Salient Index for Similarity Search Over High Dimensional Vectors

SALIENT INDEX FOR SIMILARITY SEARCH OVER HIGH DIMENSIONAL VECTORS

BY

YANGDI LU, M.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Master of Science (2018) McMaster University

(Computing & Software) Hamilton, Ontario, Canada

TITLE:	Salient Index for Similarity Search Over High Dimensional Vectors
AUTHOR:	Yangdi Lu M.Sc., (Computing & Software) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Wenbo He
NUMBER OF PAGES:	vii, 64

Abstract

The approximate nearest neighbor(ANN) search over high dimensional data has become an unavoidable service for online applications. Fast and high-quality results of unknown queries are the largest challenge that most algorithms faced with. Locality Sensitive Hashing(LSH) is a well-known ANN search algorithm while suffers from inefficient index structure, poor accuracy in distributed scheme. The traditional index structures have most significant bits(MSB) problem, which is their indexing strategies have an implicit assumption that the bits from one direction in the hash value have higher priority. In this thesis, we propose a new content-based index called Random Draw Forest(RDF), which not only uses an adaptive tree structure by applying the dynamic length of compound hash functions to meet the different cardinality of data, but also applies the shuffling permutations to solve the MSB problem in the traditional LSH-based index. To raise the accuracy in the distributed scheme, we design a variable steps lookup strategy to search the Δ-step sub-indexes which are most likely to hold the mistakenly partitioned similar objects. By analyzing the index, we show that RDF has a higher probability to retrieve the similar objects compare to the original index structure. In the experiment, we first learn the performance of different hash functions, then we show the effect of parameters in RDF and the performance of RDF compared with other LSH-based methods to meet the ANN search.

Contents

List of Figures

Chapter 1

Introduction

Nowadays, shared video websites have become increasingly popular. For example, more than 3 billion searches are processed a month in Youtube. 136,000 photos are uploaded on Facebook every 60 seconds. 500 million tweets are sent per day. The volume, velocity, and variety of the unstructured data make it very challenging in processing the data in these online multimedia applications, wherein the content-based similarity search is an indispensable and fundamental operation. To reduce the complexity of tackling the large-scale multimedia data, a commonly used approach is manually tagging the video with metadata, then apply text-mining, Natural Language Processing (NLP), or text analytics methods for structured metadata to study the unstructured data. There are two drawbacks to this type of methods: (1) The performance of online multimedia systems are dependent on the quality and completeness of the metadata, however, having humans manually annotate videos/images is labour-intensive, time consuming and error prone; (2) the contemporary online video/image systems trust the users who upload the video clips to annotate the videos/images. However, a user may intentionally use deceptive descriptions to a video in order to

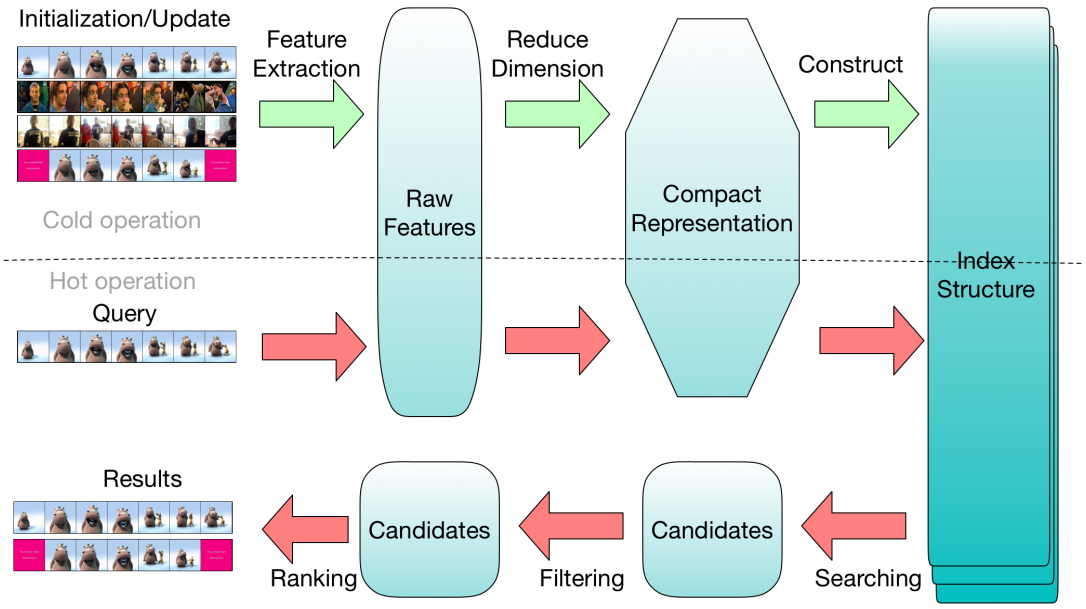

Figure 1.1: The overview of content-based video search system.

gain popularity. In this sense, the human-generated metadata can be one of the causes leading to inaccurate search results [13].

In this thesis, we design and implement a content-based indexing scheme to support efficient similarity search over large-scale unstructured data with high dimensionality. As shown in Figure 1.1, there are three stages to initialize the index:

1. **Feature extraction**: The feature descriptors are extracted directly from object data(e.g. images and videos) by using one of the state-of-the-art algorithms SIFT[18, 26] color histogram [15], or their combinations. The "similarity property" has been faithfully captured, which means the similarity between these feature descriptors(i.e. color distribution) can be used to evaluate the similarity between object data.

2. **Reduce dimension**: The feature descriptors are usually high-dimensional vectors with rich information. However, search directly over the high-dimensional data will lead to the Curse of Dimensionality [33], where query performance declines exponentially with the increasing number of dimensions. To break this curse, various algorithms of Approximate Nearest Neighbor (ANN) search [10, 9, 36] have been proposed, where feature descriptors are usually converted into compact representations (low dimension vectors).

3. **Construct index**: Based on the compact representations from above, we build an index to facilitate the search, where we reduce the complexity from $O(log(n))$ to $O(1)$.

Among ANNs locality sensitive hashing (LSH) [10, 2, 31, 17, 19, 29, 1, 37] is a sublinear algorithm to find ANN for high-dimensional data points by applying by applying the distance-preserving hash functions to project similar high-dimensional points into the same bucket. Ordering permutation indexing(OPI) [9, 22] is an efficient non-distributed algorithm to predict the similarity between objects according to how they rank their distance towards a distinguished set of pivots(anchor objects). The distance between the objects now is hinted by the distance between their respective permutations. Small world graphs(SWG) [36, 20] is a greedy searching algorithm in metric spaces. The basic concept resides on "The neighbor of my neighbor is also likely to be my neighbor", which is also called the *small world theory*. The traversing between any two nodes on a navigable small world network is of polylogarithmic time complexity, which makes it suitable for ANN search problem. While the restriction of SWG is that the queries of data points should be inserted in the SWG before searching the ANN.

To achieve the practical online similarity search system, besides the speed and quality of results need to be considered, two additional issues are also extremely important: 1) whether the algorithm can be distributed to tackle the incremental data, since the memory space is limited and expensive. 2) whether the system can deal with concurrent unknown queries not be inserted in the pre-constructed index. Due to the fact that OPI is not a distributed algorithm, which is not applicable to cope with the increasing data. The SWGs restriction makes the system update the index frequently, which is not a proper scheme in practical system view for the online system since updating the index is a resource-consumption operation. While the LSH fulfills these two conditions: meet the requirement that the search queries do not need to be in the index and requirement of parallel/distributed design in online similarity search system [2, 37]. Thus, we apply LSH to implement a practical content-based similarity search system.

Here, we present a summary of our contributions in this thesis:

1. We design a content-based index called Random Draw Forest(RDF), which uses multiple shuffling permutations to conquer the MSB problem. Each Random Draw Tree(RDT) provides a portion of similar objects for the query. As the number of objects increases in RDT, the dimension of hash values is adaptively extended in deeper level to increase the resolution for unbalanced data distribution, which significantly improves the accuracy of ANN queries.

2. We apply the distributed Layered-LSH scheme [2] to RDF, which reduces the search range and enables the parallelism/distribution of the system. The orthogonal hash family [12] is applied to partition the space more accurately. To deal with the mistakenly partitioned similar objects, a Δ-step sub-indexes

search strategy is designed to leverage the accuracy and efficiency.

3. We combine the Multi-probe LSH search strategy [19] and PHF multi-index storage [37] to reduce the index storage and implement RDF in the real key-value database. We implement the state-of-the-art LSH method to demonstrate the performance of different hash functions. Then we conduct experiments to tune the related parameters and compare with other traditional LSH methods on the video dataset. The evaluation result demonstrates RDF does have 5.4% to 8.3% improvement in terms of average accuracy(Section 5.5).

1.1 Thesis Outline

In Chapter 2, we introduce the related work, background and problem statement in this work. For example, the state-of-the-art ways to improve the LSH, the definitions of approximate nearest neighbor, locality sensitive hashing, batch orthogonal hash function, p-stable hash function. The main issues that LSH-based methods experience.

Chapter 3 first discusses the Layered Hash for reducing the dimensionality of original feature descriptors and dividing the large dataset into different content-sensitive partitions. Then we design a Δ-step lookup strategy to optimize the search process and balance the trade-off between the efficiency and accuracy. The experimental comparisons are shown in the last section of Chapter 3.

In Chapter 4, we first introduce a new index structure called Random Draw Forest(RDF), which consists of Random Draw Tree(RDT). We present the detail steps to build the RDT with pseudocode. We also design a multi-probes and Δ-step search

algorithm to search over the RDF to achieve better performance.

In Chapter 5, we implement the basic LSH(BLSH) index to compare the performance of four different hash family: P-Stable, J2JSD, SRP and BOA. Then we tune the related parameters in RDF and compare the RDF with other LSH-based methods.

In the last Chapter, we conclude the thesis and remark the advantages and disadvantages of RDF. We also propose some possible improvements, which can reduce the search range while maintaining a high accuracy.

Chapter 2

Preliminary

2.1 Related Work

Real-world objects are represented in usually vectorial form. We commonly refer to these object representations as data points in some space. To find the exact nearest neighbor for high-dimensional points, an improved binary tree structure KD-tree [4] is developed, where the points are partitioned into different level cells. However, the real-time query complexity of KD-tree is around $O(2^d log(n))$, where d is the dimension of points, n is the number of points. It spends $log(n)$ to find the cells "near" the query points, 2^d to search around cells in that neighborhood(which is typically treated as a constant). Therefore, the practical condition to use KD-tree is $n >> 2^d$. Otherwise, the efficiency is no better than brute-force linear scan approach(the query complexity is $O(dn)$), which is called the Curse of Dimensionality [33].

To pursue the speed, the searching condition is relaxed. Approximate Nearest Neighbor(ANN) methods are developed to pursue the *good-enough fast* answers by sacrificing a limited amount of accuracy. To solve the ANN problem in high

dimensional space, *Indyk* and *Motwani* proposed the idea of Locality Sensitive Hashing(LSH)[10] that has the property that projects close-by points into the same hash bucket in hash tables with a higher probability than distant ones. However, this random algorithm has the following aspects can be improved.

1. *Search strategy*: Panigraphy proposed an entropy-based LSH [23] that generates random "perturbed" objects near the query object, queries all of them plus the query object to generate candidate set. Based on the entropy-based LSH, Multi-probes LSH [19] proposed a better way to generate high quality "perturbed" query based on the hash values, which not only reduces the memory usage of index, but also achieve high accuracy.

2. *Hash functions*: P-stable hash functions [6] is designed for ℓ_p norm, where $p \in (0, 2]$. Sign random projection(SRP) hash functions [5] is designed for cosine(angular) distance. Orthogonal hash functions [12] improved the SRP where all hash functions are orthogonal to each other. Cross-polytope hash function [1] is another good choice for cosine distance. J2JSD hash function [21] is a hash family for probability distributions.

3. *Index structure*: LSB [31] transfers the hash values into Z-order values to build a B-tree structure index. SKLSH [17] defines a new distance measure of hash values, and sort these hash values to build a B^+-tree. Basically, these two methods tweak the data placement in the disk to accelerate query request processing. PHF [37] use a multi-index storage between SSD(Solid-State Drive) and memory to facilitate the query speed. LSH Forest [3] is a tree-based generation of LSH, where heavily load hash buckets are recursively partitioned with embedded hash tables.

4. *Distributed design*: DLSH [2] uses layered-LSH to partition the points into several partitions as distributed scheme. PLSH [30] broadcasts the query requests to all distributed/parallel workers in the system to achieve parallelism.

5. *Collision optimization*: C2LSH [7] optimizes the compound hash functions by dynamically computing the hash value one by one, rather than computing the compound hash functions at once.

More research works are still doing to improve the accuracy and efficiency of LSH algorithms.

2.2 Approximate Nearest Neighbor

After relaxing the condition, the high dimensional content-based similarity search problem is switched to approximate nearest neighbor(ANN) search problem. As shown in Definition 1, ANN search aims to find k nearest neighbor results for a set of queries.

