH decreases on average 90% data amount due to refinement.

Table 3: Reduction ratio(%) of distance computation.

Datasets	R=300	R=400	R=500	R=600
Benchmark Images	94.85	90.75	89.66	83.05
LabelMe	94.75	93.41	90.02	89.45
Random Image	94.32	93.49	91.69	86.90

4.2.6 Space Overhead. We compare the space overheads of three schemes by examining the size of hash tables. The capacity of each table is uniform, and we hence simply compare the number of hash tables each scheme uses, namely, the second number behind each scheme in above figures, to allow storage overhead to be comparable. We observe that in order to guarantee query accuracy, E2LSH has to utilize

multiple hash tables, such as 378 hash tables in our experiments, which is obtained by the optimal algorithm. However, DLSH and PCH obtain more space savings by only using 5 hash tables, which significantly decrease 98% space overhead. The space sizes of the schemes in our evaluation is shown in Table 4. Even if DLSH and PCH spend the same space overhead, DLSH gains better query performance and time efficiency. DLSH makes space efficiency to support real-time queries better.

Table 4: Space overheads normalized to $DLSH$.

	E2LSH	PCH	DLSH
The value of l	378	5	5
Normalization	75.6	1	1

4.3 Experimental Summary

Experimental results demonstrate DLSH has the advantages in terms of precision and F-Measure of queries, as well as time and space overheads, with some recall performance reduction. It is an efficient tradeoff between recall reduction and space and query latency improvements. Existing schemes, such as E2LSH and PCH, can also use fewer hash tables to improve the space efficiency and query latency. The key is how to efficiently determine the collision threshold. Our contribution is to leverage the refinement scheme to obtain the optimized tradeoff in terms of recall and precision. DLSH can efficiently improve the query performance and decrease time and space overheads to optimize the cloud computing system performance.

5 RELATED WORK

Cost-effective query services are needed in cloud computing. Clarinet [41] allows a query plan to be WAN-aware for optimizing query completion times for geo-distributed data analytics. Cedar [18] proposes wait-time duration selection for aggregators in clusters, to gain near-optimal improvements in response qualities. In order to guarantee the data confidentiality and query privacy in public clouds, RASP [44] combines several techniques, order preserving encryption, dimensionality expansion, random noise injection and random project, to randomly transform the multidimensional datasets, to provide efficient query services.

Multiple hashing-based query schemes in cloud computing have been proposed to improve system performance. EIRQ [22] leverages an aggregation and distribution layer to enable users of different ranks to retrieve different percentages of files that match their queries, which reduces querying costs incurred in the cloud. Moreover, NEST [13] utilizes cuckoo-driven locality-sensitive hashing to obtain load-balancing hash tables and support approximate queries. Based on Bloom filters, a new data structure, called Bloom Tree, is proposed to support multiple-set membership testing, which achieves space compactness and operates more efficiently than the previous work [47]. In order to investigate the efficient search result verification problem in large dynamic encrypted cloud data, an efficient verifiable conjunctive keyword search scheme is presented to allow file update with verification cost related to search operation rather than the file collection size [35]. The proposed lock-free cuckoo hashing algorithm [27] optimizes the synchronization between query and modification operations to offer high query throughput.

The performance of hashing-based schemes is significantly dependent on the quality of the hash functions they use. Many researchers have paid attention to improving the performance of hashing-based schemes by using data-dependent hash functions. PCH [25] exploits the properties of the distribution of stored data and projects data to principal axes. Complementary hashing [45] utilizes multiple complementary hash tables which are learned sequentially in a boosting manner, thus balancing the precision and recall more effectively. IsoHash [16] learns projection functions with equal variances for different dimensions, which is the first work to verify the effectiveness by theory and experiments. Based on boosted decision trees and GraphCut, an efficient supervised hashing method is proposed to make training and evaluation fast for learning hash functions, which shows the advantages on retrieval performance and fast training for high-dimensional data [20]. By introducing an auxiliary variable, SDH [33] handles the discrete constrains imposed on the pursued hash codes, and then generates the optimal binary hash codes for linear classification.

Existing work motivates our design of DLSH that makes further improvements upon them. DLSH utilizes principal components of data distributions as the projection vectors of LSH to map similar items into the same bucket in hash tables, thus supporting ANN query and obtaining time and space saving without the loss of query performance.

6 CONCLUSION

In this paper, we propose a cost-efficient hashing scheme, called DLSH, to support approximate nearest neighbor query for large-scale cloud computing applications. The proposed DLSH has the contributions to three main challenges in LSH-based schemes, i.e., low accuracy, space inefficiency and high query latency. DLSH takes advantage of principal components of the data distribution as the projection directions of hash functions in LSH, and handles the orthogonality constraints of PCA by quantifying the weight of hash functions, adjusting the interval value in each hash table and reducing the size of queried results based on hit frequency. Compared with state-of-the-art work, extensive experimental results demonstrate the efficiency and efficacy of DLSH. We have released the source code of DLSH for public use in Github at *https://github.com/syy804123097/DLSH*.