Definition 1 ANN Search. *Given a set of data points D in d-dimensional space $\mathbb{R}^d$ and a set of query points Q. ANN search returns the k nearest points of D to each query $q \in Q$.*

Data points v_1 and v_2 are represented as vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ in d-dimensional vector space. If the $\| v_1, v_2 \|_s$ is smaller than pre-defined constant R, we say that they are ANN, where $\| \cdot, \cdot \|_s$ denotes the ℓ_s distance. Since it is easier for a user to pick k rather than a non-intuitive threshold R, so K-NN search is more attractive. The

typical similarity measure distance are the Euclidean distance($s = 2$) and the Hamming distance($s = 1$). Cosine(angular) distance is another popular one to measure similarity of two points.

2.3 Locality Sensitive Hashing

As shown in *Definition 2*, LSH has the properties that close objects in high-dimensional space will collide with a higher possibility than distant ones.

Definition 2 Locality Sensitive Hashing. *Given a distance R, a dataset D, an approximate ratio c and two probability values $\mathcal{P}_1$ and $\mathcal{P}_2$, a hash function $h : \mathbb{R}^d \to \mathbb{Z}$ is called (R, c, $\mathcal{P}_1$, $\mathcal{P}_2$)-**sensitive** if it satisfies the following conditions simultaneously for any two points p_1, $p_2 \in D$:*

- If $\| p_1, p_2 \|_s \leq R$, then $P_r[h(p_1) = h(p_2)] \geq \mathcal{P}_1$;
- If $\| p_1, p_2 \|_s \geq cR$, then $P_r[h(p_1) = h(p_2)] \leq \mathcal{P}_2$;

Here, $P_r[e(h)]$ means the probability of event $e(h)$, both $c > 1$ and $\mathcal{P}_1 > \mathcal{P}_2$ hold. Also, to improve the distinguishing capacity(reduce the false positives), we apply a **compound LSH function** denoted as $G = (h_1, h_2, \ldots, h_m)$, where $h_1, h_2, \ldots, h_m$ are randomly picked LSH functions from a hash family. Specifically, the **compound hash value** of point p under G is $K = G(p) = (h_1(p), h_2(p), \ldots, h_m(p))$. For simplicity, we call the LSH compound hash values as hash values in the rest of the paper.

Different hash families turn out to have different performance. In the last decade, many hash families are devised for various similarity measures. For example, p-Stable hash family [6] is suitable for ℓ_s distance when $s \in (0, 2]$, random hyper-plane [5],

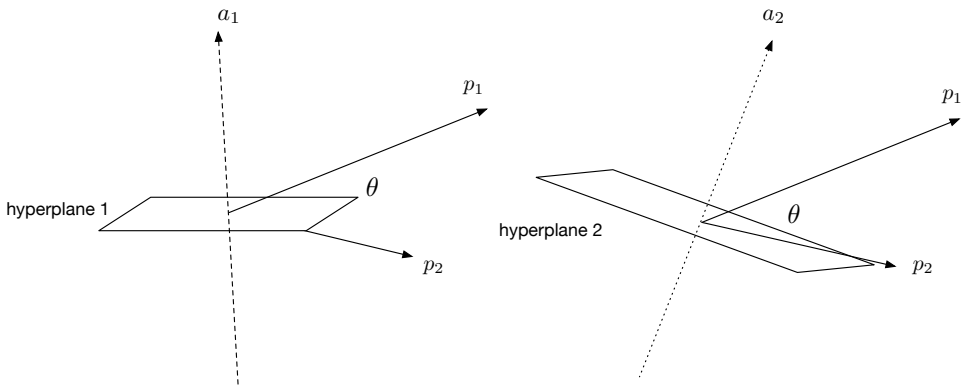

Figure 2.1: Two vectors p_1 and p_2 make an angle θ.

batch orthogonal angle(BOA) hash family [12] and cross-polytope hash family [1] are for cosine distance.

2.3.1 Batch Orthogonal Angle Hash Functions

In this thesis, BOA hash family is used to generate the hash values. As shown in *Definition 3*.

Definition 3 Batch orthogonal angle LSH. *In a d dimensional data space, given an input vector p and an orthogonal projection vector a, we define the hash functions as* $h(p) = sign(p \cdot a)$.

The sign function $sign(\cdot)$ is defined as

$$sign(z) = \begin{cases} 1 & \text{if z} \geq 0 \\ 0 & \text{if z} < 0 \end{cases}$$

Algorithm 1: BOAHashFamily(d, F_s, m)

Input: Dimension of object d; Hash family size $F_s(F_s \leq d)$; Number of hash functions m;

Output: BOA hash family O;

```
1  O = ∅ /*Save the Orthogonal angle Hash family*/ ;
2  G = ∅;
3  Generate a random matrix H with each element x being sampled
     independently from the normal distribution N(0,1). Denote H = [x_{i,j}]_{d×d};
4  Compute the QR decomposition of H, such that H = QR;
5  for i = 1; i ≤ F_s do
6      o = the i-th column of Q;
7      O.add(o);
8  for i = 1; i ≤ m do
9      w = choose a random number from 1 to F_s;
10     G.add(O[w]);
11 return G;
```

$h(p) = 1$ or 0 indicates on which side of the hyperplane p lies. As shown in the Figure 2.1, a_1 and a_2 are two vectors, p_1 and p_2 are two high dimensional points make an angle θ. In the left plot, a_1 determines the hyperplane 1, p_1 and p_2 are projected on the different sides of hyperplane 1, so $h(p_1) = 1$ and $h(p_2) = 0$. In the right plot, again a_2 determines the hyperplane 2, while p_1 and p_2 are projected on the same side of hyperplane 2, so the $h(p_1) = 1$ and $h(p_2) = 1$. Based on this, the probability that the randomly chosen vector a for p_1 and p_2 have the same hash value is (180-θ)/180. Since p and a are normalized during data preprocessing, the angle distance in LSH between p and a is measured as $arccos(a \cdot p)$. Thus, with multiple hash functions, the hash values of objects are binary sequences. For instance, if $m = 8$, $K = 01011000$, which is easy to be saved as bitmap representation in primitive data type(e.g. int and long). Another advantage is calculating the Hamming distance between the binary sequences are mainly bitwise operation, which is much faster than computing other

distance(e.g., Euclidean distance). The data space is also partitioned more accurately by using BOA LSH due to the rectangular region of space partition [12]. Algorithm 1 shows how to generate the BOA hash family.

2.3.2 P-Stable Hash Functions

The p-stable hash functions are used in the compared LSH-based methods is described as follows:

$$h^{a,b}(p) = \lfloor \frac{p \cdot a + b}{w} \rfloor$$

where $\lfloor \cdot \rfloor$ is the floor operation, a is a vector with entries chosen independently from the Gaussian distribution $N(0, 1)$, w is the bucket width, b is a random real variable uniformly chosen from the range $[0, w)$. In intuitively, an LSH function $h^{a,b}(p)$ works as follows: It projects the object p onto a line L_a whose direction is identified by a. Then, the projection of p is shifted by a constant b. With the line L_a being segmented into intervals with size w, the hash function returns the number of the interval that contains the shifted projection of p.

2.3.3 Minimize Index Storage

Generally, the main problem affecting the performance of LSH is the number of hash tables, since each hash table is a whole copy of the dataset in traditional indexing design. To make better use of a smaller number of hash tables, Multi-probe LSH(MPLSH) [19] is designed to reduce the number of hash tables by step-wise probing more perturbs of a query to more near buckets within a hash table. The search algorithm not only considers the main bucket where the query object falls, but also the buckets that "close" to the main bucket. For example, the query object is hashed

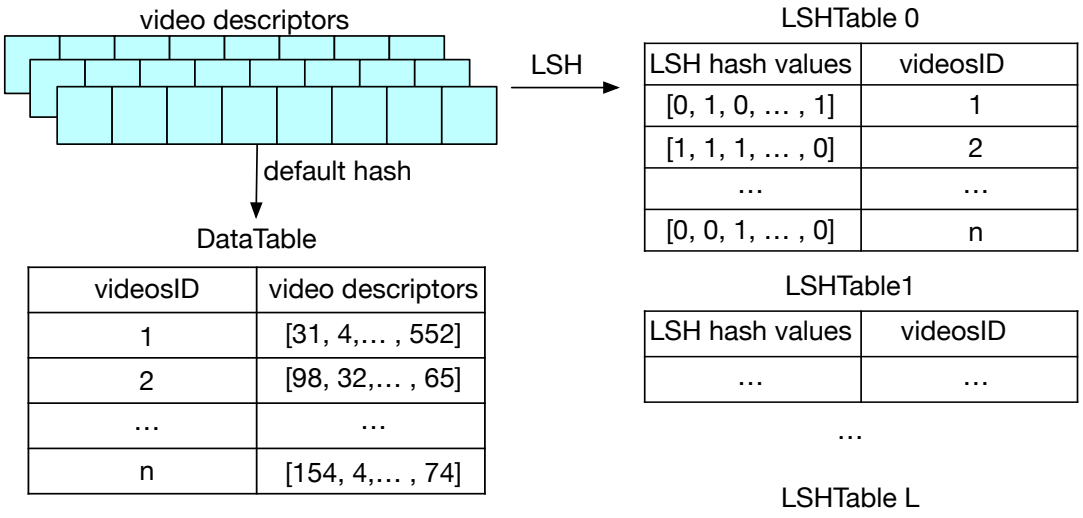

Figure 2.2: The multi-index storage scheme.

to (3,6,4), besides searching the similar objects in the main bucket (3,6,4), the algorithm also searches the other buckets such as (4,6,4), (3,6,4) or (4,5,4), which are close to the main bucket. Through the MPLSH search algorithm, it achieves a high accuracy even with a small number of hash tables.

As shown in the Figure 2.2, Partitioned Hash Forest(PHF) [37] designs a multi-index storage scheme by using two types of tables: 1)DataTable: saves a copy of original dataset as (key, value) for fast querying original vectors 2) LSHTables: only save the keys from DataTable as values for the index. By this design, the index(hash tables) only contains the keys, rather than the original data, which reduces storage space while using enough hash tables to achieve high accuracy. The original design of search algorithm in PHF includes two steps: 1) Search the DataTable by key to find the feature vector; 2) Calculate the feature vector's hash value, then search the all LSHTables to retrieve all candidates. However, the serialization and deserialization

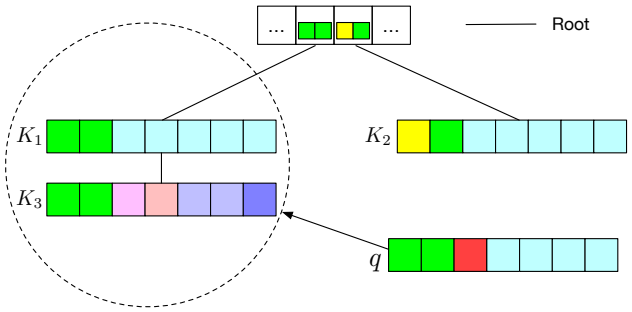

Figure 2.3: The most significant bits problem in current index structure, different colors indicate the different number of hash values.

in the first step cost too much time [37].

In this thesis, we combine the two technologies together to minimize the indexing storage. Also, the search algorithm directly passes the feature vector to calculate its hash value, which avoids the overhead to query the DataTable, results in improving the performance a lot.

2.3.4 Most Significant Bits Problem

Current representative LSH-based index methods [31, 17, 37] use the $1/LLCP(K_1, K_2)$, where $LLCP(K_1, K_2)$ is the *length of longest common prefix*(LLCP) of K_1 and K_2. For example, given $K_1 = 000100$ and $K_2 = 001100$, then $LLCP(K_1, K_2) = 2$, or own defined distance measure approach to calculate the distance between two hash values K_1 and K_2. These indexing strategies have an implicit assumption that different bits in the data item have different significance(e.g., left bits are more important than the right bits in a hash value). However, the hash functions that randomly picked from a pre-generated hash family without priority. The error is involved when minor different bits happen in the left part of the hash value. As shown in the Figure 2.3, K_1,

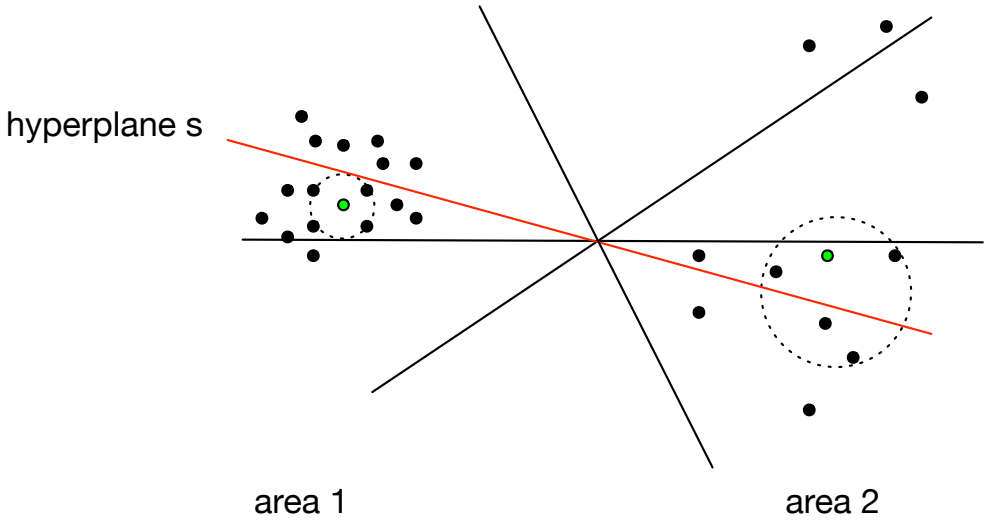

Figure 2.4: The difficulty to make a choice of the number of hash functions.

K_2 and K_3 are three hash values in the index, q is the hash value of the query object. Since K_1 and K_3 have the same first two bits from left, according to the previous indexing strategy, K_1 and K_3 are in the left subtree. K_2 is in the right subtree of the index. However, the distance between q and K_2 is obviously less than the distance between q and K_3 under ℓ_2. When the query q comes, it only retrieves the similar objects in left subtree and ignores K_2 in the right subtree, which affects the accuracy by losing the part of similar objects(e.g. K_2 in the Figure 2.3 in the right subtree).

2.3.5 The Deficiency of Static Compound Hash Functions

Most of the time we don't know the distribution of the dataset. For example, in Figure 2.4, we find the dataset has two main clusters: area 1 and area 2. The density of points in area 1 is much larger than the density in area 2. The red line is hyperplane s. For querying the nearest neighbors of the green point in area 1, it is crucial to add the

hyperplane s. Because the high resolution is needed to remove the irrelevant points in larger density area. The search circle is small as we can see in area 1. However, for querying the nearest neighbors of the green point in area 2, the hyperplane s is not necessary. The reason is in area 2, the small density doesn't need hyperplane s to differentiate the points, since the number of hyperplanes is enough. Thus, the search circle in area 2 is larger. Reflect to LSH, whether we can use the *dynamic* number of hash functions to distinguish different density area in dataset is also a good point to improve the performance.

Some frequently used notations in this paper are given in Table 2.1.

Notation	**Description**
$D = \langle p_1, ..., p_n \rangle$	Index data set consists of n d-dimenional objects
$Q = \langle q_1, ... \rangle$	Query data set that not in D
L	Number of LSHTables
F_s	Size of hash family
m	Number of hash functions in LSHTable
$G(\cdot) = (h_1(\cdot), ..., h_m(\cdot))$	Hash functions in reducing dimension LSH layer, consists of m BOA functions
$G^{'}(\cdot) = (h_1^{'}(\cdot), ..., h_M^{'}(\cdot))$	Hash function in partition LSH layer, consists of M The number of BOA functions in Partition LSH layer
$K = \langle K_1, K_2, ...K_n \rangle$	Hash values in a LSHTable after applying RD LSH layer on D, K_1=$G(p_1)$=$(h_1(p_1), ..., h_m(p_1))$
$K^{'} = \langle K_1^{'}, K_2^{'}, ...K_n^{'} \rangle$	Hash values in a LSHTable after applying partition LSH layer on K, $K_1^{'}$=$G^{'}(K_1)$=$(h_1^{'}(K_1), ..., h_M^{'}(K_1))$
n_s	The number of shuffling permutations
$P(x, y)$	y-permutation of x
$K^{t,i,j}$	Twisted hash values by applying shuffling permutation $P_i(m, m)$ on j-th sub-index, for $1 \leq i \leq n_s$ and $0 \leq j \leq 2^M - 1$
$l = \{l_1, l_2, l_3, \ldots, l_i\}$	The length of d-node in each level i of RDT
T_h	The threshold of k-nodes under the same slot

Table 2.1: Summary of Notations

Chapter 3

Layered Hash for Partitioning the Data

As discussed in the Chapter 1, the first phase is to get the feature descriptors by conducting the feature extraction algorithms on raw dataset. However, this is not the main topic in this thesis. There are many public datasets of feature descriptors for image, video and text, which will be described in Chapter 5.

3.1 Reduce Dimension LSH Layer

Most feature descriptors of image, video, and text are one high dimensional vectors or several high dimensional vectors. It is impossible to directly construct an efficient index for them due to the "Curse of Dimensionality". Therefore, reducing the dimension of these vectors are the second steps for constructing the index. By using the BOA LSH as the first layer LSH, the high dimensional vectors are descended into low dimensional vectors(binary sequences) without losing the "similarity property",

which means the similar objects are most likely to have similar hash values. The algorithm of first layer LSH to generate one LSHTable hash values is described in Algorithm 2. Basically, it contains two parts: 1) Generation of LSH functions in line 1. 2) Computation of hash values as we explained in 2.3.1 in line 5 and save into primitive data type from line 6 to 10.

Algorithm 2: RDLSHLayer(d, F_s, m, D)

Input: Dimension of object d; Hash family size F_s; Number of hash functions in RD LSH layer m; Index data set D

Output: Hash values K;

```
1  G = BOAHashFamily(d,F_s,m);
2  K = ∅;
3  Transfer the G into a d by m matirx, each column is an orthogonal angle hash
     function;
4  Transfer the D into a n by d matrix, each row is d-dimensional feature vector;
5  compute T = sign(D · G) /* · is the matrix multiplication; sign() is a element
     operation apply sign function on each item in matrix*/ ;
6  for i = 1; i ≤ n do
7      k = 0;
8      for j = 1; j ≤ m do
9          compute k = k << 1 | T[i, j];
10     K.add(k);
11 return K;
```

As demonstrated in PHF [37], there is a higher probability to put the content-similar objects in different buckets if the m is too large. However, if m is too small, the hash value may not have enough resolution to achieve "locality sensitive" of raw features. In addition, to reduce the space of index, it is ideal that the hash values are able to be saved as a bitmap. In addition, suppose the dataset is uniformly distributed, the choice of m is recommended to keep at the same order of magnitude of objects. For example, $m = 10$ is the good choice for the dataset contains $2^m = 1024$

objects. However, to design a general index for online query system, the number of objects is always increasing. Thus we set $m_{max} = 32$ and use integer which saved as m binary bits in real implementation, where $2^{m_{max}} = 4294967296$ is large enough for most dataset. Specifically, $K_i = \underbrace{01001011\ldots001}_{m}$.

3.2 Partition LSH Layer

Nonetheless, if we directly build the index based on the whole hash values K, the multiple concurrent queries does introduce memory overheads. To handle the concurrency when multiple queries reach the index, partition strategy is applied to divide the index into as much as possible sub-indexes without losing accuracy. However, it is challenging(if not impossible) to partition data in a way that guarantees that all queries will only need to access a single sub-index since the objects are required to be similar with each other within a sub-index.

Two approaches are developed to divide the index by objects' distance. 1) K-means clustering partition methods [32]: dividing the index based on how many clusters the dataset has. However, due to the frequent update, it's difficult to implement suitable parameters to achieve high performance for the online system. 2) Partition LSH [2, 37]: after calculating the hash values K from RD layer of LSH, we apply a new set of M LSH functions as partition LSH layer on K. For each hash value K_i, the result $K_i^{'}$ is an M long bit sequence, which indicates the sub-index that the object belongs to. For example, if $M = 2$, $K_i^{'} = 01$, the K_i belongs to sub-index-01. Based on the property of LSH as we introduced in Section 2.3, the principle behind partition LSH layer resides here:

1). Similar objects have high possibility to have the similar hash values after RD LSH layer.

2). Similar hash values have high possibility to have the similar hash values(sub-index-ID) after partition LSH layer.

The algorithm of partition LSH layer is presented in Algorithm 3. Basically, it consists of two parts: 1) Generation of LSH functions in line 1. 2) Computation of sub-index-ID for each hash value and save into primitive data type from line 2 to 12.

Algorithm 3: PLSHLayer(m,F_s,M,K)

Input: Number of hash functions in RD LSH layer m; Hash family size F_s; Number of hash functions in parititon LSH layer M; Hash values K;
Output: sub-index-ID $K^{'}$;

```
1  G' = BOAHashFamily(m,F_s,M);
2  K' = ∅;
3  for i = 1; i ≤ n do
4      Vector tmpK = zeros(m);
5      ID = 0, tmpBit = 0;
6      for j = 0; j < m do
7          compute tmpK(j) = (K.get(i) & (1 << j)) >>> j;
8      for t = 1; t ≤ M do
9          compute tmpBit = sign(tmpK · G'.get(t));
10         compute ID = ID << 1 | tmpBit;
11     K'.add(ID);
12 return K';
```

The distributed version of in-memory online similarity search system, each machine keeps one sub-index in the memory. When the system handles concurrent queries, each query only accesses to one sub-index(or machine). Through this, there is no overhead network delay between the work nodes in the distributed version of the

system, which improves the network efficiency. The parameters M influences the concurrency handling capacity of the system. Specifically, 2^M sub-indexes are generated. The larger M we set, the more robust ability of the system to handle concurrency.

Even though the n objects are divided into 2^M pieces, the balanced distribution of data percentage in each sub-index for real dataset is important to the system performance. The partition LSH layer turns out to be a computation waste if the unbalanced distribution of data percentage occurs. For example, before the partition LSH layer, only one piece contains n objects. Suppose $M = 2$, the n objects are separated into 4 sub-index, denoted as sub-index-00, sub-index-01, sub-index-10, sub-index-11. However, if the corresponding data percentage over these sub-indexes is like 5%, 3%, 2% and 90%. It is regarded as a bad partition strategy since the largest percentage is 90%, which is almost close to the whole dataset. Most of the queries still retrieve the sub-index-11 rather than other three sub-indexes. The unbalanced workload makes most the of queries slow as before the partition. Therefore, the balanced distribution of data percentage is an important evaluation metric for partition strategy.

To evaluate the stability of Partition LSH layer, we count the number of objects in each sub-indexes. The dataset GloVe [24], SIFT [11], NYTimes [16], Fashion-MNIST [35] and CC_WEB_VIDEO [34] are used. After applying the Partition LSH layer, each sub-index is ideal to keep a $100/2^M\%$ percentage of the whole dataset and the sum of these rates is equal to 100%. Due to the fact that only one sub-index will be retrieved. The query's search range drops off dramatically if the percentage is stable. The σ(standard deviation) is calculated to measure how far the percentage in each sub-index are spread out from the ideal percentage, which makes the results more

Table 3.1: Stability of Partition LSH Layer, each number in a column is the data percentage of sub-indexes. σ is standard deviation, which is used to measure how far the percentage in each sub-index are spread out from the ideal(average) percentage. For each dataset, $m = 20, 20, 18, 16, 14$ respectively.

M=2	GloVe	SIFT	NYTimes	Fashion-MNIST	CC_WEB_VIDEO
sub-index-00	28.0	23.8	23.8	**15.0**	26.0
sub-index-01	25.1	21.7	24.9	**30.8**	18.2
sub-index-10	24.1	25.2	28.1	23.9	26.4
sub-index-11	22.8	29.3	23.2	30.3	29.4
σ	1.91	2.78	**1.89**	6.38	4.14

M=3	GloVe	SIFT	NYTimes	Fashion-MNIST	CC_WEB_VIDEO
sub-index-000	14.8	5.5	18.1	20.5	9.2
sub-index-001	6.7	5.3	13.6	19.1	13.7
sub-index-010	10.2	20.1	11.1	11.1	11.1
sub-index-011	8.6	15.0	12.9	9.3	13.7
sub-index-100	**25.6**	14.3	13.9	11.9	19.1
sub-index-101	8.4	**5.2**	8.9	13.4	11.8
sub-index-110	14.6	17.1	10.8	8.5	8.7
sub-index-111	11.1	17.5	10.7	6.2	12.7
σ	5.64	5.78	**2.64**	4.70	3.05

M=4	GloVe	SIFT	NYTimes	Fashion-MNIST	CC_WEB_VIDEO
sub-index-0000	5.2	3.6	10.5	8.2	10.3
sub-index-0001	4.7	7.7	5.3	4.9	7.9
sub-index-0010	4.5	11.6	5.3	4.9	2.1
sub-index-0011	3.0	2.1	2.4	4.2	3.9
sub-index-0100	7.1	4.9	7.3	2.2	7.7
sub-index-0101	4.9	4.7	8.1	3.9	5.2
sub-index-0110	5.8	8.7	8.9	4.7	4.3
sub-index-0111	**1.6**	2.4	8.0	6.3	6.0
sub-index-1000	12.7	7.1	6.3	8.9	8.9
sub-index-1001	3.1	6.9	5.1	14.4	9.5
sub-index-1010	10.8	10.5	5.4	4.6	2.2
sub-index-1011	2.5	7.2	2.8	12.0	2.8
sub-index-1100	**17.2**	7.1	4.0	2.7	7.7
sub-index-1101	4.0	4.5	9.6	7.7	5.1
sub-index-1110	11.0	5.3	3.8	2.1	8.7
sub-index-1111	1.9	5.7	7.1	8.3	7.8
σ	4.29	2.56	**2.32**	3.37	2.58

obviously. In Table 3.1, we test the different M for each dataset. When $M = 2$, Partition LSH layer has good performance for all datasets, especially the GloVe and NYTimes with lower σ. The data percentage of each sub-index is around 25%. For the worst dataset Fashion-MNIST, the highest rate is 30.8%, and the lowest is 15.0%. It means the worst case of search is only need to look up 30.8% of the whole dataset. When $M = 3, 4$, more sub-indexes are generated to support concurrent search. The NYTimes has lower σ than other four datasets. The smallest and largest rate of $M = 4$ is 1.6% and 17.2% in GloVe. It is acceptable since the σ is only 4.29. However, the tiny rate(e.g. less than 0.5%) is unsatisfactory for the distributed vision. The tiny rate is only able to deal with a tiny part of queries. However, all the workers are normally configured as the same setup in the current distributed framework. Thus, the computation waste occurs in tiny rate workers, and the overhead happens in large rate workers. Based on the Table 3.1, after the partition LSH layer, the cardinality of data in each sub-index is stable, which reduces a large searching range for similarity search.