ACKNOWLEDGMENTS

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202 and State Key Laboratory of Computer Architecture under Grant CARCH201505. The authors are grateful to anonymous reviewers and our shepherd, Henggang Cui, for their constructive feedbacks and suggestions.

REFERENCES

[1] December 2015. How many photos are uploaded to Flickr every day, month, year? *https://www.flickr.com/photos/franckmichel/6855169886/in/photostream/* (December 2015).

[2] Updated July 2016. The Top 20 Valuable Facebook Statistics. *https://zephoria.com/top-15-valuable-facebook-statistics/* (Updated July 2016).

[3] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics* 2, 4 (2010), 433–459.

[4] Alexandr Andoni and Piotr Indyk. 2005. E2LSH 0.1 user manual. (2005).

[5] Alexandr Andoni and Piotr Indyk. 2006. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Sciemce (FOCS'06)*. IEEE, 459–468.

[6] Alexandr Andoni, Piotr Indyk, Huy L. Nguyen, and Ilya Razenshteyn. 2014. Beyond Locality-Sensitive Hashing. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1018–1028.

[7] Alexandr Andoni and Ilya Razenshteyn. 2015. Optimal Data-Dependent Hashing for Approximate Near Neighbors. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. ACM, 793–801.

[8] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. 2014. Privacy-Preserving Multi-Keyword Ranked Search over Encrypted Cloud Data. *IEEE Transactions on parallel and distributed systems* 25, 1 (2014), 222–233.

[9] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 253–262.

[10] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2012. Locality-Sensitive Hashing Scheme Based on Dynamic Collision Counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 541–552.

[11] Jinyang Gao, Hosagrahar Visvesvaraya Jagadish, Wei Lu, and Beng Chin Ooi. 2014. DSH: Data Sensitive Hashing for High-Dimensional k-NN Search. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 1127–1138.

[12] Yu Hua, Bin Xiao, Dan Feng, and Bo Yu. 2008. Bounded LSH for Similarity Search in Peer-to-Peer File Systems. In *Proceedings of the 37th International Conference on Parallel Processing*. IEEE, 644–651.

[13] Yu Hua, Bin Xiao, and Xue Liu. 2013. NEST: Locality-aware Approximate Query Service for Cloud Computing. In *Proceedings IEEE INFOCOM*. IEEE, 1303–1311.

[14] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-Aware Locality-Sensitive Hashing for Sproximate Nearest Neighbor Search. *Proceedings of the VLDB Endowment* 9, 1, 1–12.

[15] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 604–613.

[16] Weihao Kong and Wu-Jun Li. 2012. Isotropic Hashing. In *Advances in Neural Information Processing Systems*. 1646–1654.

[17] Simon Korman and Shai Avidan. 2016. Coherency Sensitive Hashing. *IEEE transactions on pattern analysis and machine intelligence* 38, 6 (2016), 1099–1112.

[18] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. 2016. Hold'em or Fold'em?: Aggregation Queries under Performance Variations. In *Proceedomgs pf tje Eleventh European Conference on Computer Systems (EuroSys'16)*. ACM.

[19] Ming Li, Shucheng Yu, Ning Cao, and Wenjing Lou. 2011. Authorized Private Keyword Search over Encrypted Data in Cloud Computing. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS'11)*. IEEE, 383–392.

[20] Guosheng Lin, Chunhua Shen, Qinfeng Shi, Anton van den Hengel, and David Suter. 2014. Fast Supervised Hashing with Decision Trees for High-Dimensional Data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1963–1970.

[21] Jia Liu, Bin Xiao, Kai Bu, and Lijun Chen. 2014. Efficient Distributed Query Processing in Large RFID-enabled Supply Chains. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'14)*. IEEE, 163–171.

[22] Qin Liu, Chiu C Tan, Jie Wu, and Guojun Wang. 2012. Efficient Information Retrieval for Ranked Queries in Cost-Effective Cloud Environments. In *Proceedings of the 31st Annual IEEE International Conference on Computer Communications*. IEEE, 2581–2585.

[23] David G Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision* 60, 2 (2004), 91–110.

[24] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment, 950–961.

[25] Yusuke Matsushita and Toshikazu Wada. 2009. Principal Component Hashing: An Accelerated Approximate Nearest Neighbor Search. *Pacific-Rim Symposium on Image and Video Technology* (2009), 374–385.

[26] Krystian Mikolajczyk, Tinne Tuytelaars, Cordelia Schmid, Andrew Zisserman, Jiri Matas, Frederik Schaffalitzky, Timor Kadir, and Luc Van Gool. 2005. A Comparison of Affine Region Detectors. *International Journal of Computer Vision* 65, 1-2 (2005), 43–72.

[27] Nhan Nguyen and Philippas Tsigas. 2014. Lock-Free Cuckoo Hashing. In *Proceedings of the 34th International Conference on Distributed Computing Systems*. IEEE, 627–636.

[28] David Nister and Henrik Stewenius. 2006. Scalable Recognition with a Vocabulary Tree. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Version and Pattern Recognition (CVPR'06)*, Vol. 2. IEEE, 2161–2168.