3.3 Look Up Δ-step Sub-indexes

The ideal partition strategy is dividing all the similar objects into one sub-index. However, due to the approximate property of the partition layer LSH, similar objects are still likely to be divided into the different sub-indexes, which results in degrading the accuracy and consistency of the system. To increase the accuracy, a Δ-step search approach is designed based on another LSH property: The sub-indexes that are one step away are most likely to contain objects that are close to the query object than sub-indexes that are two steps away.

There are only two possible values 0 or 1 in each bit of a hash value, we denote the Hamming distance between two $K^{'}s$ as Δ and $\Delta_{max} = M$. To prove that the Hamming distance between $K^{'}s$ after BOA partition LSH layer has an unbiased estimate of the angle(similarity) between the corresponding two given Ks from RD LSH layer, we first have **Lemma 1** [8](*Lemma* 3.2).

Lemma 1 *Given a random vector r drawn uniformly from the unit sphere S^{d-1} in $\mathbb{R}^d$, any two vectors v_i and v_j from S^{d-1}, we have*

$$P_r[sign(v_i \cdot r) \neq sign(v_j \cdot r)] = \frac{\theta_{v_i,v_j}}{\pi}$$

Since the BOA LSH is used in this thesis, as defined in Section 2.3.1, the hash functions are orthogonal vectors and the objects are all normalized before hashing. Thus, we have

$$P_r[h_r(v_i) \neq h_r(v_j)] = \frac{\theta_{v_i,v_j}}{\pi}$$

As the hash functions contain $sign(\cdot)$, which means the final result only depends on the positive or negative of the vectors inner product. So the normalized vectors from the RD LSH layer have the same hash result as the non-normalized vectors. Through the former proof in [12], we have

Theorem 1 *Given M orthorgonal vectors $h_1, h_2, \ldots, h_M$ from the orthogonal angle hash family, then for any two normalized binary vectors $p, q \in S^{m-1}$ from the RD LSH layer, by defining M indicator random variables $X_1^{p,q}$, $X_2^{p,q}$, $\ldots$, $X_M^{p,q}$ as*

$$X_i^{p,q} = \begin{cases} 1 & h_i(p) \neq h_i(q) \\ 0 & h_i(p) = h_i(q) \end{cases}$$

We have $\mathbb{E}[X_i^{p,q}] = P_r[X_i^{p,q} = 1] = P_r[h_i(p) \neq h_i(q)] = \frac{\theta_{p,q}}{\pi}$, *for any* $1 \leq i \leq M$.

So the expectation of Δ is

$$\mathbb{E}[\Delta] = \mathbb{E}[d_{Hamming}(h(p), h(q))] = \mathbb{E}[\sum_{i=1}^{M} X_i^{p,q}]$$

$$= \sum_{i=1}^{M} \mathbb{E}[X_i^{p,q}] = \sum_{i=1}^{M} \theta_{p,q}/\pi = C\theta_{p,q}$$

where $C = M/\pi$. Thus, smaller Δ-step sub-index is more likely to contain the hash values near to the query's hash value. It also explains that why we search from the original(0-step) sub-index, then the 1-step sub-indexes.

To generate the Δ-step sub-indexes, we only need to apply +1(for bit=0) or -1(for bit=1) on the Δ number of bits in original sub-index-ID. Suppose the original sub-index-ID of q_1 is $K_1^{'}$, so the 1-step sub-index-ID is applying +1/-1 operation on one random bit of $K^{'}$, the 2-step is applying +1/-1 operation on two random bits of $K_1^{'}$ and so on. The total number of Δ-step wise sub-index is $\binom{M}{\Delta}$. For example, if $M = 3$, as we can see from the Figure 3.1, the original sub-index-ID is 010, the 1-step sub-index-IDs are 110, 000, 011, the 2-step sub-index-IDs are 100, 111, 001, the 3-step sub-index-IDs is 101.

To see whether our partition LSH layer can efficiently divide the similar objects into one sub-index. Figure 3.2 shows the distribution of different sub-index of top k nearest neighbors on GloVe dataset. Since each queries' top k nearest neighbors will fall into different sub-indexes, we calculate the 2000 queries' top k nearest neighbors' Δ-step sub-index and get the average distribution. As we can see from the plots, 90%

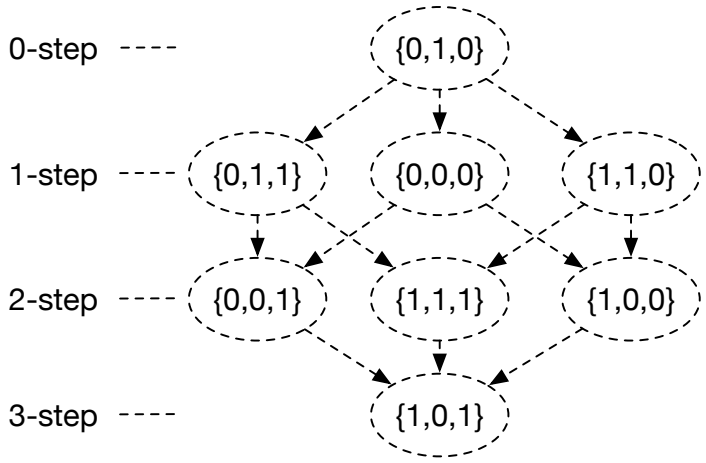

Figure 3.1: Different Δ-step sub-index-ID graph with $M = 3$. Each dotted line represents one step.

of the top k nearest neighbors are hashed in the original sub-index even with the increment of k. When the M rises, the top k nearest neighbors are lightly decentralized into different steps sub-index, but the original(0-step) sub-index still maintain a high percentage 88%. The efficiency of parallelism search over index is earned by losing part of the ground truth. Another conclusion we find that most mistakenly partitioned top k nearest neighbors are more likely to fall into 1-step sub-indexes rather than 2-step sub-indexes. Thus, we adopt the 1-step sub-indexes look up to improve the accuracy if the requirement is high accuracy. The impact of Δ on accuracy and efficiency is discussed in the Section 5.4.4.

In conclusion, the Reduce Dimension LSH layer transfers the high dimensional vectors into low dimension representations. To partition the big dataset, we use the Partition LSH layer to divide the data into several sub-indexes. Each of the sub-index only shares a small percentage of data. We also retrieve the mistakenly partitioned similar objects through a Δ-step sub-indexes lookup strategy.

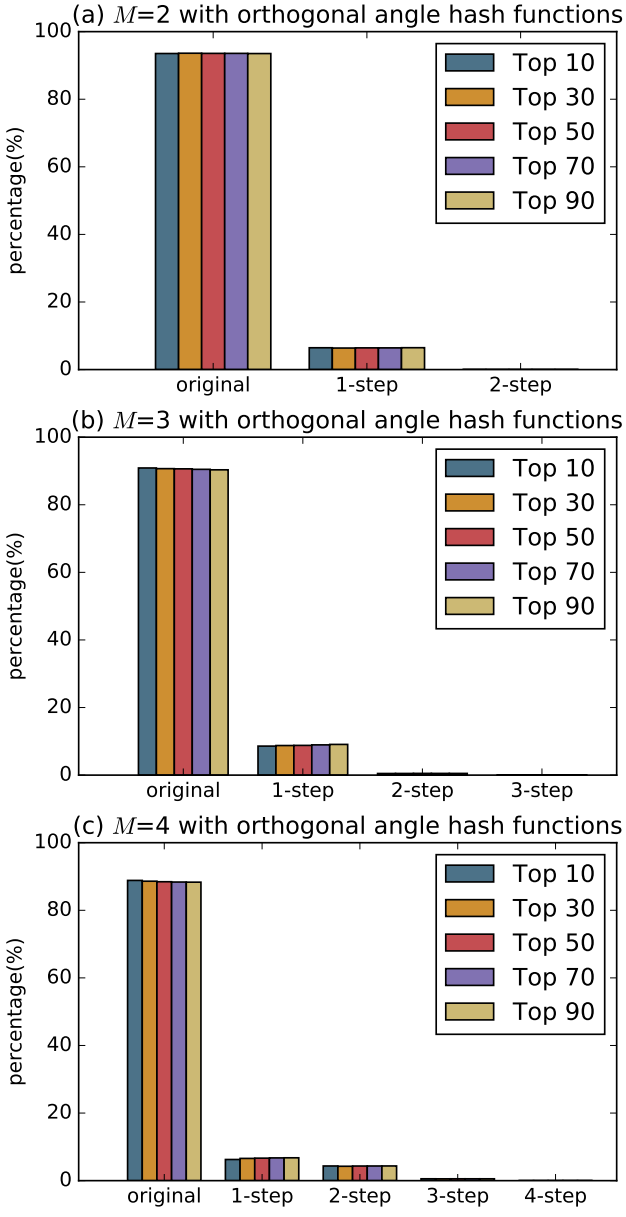

Figure 3.2: Δ-step sub-index distribution on GloVe. $m = 19$

Chapter 4

Random Draw Forest

As shown in Figure 4.1, the step 1,2,3,4 shows that after reducing the dimensionality of feature data by applying layered hash as described in Chapter 3, every sub-index consists of a sufficient number of hash values, whose dimensionality m is around $log_2(n)$. However, what if the n reaches 1000,000. The $log_2(n) \approx 20$, which is still a large number for the traditional tree structure. Also, the number of n is unknown or always incremental in the practical online similarity search system. If we directly search the hash values, the complexity is O(log(n)) if we use binary search. Therefore, it is necessary for us to design an index structure which not only supports to fast search the ANN of query's hash value but also provides a dynamic m to solve the deficiency as we discussed in section 2.3.5. Also, the MSB problem as mentioned in Section 2.3.4 need to be conquered to maintain a high accuracy by capturing enough top k of the query.

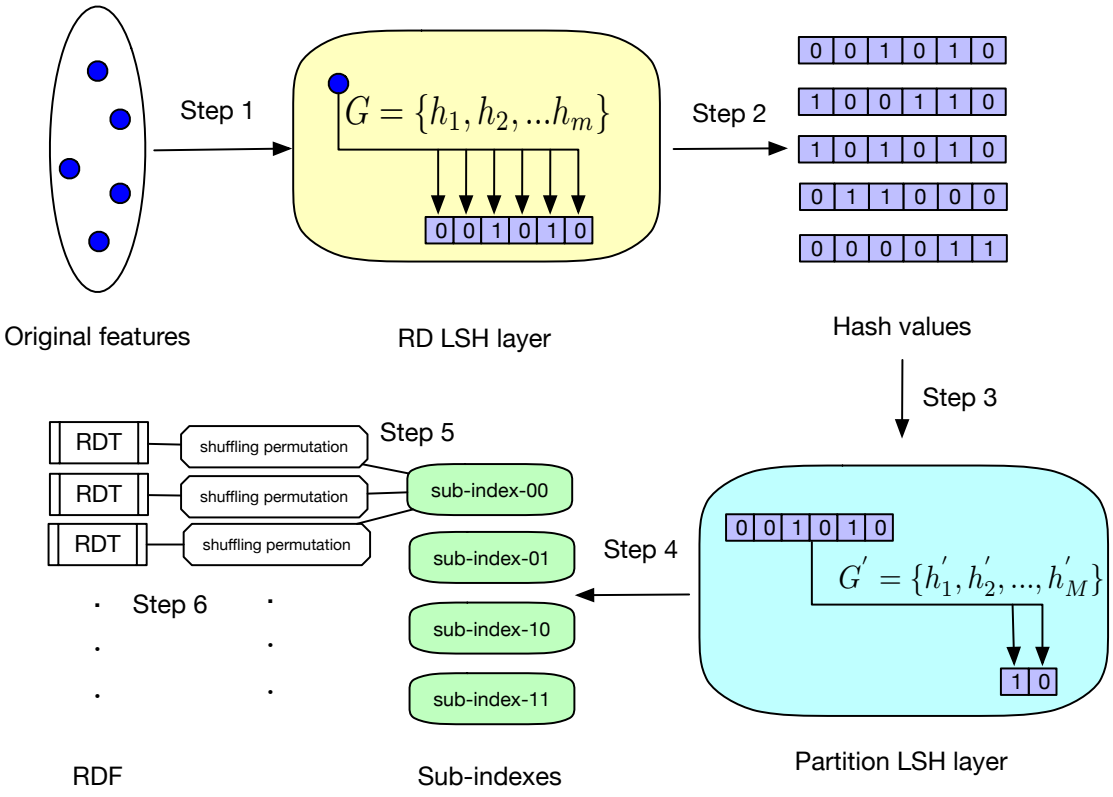

Figure 4.1: In the example, $m = 6$ and $M = 2$, so we have $2^M = 4$ sub-indexes for parallelism. In each sub-index, the hash values are converted to twisted hash values by applying several shuffling permutations on hash values. Each set of twisted hash values corresponding to a random draw tree(RDT), multiple of RDT make up the RDF.

4.1 Index Structure

The random draw forest(RDF) consists of random draw trees(RDT). The structure of the RDT is similar to the R-tree which is formed by hierarchically partitioning the hash values in each sub-index. The difference is when the level goes deeper, the more bits of hash values are evaluated to determine the position, results in making the real m adaptively for different dataset, which is inspired by PHF [37]. To minimize the

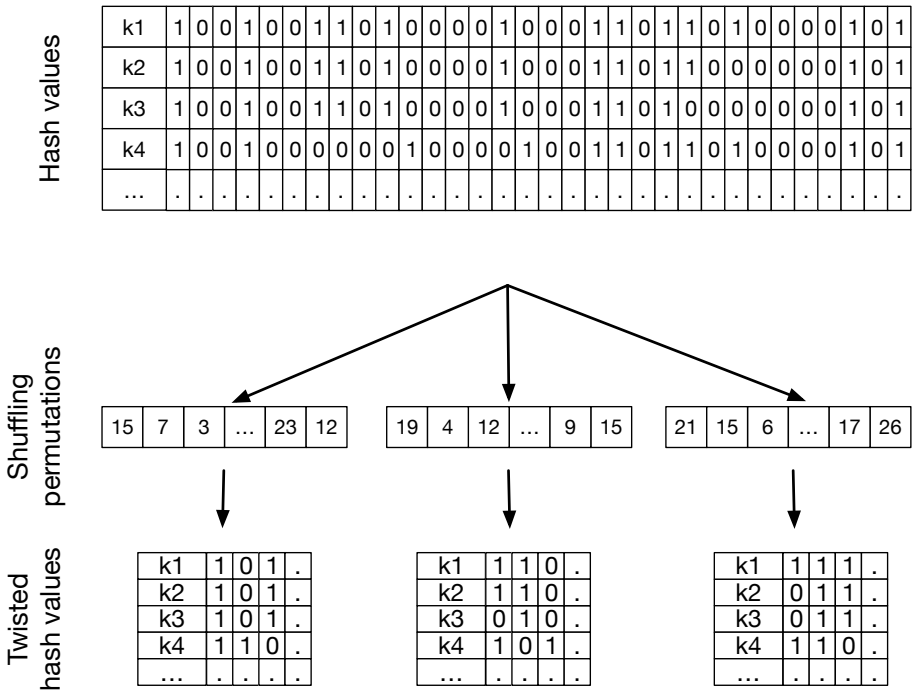

Figure 4.2: Generation of twisted hash values.

index space, the multi-index storage scheme as described in Section 2.3.3 is used. As the leaf-nodes and internal nodes in the tree structure, we introduce two types of nodes: (1) k-node: contains two fields KEY and $POINT$, KEY is the objectID in DataTable, and $POINT$ keeps the reference to the next k-node in the same slot. (2) d-node: an array contains l slots, which is mutable in different levels, and we treat each slot as a bucket in an LSHTable. The value in each slot saves the reference to the first k-node in the slot or the first d-node in the slot.

4.2 Construction of Random Draw Tree

To overcome the MSB problem, instead of building the index tree directly on the hash values, we randomly generate n_s shuffling permutations $P_(m, m)$ for each sub-index, as shown in Figure 4.1(step 5). Based on these permutations, we then have n_s set of twisted hash values as depicted in Figure 4.2. For example, $m = 10$, given a hash value $K_i = 0111001010$, the $P(10, 10) = (9, 6, 1, 4, 3, 10, 2, 8, 5, 7)$, the twisted hash value is $K^t = K \odot P(10, 10) = 1001101001$. To facilitate the query speed, each set of twisted hash values corresponding to a random draw tree(RDT). In Figure 4.3, the example shows how to build the RDT based on i-th twisted hash values of j-th sub-index $K^{t,i,j}$. The detail steps of inserting an twisted hash value from $K^{t,i,j}$ into a RDT are explained as follows:

- **Initialization**: To simplify, let $\hbar$ denote this twisted hash value. Initializing $level = 1$, The l_{level} long d-node is used as root d-node(also treated as level 1). Then we calculate the number of bits used in root d-node to determine slot by $\log_2(l_{level})$. The max level of tree is depended on the $l_1, l_2, \ldots, l_{max}$ we set.

- **Step 1**: According to the first $\log_2(l_{level})$ bits of $\hbar$, we generate a Integer range from 0 to $l_{level} - 1$ as the slot of level 1. For instance, if $l_1 = 32$, $m = 10$, $\hbar = 1001001101$, and $\log_2(l_1) = 5$. Then first 5 bits extracted from $\hbar$ is $(10010)_b = 18$ to determine the slot in root level of the RDT.

- **Step 2**: If the slot has not been occupied, we update the value in the corresponding slot of the root node as the address of object whose twisted hash value under i-th permutation is $\hbar$ in storage space and terminate the insert processing. The *Insert k1* and *Insert k4* in figure 4.3 shows this step.

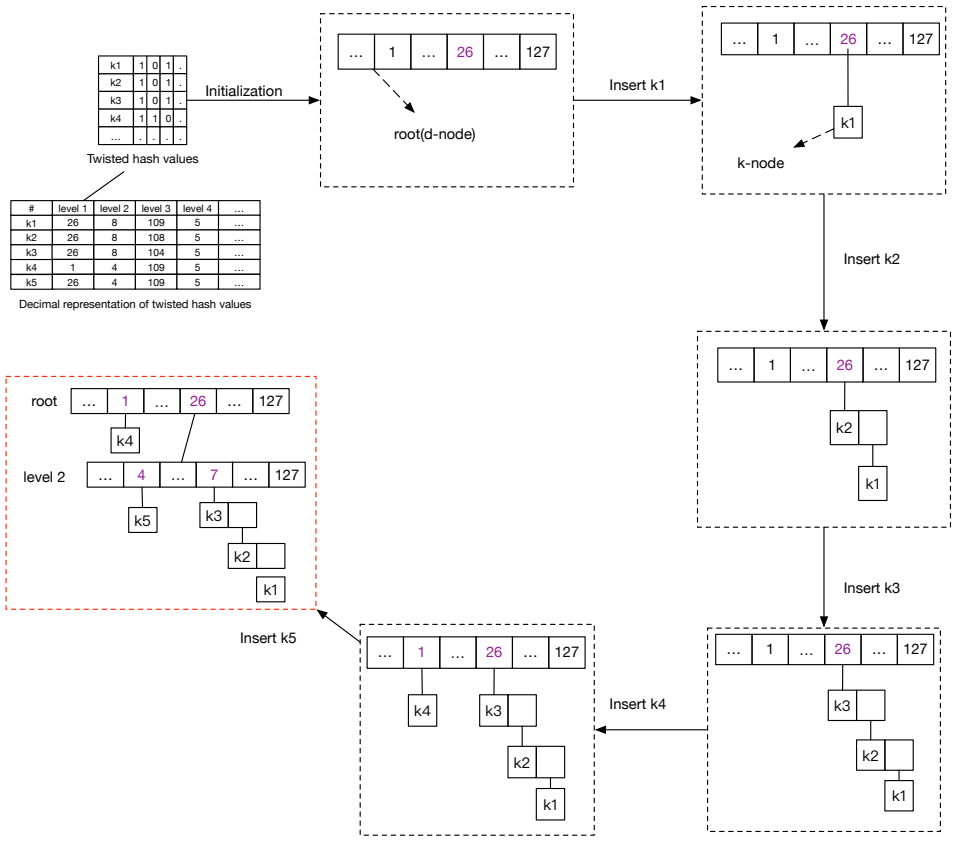

Figure 4.3: Random draw tree: we progressively include more bits of twisted hash values to locate them in the certain level of d-node. When there are more than T_h nodes under the same buckets, we redistribute them to the next level of the hash tree. For this example, $T_h = 3$ and $l_1 = l_2 = 128$.

- **Step 3**: If the slot has been occupied, and the corresponding node is d-node, we do $level = level + 1$, then progressively use the next $log_2(l_{level})$ bits of $\hbar$ as the slot in the current level d-node, then go back to **Step 2**.

- **Step 4**: If the slot has been occupied, and the corresponding node is k-node, also the number of objects under this slot is equal or less than T_h, we insert $\hbar$ as a k-node under this slot in the RDT, then terminate, as *Insert k2* and *Insert*

k3 show in figure 4.3. If the number of objects under this slot is larger than T_h, we add a new d-node under this slot, then go to **Step 5**.

- **Step 5**: $level = level + 1$, we progressively use next $log_2(l_{level})$ bits of $\hbar$ as the slot in the new d-node, and redistribute the k-nodes under the former slot in this new d-node, as *Insert k5* describes in figure 4.3. If the number of objects in the new d-node under one slot is still larger than T_h, we do **Step 5** repeatedly until less than T_h or reaching to the max level, then terminate. So, it means at the max level, there is no T_h limitation for each slot in d-node.

The pseudocode of the index algorithm is shown in Algorithm 4, as follows the steps described above. Through the RDT construction steps, we totally have $2^M n_s L$ random draw trees.

4.3 Search Strategy

As shown in the Figure 1.1, after building the index, the query operation includes the following steps:

- **Generate compact representations:** The steps are the same as building the index.
- **Index search strategy:** Based on the query's twisted hash value, search the RDF, the evaluation set is narrowed down to a *Candidiates set*, which is likely to hold the similar objects to query.
- **Filtering:** Assessing the *Candidiates set* through their hash values, filter the bad candidates.

- **Ranking:** The remaining high-quality results in *Candidiates set* are evaluated by comparing the distance with the raw features. The high ranking results will be presented.

The basic index search strategy for query q is described as following:

1. By applying the n_s shuffling permutations, we get the twisted hash values as $K^{t,1}, K^{t,2}, \ldots, K^{t,n_s}$.

2. Using the $K^{t,1}, K^{t,2}, \ldots, K^{t,n_s}$ to search each corresponding hash trees. The search starts from the root node(level=1). Take the $K^{t,1}$ as an example, we take the first l_{level} bits of $K^{t,1}$ to calculate the slot number in root node.

3. If the content in the slot are all the k-nodes, then return all these k-nodes as candidates, and terminate. Otherwise, go 4.

4. If the content in the slot is a d-node, then level=level+1, and continually take the next l_{level} bits of the $K^{t,1}$ to calculate the slot number in this d-node. Again, evaluating the content under this new slot as said in 3.

To achieve a higher recall with smaller index storage, we involve the multi-probes strategy [19] to generate multiple tweak queries to search over RDF. The Δ-step sub-indexes lookup strategy is also applied as we discussed in Section 3.3. The algorithm of search one RDT is shown in Algorithm 5. The way to generate multi-probes is similar to the way generate Δ-step sub-indexes, because the Hamming distance between K's has the unbiased estimate of similarity between the corresponding objects.