[29] Rina Panigrahy. 2006. Entropy based Nearest Neighbor Search in High Dimensions. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*. Society for Industrial and Applied Mathematics, 1186–1195.

[30] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. 2015. Neighbor-Sensitive Hashing. *Proceedings of the VLDB Endowment* 9, 3 (2015), 144–155.

[31] Sébastien Poullot, Olivier Buisson, and Michel Crucianu. 2007. Z-grid-based Probabilistic Retrieval for Scaling Up Content-Based Copy Detection. In *Proceedings of the 6th ACM International Conference on Image and Video Retrieval*. ACM, 348–355.

[32] Maxim Raginsky and Svetlana Lazebnik. 2009. Locality-Sensitive Binary Codes from Shift-Invariant Kernels. In *Advances in Neural Information Processing Systems*. 1509–1517.

[33] Fumin Shen, Chunhua Shen, Wei Liu, and Heng Tao Shen. 2015. Supervised Discrete Hashing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 37–45.

[34] Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska. 2015. Automating Model Search for Large Scale Machine Learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 368–380.

[35] Wenhai Sun, Xuefeng Liu, Wenjing Lou, Y Thomas Hou, and Hui Li. 2015. Catch You If You Lie to Me: Efficient Verifiable Conjunctive Keyword Search over Large Dynamic Encrypted Cloud Data. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'15)*. IEEE, 2110–2118.

[36] Yuzhe Tang and Ling Liu. 2015. Privacy-Preserving Multi-Keyword Search in Information Networks. *IEEE Transactions on Knowledge and Data Engineering* 27, 9 (2015), 2424–2437.

[37] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and Efficiency in High Dimensional Nearest Neighbor Search. In *Proceedings of the 2009 ACM SIGMOD Conference on Management of data*. ACM, 563–576.

[38] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2010. Efficient and Accurate Nearest Neighbor and Closest Pair Search in High-Dimensional Space. *ACM Transactions on Database Systems (TODS)* 35, 3 (2010), 20.

[39] Shixin Tian, Ying Cai, and Zhenbi Hu. 2016. A Parity-Based Data Outsourcing Model for Query Authentication and Correction. In *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS'16)*. IEEE, 395–404.

[40] Vernon Turner, John F Gantz, David Reinsel, and Stephen Minton. 2014. The digital universe of opportunities: rich data and the increasing value of the internet of things. *International*

Data Corporation, White Paper, IDC_1672 (2014).

[41] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. CLARINET: WAN-Aware Optimization for Analytics Queries. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 435–450.

[42] Dongsheng Wang, Xiaohua Jia, Cong Wang, Kan Yang, Shaojing Fu, and Ming Xu. 2015. Generalized Pattern Matching String Search on Encrypted Data in Cloud Systems. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'15)*. IEEE, 2101–2109.

[43] Yair Weiss, Antonio Torralba, and Rob Fergus. 2009. Spectral Hashing. In *Advances in Neural Information Processing Systems*. 1753–1760.

[44] Huiqi Xu, Shumin Guo, and Keke Chen. 2014. Building Confidential and Efficient Query Services in the Cloud with RASP Data Perturbation. *IEEE Transactions on Knowledge and Data Engineering* 26, 2 (2014), 322–335.

[45] Hao Xu, Jingdong Wang, Zhu Li, Gang Zeng, Shipeng Li, and Nenghai Yu. 2011. Complementary Hashing for Approximate Nearest Neighbor Search. In *Proceedings of the 2011 IEEE International Conference on Computer Vision*. IEEE, 1631–1638.

[46] Lei Xu, Hong Jiang, Lei Tian, and Ziling Huang. 2014. Propeller: A Scalable Real-Time File-Search Service in Distributed Systems. In *Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS'14)*. IEEE, 378–388.

[47] Myung Keun Yoon, JinWoo Son, and Seon-Ho Shin. 2014. Bloom Tree: A Search Tree Based on Bloom Filters for Multiple-Set Membership Testing. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'14)*. IEEE, 1429–1437.

[48] Felix X Yu, Sanjiv Kumar, Yunchao Gong, and Shih-Fu Chang. 2014. Circulant Binary Embedding. In *Proceedings of the International Conference on Machine Learning*, Vol. 6. 7.

[49] Deli Zhang and Damian Dechev. 2016. An Efficient Lock-Free Logarithmic Search Data Structure Based on Multi-dimensional List. In *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS'16)*. IEEE, 281–292.

[50] Lan Zhang, Taeho Jung, Cihang Liu, Xuan Ding, Xiang-Yang Li, and Yunhao Liu. 2015. POP: Privacy-Preserving Outsourced Photo Sharing and Searching for Mobile Devices. In *Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS'15)*. IEEE, 308–317.

[51] Wei Zhang, Ke Gao, Yong-dong Zhang, and Jin-tao Li. 2010. Data-Oriented Locality Sensitive Hashing. In *Proceedings of the 18th ACM international conference on Multimedia*. ACM, 1131–1134.