Algorithm 4: Index(d,F_s,m,D,M,$P^i(m,m)$,l,T_h)

Input: Dimension of object d; Hash family size F_s; Number of hash functions in RD LSH layer m; Index data set D(Here D is a key-value map); Number of hash functions in parititon LSH layer M; i-th shuffling permutation $P^i(m,m)$; Number of slots in each level $l = l_1, l_2, \ldots, l_{max}$,Threshold of k-nodes under the same slot T_h

Output: Index I

```
1  I = initialize 2^M number of empty RDTs, each RDT is a bitmap;
2  /*Bitmap is space save design, which is better than ArrayContainer. For example,
     for 128 int array, if we use bitmap, we only need 128 bits, totally 4 int array.*/;
3  K = RDLSHLayer(d,F_s,m,D);
4  K' = PLSHLayer(m,F_s,M,K);
5  K^{t,i} = K ⊙ P^i(m,m) /*Applying i-th permutation on K, get a re-ordered K^{t,i}*/ ;
6  max=l.length;
7  maskArray=Array(l_1 − 1, l_2 − 1,..., l_max − 1);
8  for j = 0; j < n do
9      curHash=K^{t,i}.get(j);
10     curSID=K'.get(j);
11     curRDT=I.get(curSID);
12     level = 1;
13     while true do
14         slot = (curHash >>> (m − Σ_{w=1}^{level} log_2(l_w))) & maskArray(level);
15         (valueInSlot, nodeType)=curRDT.find(slot,level); /*Find the slot in
             bitmap, to see if it is 0(empty) or 1(non-empty) */;
16         if valueInSlot is 0 then
17             curRDT.add(D.getKey(j),level,slot);
18             break;
19         else
20             /*If it is non-empty, to see what type of node under this slot*/;
21             if nodeType is d-node then
22                 level = level + 1;
23                 continue;
24             else
25                 /*It is k-node, traverse the all k-nodes to check the number of
                     objects under this slot*/;
26                 numOfObjects=curRDT.add(D.getKey(j),
27                 level,slot);
28                 if numOfObjects ≤ T_h then
29                     /*create a new d-node in next level and redistribute the objects
                         in next level*/;
30                     curRDT.addDNode(slot,level);
31                     curRDT.redistributeOldObjects();
32                     curRDT.add(D.getKey(j),level+1);
33                 break;
34 return I;
```

Algorithm 5: Search(Q,d,F_s,m,I,M,$P^i(m,m)$,l,Δ)

Input: Query data set Q; Dimension of object d; Hash family size F_s; Number of hash functions in RD LSH layer m; Index I; Number of hash functions in parititon LSH layer M; i-th shuffling permutation $P^i(m,m)$; Number of slots in each level $l = l_1, l_2, \ldots, l_{max}$; Δ-step;

Output: Result R

```
1  R = ∅/* a empty set list*/;
2  K = RDLSHLayer(d,F_s,m,Q);
3  K' = PLSHLayer(m,F_s,M,K);
4  K^{t,i} = K ⊙ P^i(m,m) /*Applying i-th permutation on K, get a re-ordered K^{t,i}*/ ;
5  max=l.length;
6  maskArray=Array(l_1 − 1, l_2 − 1,..., l_max − 1);
7  for j = 0; j < n do
8      tmpR=∅;
9      Hash=K^{t,i}.get(j);
10     multiProbes=GenerateMultiProbes(curHash);
11     curSID=K'.get(j);
12     /* The way to generate Δ-step sub-indexes is described in Section 3.3*/;
13     Δ-stepSID=GenerateDeltaSID(curSID,Δ);
14     curRDTs=I.get(Δ-stepSID);
15     level = 1;
16     while multiProbes.hasNext do
17         curR=∅;
18         curHash=multiProbes.next();
19         while true do
20             slot = (curHash >>> (m − Σ_{w=1}^{level} log_2(l_w))) & maskArray(level);
21             (valueInSlot, nodeType)=curRDTs.find(slot,level); /*Find the slot in
                bitmap, to see if it is 0(empty) or 1(non-empty) */;
22             if valueInSlot is 0 then
23                 tmpR.add(curR);
24                 break;
25             else
26                 /*If it is non-empty, to see what type of node under this slot*/;
27                 if nodeType is d-node then
28                     level = level + 1;
29                     continue;
30                 else
31                     /*It is k-node, retrieve the all k-nodes under this slot*/;
32                     curR=curRDTs.get(level,slot);
33                     tmpR.add(curR);
34                     break;
35     R.add(tmpR);
36 return R;
```

4.4 Analysis of Index

To see whether random draw solves the MSB problem, we first analyze it in probability view. Due to the fact that K_1 and K_2 won't be treated as similar to each other if they don't in the same slot at root level of d-node, we only need to consider the first level to calculate the probability of K_1 and K_2 in the different slot in one RDT, denoted as $Pr[K_{1,level=1} \neq K_{2,level=1}]$. Let $\mathcal{D}$ denotes the Hamming Distance between K_1 and K_2. The

$$\mathbb{E}[\mathcal{D}] = \frac{m\theta_{K_1,K_2}}{\pi}$$

$$\begin{aligned} Pr[K_{1,level=1} \neq K_{2,level=1}] &= 1 - \frac{C^{\mathcal{D}}_{m-\log_2(l_1)}}{C^{\mathcal{D}}_m} \\ &= 1 - \prod_{i=1}^{\mathcal{D}} \Big(1 - \frac{log_2(l_1)}{m-i+1}\Big) \end{aligned}$$

Thus, the probability of K_1 and K_2 are treated as similar objects in RDT is

$$Pr[K_{1,level=1} = K_{2,level=1}] = \prod_{i=1}^{\mathcal{D}} \Big(1 - \frac{log_2(l_1)}{m-i+1}\Big)$$

Here, ! is the factorial. After doing n_s shuffling permutation, when the query's hash value is K_1, the final probability of the candidates set contains object p_2(whose hash value is K_2) is

$$P_s = 1 - (1 - Pr[K_{1,level=1} = K_{2,level=1}])^{n_s}$$

It's hard to find the obvious clue from above formula, so we analyze it based on the real parameters in our implementation: $m = 20$, $l_1 = 32$, n_s can be any Integer. As depicted in the left plot from Figure 4.4, P_s decreases as the $\mathcal{D}$ increases, the objects

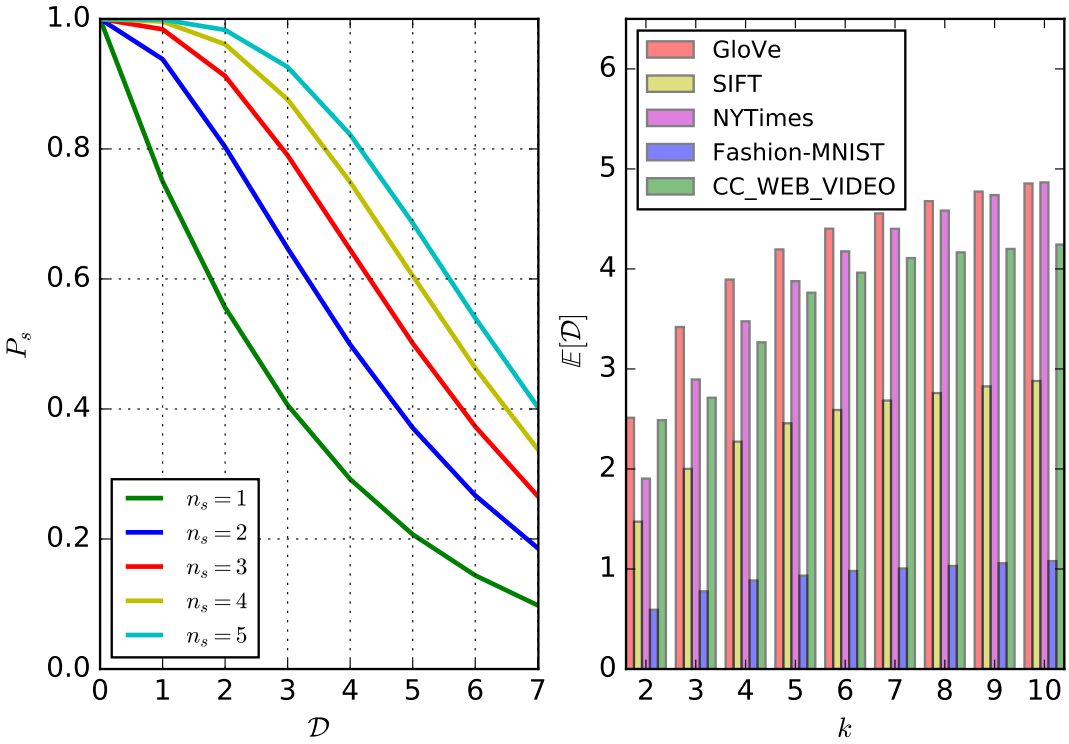

Figure 4.4: The left plot is P_s with different n_s and $\mathcal{D}$. The right plot is expectation of $\mathcal{D}$ with different top k ground truth, we set $L = 10$, $m = 20, 20, 18, 16, 14$ respectively to calculate average $\mathcal{D}$ of 2000 queries.

with a smaller $\mathcal{D}$ between their hash values are more likely to be indexed as similar objects in the RDT. Particularly, P_s declines dramatically when $n_s = 1$, since a large number of ground truth objects are excluded due to the MSB problem. For the same $\mathcal{D}$, P_s rises with the increment of n_s. It means more similar objects at the same degree of $\mathcal{D}$ are retrieved to improve the quality of candidates.

As for the real datasets, our system builds the index by using the hash values, which are supposed to represent the feature of real objects. However, "Similarity" of objects is a rough concept. The accuracy loss is already involved when the raw features are transferred into hash values, while the "locality sensitivity" is kept. In

another word, similar objects have several different bits in their hash values. In the right plot of Figure 4.4, the expectation of $\mathcal{D}$ is evaluated by the various top k ground truth over different datasets. Two important conclusions can be seen from this: 1) Even when we search the top 2 nearest neighbor, $\mathcal{D} \approx 2$ and $P_s \approx 0.55$ due to the MSB problem. 2) When k reaches 10, the expectation of $\mathcal{D}$ increases to around 4. Combined with the two plots in figure 4.4, P_s of $n_s = 5$ is incredibly larger than the P_s of $n_s = 1$ when $\mathcal{D} = 4$, which proves the importance of shuffling permutations in the RDF.

Another advantage of RDT is we don't need to care about the real m for different datasets with different cardinality or distribution. As we discussed in Section 2.3.5, the fixed length of hash values may cause the two problems: 1) have too many candidates in smaller density area 2) don't have enough candidates in larger density area. However, in RDT, if the cardinality of objects in a certain area is larger than the threshold T_h, the level will increase adaptively. Thus, more bits of hash value will be involved to calculate the position in deeper d-node. The relationship between n, l and T_h is roughly like

$$\frac{n}{\prod_{i=1}^{level_a} l_i} < T_h < \frac{n}{\prod_{i=1}^{level_a - 1} l_i}$$

Let's denote the actual m as m_a and actual $level$ as $level_a$

$$m_a = \sum_{i=1}^{level_a} log_2(l_i)$$

In the practical online similarity search system, since we never know the actual cardinality of the data, by using the RDT, it will choose the best m_a for data.

The variable l controls the number of bits to locate the objects in different level of

RDT. Given a m, we have to keep $2^m \geq l_1 \times l_2 \times \cdots \times l_{max}$. For instance, given $m = 20$, $l = (l_1, l_2, \ldots, l_{max}) = (16, 16, 16, 16, 16)$, so $\log_2(l_1) = \log_2(l_2) = \log_2(l_3) = \log_2(l_4) = \log_2(l_5) = 4$, the system always uses 4 bits of twisted hash value to determine the slot in d-node for every level. By this design, each level is treated with the same degree of resolution. Nonetheless, the adaptive resolution in different levels is more convincing to capture enough ground truth or improve the value of candidates. Therefore, the variable bits for each level is more appealing. It means the number of bits to locate the hash value are adaptive for different levels. For example, we set $l = (l_1, l_2, \ldots, l_{max}) = (4, 8, 16, 32, 64)$, so $\log_2(l_1) = 2, \log_2(l_2) = 3, \log_2(l_3) = 4, \log_2(l_4) = 5, \log_2(l_5) = 6$, through this l, in the root level of RDT, we only use 2 bits in calculate the slot, so the system doesn't lose too much ground truth in first level, it helps the small number similar objects group gain enough efficient candidates. In the next levels, we increase more bits as 3,4,5,6 for each level, because the only condition for objects going deeper is number of the "similar" objects under the same slot is equal or larger than T_h, therefore, for the depper level, we need to increase resolution to make these "similar" objects be divided into different "similar" groups.

For each query, the time includes three parts:

- Hash the query: calculate the hash value and sub-index-ID costs $O((d+M)mL)$, twist the hash values costs $O(m^2 n_s L)$
- Retrieve the RDF: Since retrieve process is directly use the twist hash value to find the candidates, so it costs $O(1)$.
- Filter and Rank: it roughy costs O($log(\frac{nn_sL}{2^{m_a}2^M})$).

So totally, the average query time complexity is around $O(dL + log(\frac{n}{2^{m_a}}))$.

Chapter 5

Experiments

To evaluate our similarity search method, we implemented our experiment on a Linux Intel(R) Xeon(R) server(2.2-0GHz, 32.0GB memory). The index is stored in MapDB [14]. MapDB is a pure-Java database engine, and we can easily customize it to achieve our goal due to its clear interfaces and implementation. We inherited the MapDB's storage module and implemented the RDF.

5.1 Datasets

The datasets are used to evaluate our methods is list in Table 5.1. GloVe, SIFT, NYTimes and Fashion-MNIST are all well-known public datasets for ANN search.

GloVe: it is generated by using an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

SIFT: each data point is a SIFT feature which is extracted from Caltech-256 by

Table 5.1: **Datasets**

Dataset	Description	Dimension	Size
GloVe	Global Vectors for Word Representation on tweets [24]	100	1,133,628
SIFT	Image feature vectors [11]	128	1,000,000
NYTimes	Bag of Words Data Set [16]	256	290,000
Fashion-MNIST	Zalando articles' images [35]	784	60,000
CC_WEB_VIDEO	Near-Duplicate Web Video [34]	256	12,870

the open source VLFeat library.

NYTimes: contains five text collections in the form of bags-of-words. After tokenization and removal of stopwords, the vocabulary of unique words was truncated by only keeping words that occurred more than ten times.

Fashion-MNIST: it is a dataset of Zalando's article imagesconsisting of a training set of 60,000 examples. Each example is a 28x28 grayscale image.

CC_WEB_VIDEO: it is divided into 24 categories based on the 24 queries and totally 12,870 videos. In each category, there are different labels to indicate the relation between the labeled video and the query video. Specifically, "E" represents "Exactly the same", "S" represents "Similar" and others represent "Dissimilar". For the ANN search, we build the index for all 24 categories and only use the "S" videos as the queries, the ground truth is also the "S" videos. We apply the method of the DVD(Discriminative Video Descriptor) to extract the feature descriptors of the video. Three descriptors HSV_blue, HSV_green and HSV_red are used as raw features. Each descriptor is a 256-dimension vector.

Then, we normalize and center the dataset and queries(except for the Fashion-MNIST, since the performance is better without center the dataset). Due to the fact that these three datasets don't have the query and ground truth. Thus, we randomly pick 2,000 queries and use the brute-force linear scan [25] approach to get the top 100 ground truth.

5.2 Performance Metrics

We utilize the following measures to evaluate the performance of our method.

- *recall*: *recall* is used to evaluate the accuracy of the return objects, which is widely used in many ANN research work [22, 20]. Given a query q, let $R^* = \{o_1^*, o_2^*, o_3^* \ldots, o_k^*\}$ be the ground truth of top k nearest neighbors with respect to q, our method for ANN search returns k points $R = \{o_1, o_2, o_3 \ldots, o_k\}$. Both results are ranked by the increasing order of their distance to q. Therefore, the *recall* for ANN with repect to q is computed as

$$recall(q) = \frac{|\{o_1^*, o_2^*, o_3^* \ldots, o_k^*\} \bigcap \{o_1, o_2, o_3 \ldots, o_k\}|}{k}$$

  Here, $|set|$ means the number of objects in the set. Since the $|R*| = |R| = k$, the recall actually equals to the precision.

- *candidates percentage(cp)*. For each query, the number of candidiates is different, we use the *cp* to indicates how much the searching range of dataset is reduced by applying our indexing method. The *cp* is defined as follows:

$$cp = \frac{1}{n_q} \sum_{i=1}^{n_q} \frac{|Candidates(q_i)|}{n}$$

  where n is the cardinality objects in dataset, n_q is the number of queries.

- *Average Response Time(ART)*. For each query, the time cost mainly consists of two parts: 1) The searching time in each sub-index to find the closest objects to the query object; 2) The calculating time to verify all objects in candidates set to get top k nearest neighbors. We use the average response time to evaluate

the system time performance. Here, we use t_i to denote the time cost for the i-th query object. The ART is computed as

$$ART = \frac{1}{n_q} \sum_{i=1}^{n_q} t_i$$

5.3 The-state-of-art LSH Index Methods

The several state-of-the-art LSH-based index methods are developed in the last decade. These methods use different hash functions or different index schemes to improve the performance. We implement the basic LSH index [27], where the p-stable hash functions as we discussed in 2.3.2 are originally applied to project the objects.

Thus, if m hash functions are applied, the original object will be described by m integer indices, for example, (8,10,13), which are also known as hash value. As for the index, in order to find the objects that fall into common buckets, a conventional hash is used to map the m-dimensional hash value K into a single linear index by computing

$$T = \big(\sum_{i=1}^{m} H_i K_i\big) \mod P$$

where H_i are integer weights and P is the hash table size. By transferring the K into a fingerprint T, similar objects are most likely to have the same T bucket. Then on retrieval it directly gets the candidates from the exact matching bucket. We call this method as BLSH.

5.3.1 Compare different hash functions

To test the performance of hash functions, four different hash families are used to test the BLSH, the P-Stable and BOA are introduced in Section 2.3. We also test the Sign-random-projection(SRP) [5] and S2JSD-LSH [21] for S2JSD(Square root of two times the Jensen-Shannon Divergence) distance, the S2JSD is defined as:

$$h^{a,b}(p) = \lfloor \frac{\sqrt{\frac{4a \cdot p}{W^2} + 1} - 1}{2} + b \rfloor$$

where b is a real number uniformly taken from 0 to 1, a is a vector with entries chosen independently from the Gaussian distribution $N(0, 1)$.

For testing P-Stable hash functions on SIFT and Fashion-MNIST, we set $w = 4$, $P = 8191$ as a large enough prime number, m and L are two variable parameters. We test different $m = 10, 15, 20$ and gradually increase L with the rise of *recall* and *cp*. The $L = 1, 3, 5, 10, 20, 30, \ldots, 130, 140, 150$ for SIFT and $L = 1, 2, 3, 4, 5, \ldots, 14, 15$ for Fashion-MNIST. As for p-Stable hash functions on NYTimes, we set $w = 2$, $P = 8191$ and $L = 1, 2, 3, 4, 5, \ldots, 14, 15$. We also set $k = 10$, so the algorithms return the top 10 approximate nearest neighbors.

For testing S2JSD on SIFT, we set $W = 0.8$, $P = 8191$ and also use different $m = 10, 15, 20$. $L = 1, 2, 3, 4, 5, \ldots, 19, 20, 30, 40, 50, 60, 70, 80$. For dataset Fashion-MNIST, we set $W = 0.4$, $P = 8191$ and $L = 1, 2, 3, 4, 5, \ldots, 19, 20$. For NYtimes, we set $W = 0.4$, $P = 8191$ and $L = 1, 2, 3, 4, 5, \ldots, 19, 20$.

When we test SRP on NYTimes, we set $L = 1, 2, 3, 4, 5, \ldots, 19, 20$. Also, SRP only need one table for SIFT. We use $L = 1, 2, 3, \ldots, 14, 15$ for BOA on three dataset.

The Figure 5.1 shows the result of four different hash functions on different

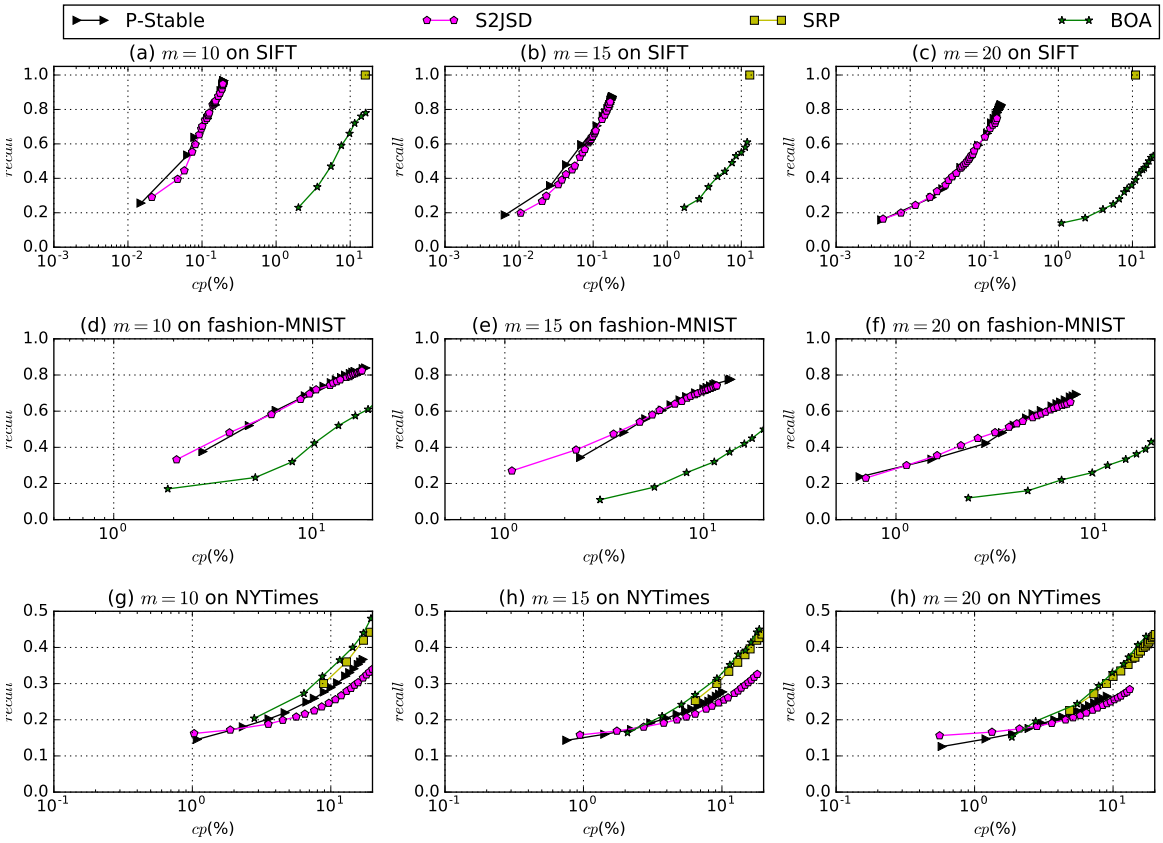

Figure 5.1: Four hash functions comparison

datasets by using BLSH. As we can see from the plots (a),(b),(c), the P-Stable and S2JSD almost have the same performance. For $m = 10, 15$ when L is larger than 10, the P-Stable still a little bit better than the J2JSD. The SRP always has the *recall*=1, it means SRP doesn't have enough capacity to partition the dataset SIFT. The BOA has better ability than SRP to partition the dataset, however, compare to the P-Stable and J2JSD, it uses higher *cp* to achieve the same *recall*. We also find that with the increase of m, P-Stable and J2JSD achieve higher *recall* with the same *cp*, while BOA is the opposite. Overall, for SIFT dataset, P-Stable is a good choice.

In plots (d),(e),(f), we find there is no SRP performance point, because it always retrieves 100% data in our experiment, so SRP cannot be applied to the Fashion-MNIST. BOA can be used for Fashion-MNIST, but the performance is way worse than P-Stable and J2JSD. In addition, J2JSD uses smaller L to achieve the same performance of P-Stable. Thus, J2JSD is suitable for fashion-MNIST. As shown in plots (d),(e),(f), the BOA has the highest *recall* in same *cp*, even though it's only a little bit higher than the SRP. P-Stable is better than the J2JSD, however, both of them are worse than the SRP and BOA.

In conclusion, based on the performance of BLSH index, P-Stable and J2JSD are the choices for dataset SIFT and Fashion-MNIST. BOA and SPR are suitable for NYTimes. The reason is that the dataset SIFT and Fashion-MNIST are image datasets. By applying the machine learning methods, the similarity of features is kept by Euclidean distance. Thus, the P-Stable and J2JSD have better performance. As for dataset NYTimes, it is a text dataset, the similarity of features is preserved by cosine distance, so BOA and SPR have the better performance.

5.4 Parameter Sensitivity of RDF

Overall, not only do we expect our system reduces as much as the possible searching range to response the query faster, but also keeps a high recall of top k nearest neighbor search. To achieve this goal, there are some parameters to be adjusted as list in Table 5.2. L is the number of hash tables, obviously, the recall and space cost rise together with the increment of L. In addition, the requirement of hundreds of hash tables to be created in order to achieve a high recall(e.g. 0.9), which causes the memory overhead [19]. As we introduced in Section 2.3.3, generating a large

Table 5.2: Related Parameters

Parameter	Description
L	Number of hash tables
T_h	The threshold of k-nodes under the same slot
M	The number of BOA functions in Partition LSH layer
$l = \{l_1, l_2, \dots l_{max}\}$	The length of d-node in each level of RDT
n_s	The number of shuffling permutations
Δ	The step from original sub-index to all sub-indexes which need to be looked up

number of probes and multi-index storage scheme are the good choice to conquer it. When implementing the RDF, we combine the multi-probes and multi-index storage scheme together, so only one copy of the whole dataset is kept in the memory. In the following experiment, we only test the RDF on GloVe and NYTimes datasets, due to the fact that BOA hash functions perform well based on our experiment.

5.4.1 The influence of M

As we discussed in Section 3.2, by introducing M BOA hash functions, the index is split into 2^M sub-indexes. The number of sub-indexes determines how much the system support parallelism and M is directly related to this number. For instance, given $M = 2$, the sub-index-ID = 00, 01, 10 and 11. Since the query steps are as follows: 1) calculate the object's hash value in RD LSH layer; 2) apply Partition LSH layer on hash value to get the original sub-index-ID; 3) search all RDTs in Δ-step sub-indexes to get the candidates. Thus, each query only involves in one sub-index when $\Delta = 0$. To compare the impact of different M, we use the GloVe and set the $l = \{32, 32, 32, 32, 32\}$, $n_s = 1$, $T_h = 5000$ and with 0-step search. In each different M, each signal(e.g. star, triangle and so on)the first three experiments have L=1, 5 and 10, then always add 10 to L for latter experiments, and stops when the CP is

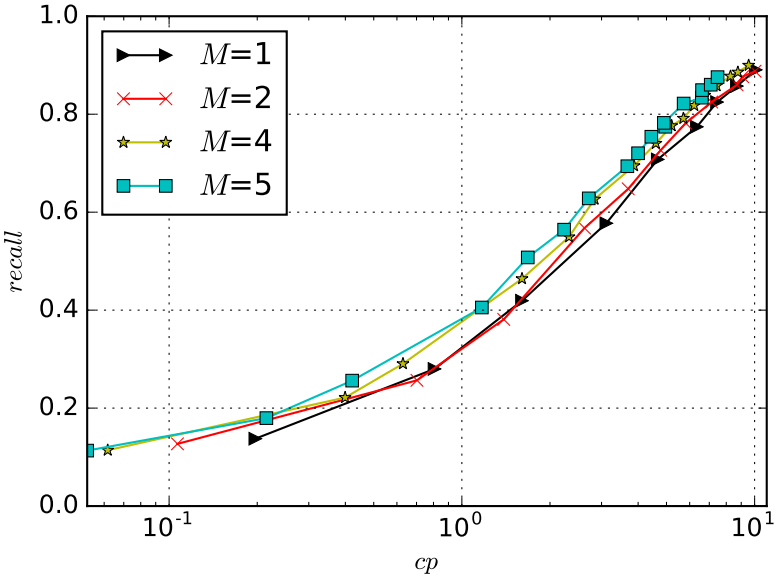

Figure 5.2: *recall* and *cp* with different M

over than 10% or $L > 150$. For each different L, we request 2000 random queries after constructing the index, then calculate the top 10 NN *recall* for these queries. In the Figure 5.2, the larger M we set, the more slowly *recall* rises with the increment of L. Because part of the ground truth is lost when the system applies Partition LSH layer, two hash values are probable to be separated into different sub-indexes even though they are similar to each other. However, if we pursue the query speed, larger M need to be considered. $M = 1$ only supports 2 workers while $M = 4$ serves 16 workers in distributed vision.

5.4.2 The influence of l

Basically, the length of d-node in each level leverages the structure of RDT, as we said in the Section 4.4, the variable l controls the number of bits to locate the objects in

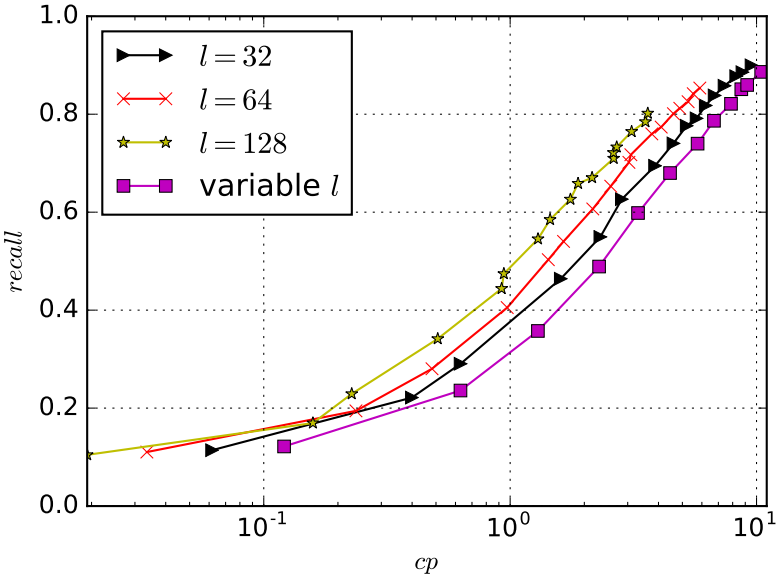

Figure 5.3: *recall* and *cp* with different l

different level of RDT. To test the influence of l, we use GloVe and set $m = 32$, $n_s = 1$, $M = 0$ and $T_h = 5000$, we compare the 4 different l set, which is {32,32,32,32,32,32}, {64,64,64,64,64}, {128,128,128,128} and {4,8,16,32,64,128}. In Figure 5.3, with the same CP, $l = \{128, 128, 128, 128\}$ have the highest *recall* because the more bits of hash value are involved to calculate the slot in each level, the high degree of the data are divided into different slot. The variable $l = \{4, 8, 16, 32, 64, 128\}$ is the fastest one to increase the recall. As we have explained in Section 4.4, the variable l captures more candidates in former levels, it helps the small number similar objects group gain enough efficient candidates.

5.4.3 The threshold of objects under the same slot

The different number of threshold T_h affects the actual level under each slot. The level is small with large T_h. In Section 4.4, we analyze the relationship between the n, l and T_h. However, in practice, the distribution of feature data in the cloud application is not perfectly uniform. To find the influence of T_h, we test it on GloVe dataset. Given the fixed parameters: $l = \{32, 32, 32, 32, 32, 32\}$, $n_s = 1$, $M = 1$. As for the steps of L, we follow the same strategy as used to find the influence of M. From the Figure 5.4, with the increase of T_h, it is easier to get a higher recall with less L, in that there are more objects under the same slot in RDT, also, the actual level of RDT becomes smaller. However, with the same CP, we find that the smaller T_h has higher *recall*. The number of objects under the same slot decreases, which causes a large number of similar groups must be divided due to higher resolution in the deeper level(where the objects are redistributed into different slots in a new d-node). It also leads the average level of objects in RDT increases, thus the building time also extends. In addition, with smaller T_h, it is difficult to increase the recall to a high level(etc. over 0.9) even we increase L to 150. And when we increase T_h to 7000, there is no major difference compared to $T_h = 5000$, it means that T_h has reached saturation.

5.4.4 Δ-step search

In Section 3.3, the Figure 3.2 shows the top k objects' data percentage of Δ-step indexes cover. Even though the similar objects are still likely to be divided into different sub-indexes, the original(0-step) sub-index still keeps the highest rate(around 90%) of top k similar objects. The 1-step sub-indexes keep the second highest rate. To test the different Δ-steps search performance, we use the dataset GloVe and measure

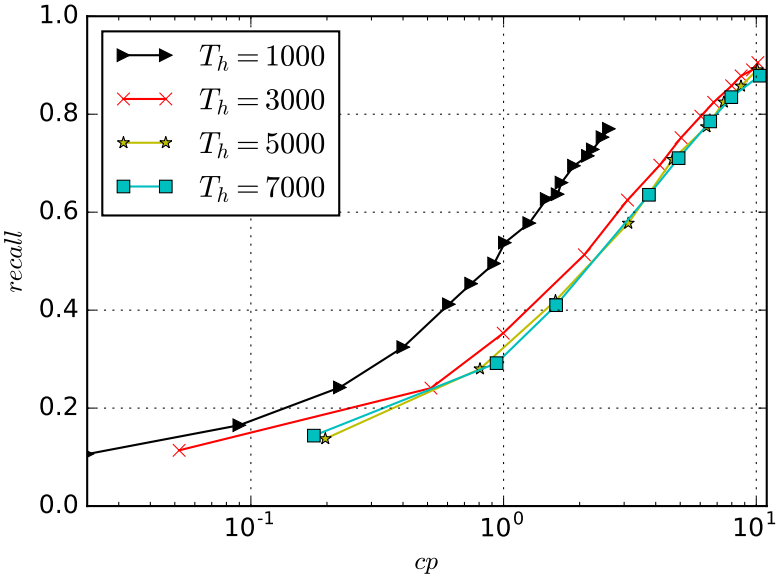

Figure 5.4: *recall* and *cp* with different T_h

it by using *recall* and *ART*. Given the parameters $L = 20$, $n_s = 3$, $T_h = 5000$ and $l = \{32, 32, 32, 32, 32, 32\}$, because the $\Delta_{max} = M$, we conduct experiment on different M, and evaluate all possible Δ-step searches. As we can see from the Table 5.3, the 0-step search always has the shortest *ART* with lowest *recall*. It is because the 0-step search only involves one sub-index, which only contains around $100/2^M\%$ data, as shown in Table 3.1. With the increment of Δ, the *recall* rises because more sub-indexes are searched, which also costs more query time. In addition, we find that only searching the 1-step sub-indexes increases the *recall* a lot and the others not. It is because the other Δ-step sub-indexes don't contain much of the similar objects, which also proves the Partition LSH layer works well.

Table 5.3: Δ-step search performance with different M on GloVe

	M=2			M=3				M=4				
Δ	0	1	2	0	1	2	3	0	1	2	3	4
recall	0.78	0.87	0.88	0.73	0.86	0.88	0.89	0.68	**0.85**	0.89	0.90	0.90
ART(ms)	41.6	67.9	70.5	33.1	62.8	76.5	81.8	28.2	**43.2**	93.4	117.4	127.4

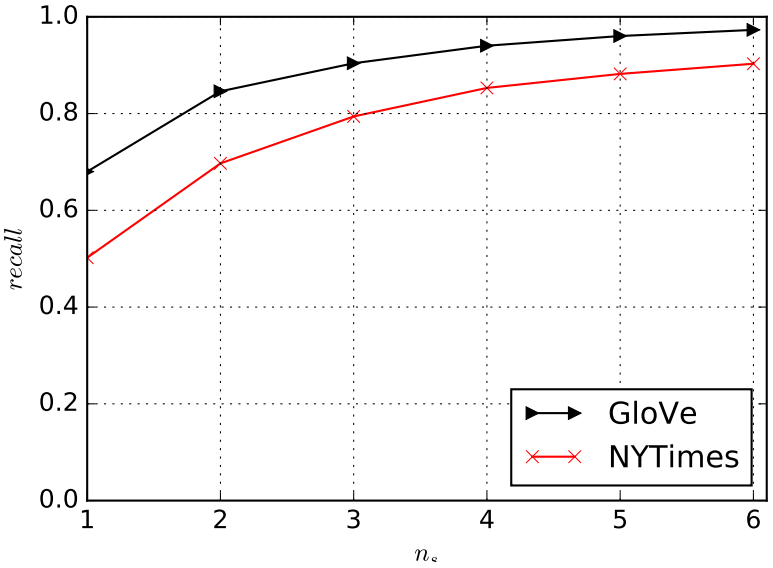

Figure 5.5: *recall* with different n_s

5.4.5 Number of shuffling permutations

The number of shuffling permutations effect the real number of RDTs in the RDF. As depicted in Figure 4.4, by increasing n_s, P_s go down more smoothly, which results in capturing more similar objects in the results. However, larger n_s also means creating more RDTs. The *ART* definitely rises because of traversing more RDTs to find the candidates. We test the influence of n_s on GloVe and NYTimes dataset. We generate 2000 random queries and set $L = 20$, $l = \{32, 32, 32, 32, 32, 32\}$, $M = 2$, $\Delta = 0$ and $T_h = 5000$. In the Figure 5.5, with the rise of n_s, the *recall* increases. However, due to the fact that LSH algorithm is based on probability. Thus, when $n_s > 3$, the

recall doesn't have large difference. So the $n_s = 3$ is a fair choice to keep both a high *recall*(over 0.9) and a relative low number of RDTs.

5.5 Comparison with Other LSH Methods

The FALCONN [1], LSHForest [3], PHF [37] are used to compare the performance with our method on CC_WEB_VIDEO dataset. For CC_WEB_VIDEO, we use three different features: HSV_blue, HSV_green and HSV_red. Without loss of generality, the experiments stop until the recall doesn't increase anymore. In the Table 5.4, for the single feature, the HSV_red is the better than other two features to find the similar objects. The combined feature means using three of them to get the better results, while it costs more time through query more indexes. In terms of query speed, FALCONN is the fastest method to get the query results. LSHForest costs more time than other three methods. PHF and RDF have the almost same query speed, a little slower than FALCONN. As for the recall, FALCONN is unstable, with the combined features, the highest recall can be 0.997, the lowest recall is 0.418, while the average is the worst one in these four methods. By conquering the MSB problem, our method RDF is superior to the other three methods with the average recall 0.744 by using combined features. Due to the fact that part of similar objects are missing because of MSB problem. By applying n_s shuffling permutations on original hash values, each set of twisted hash values are used to construct the different shape of RDTs. More similar objects can be preserved to improve the overall performance than other three methods.

Table 5.4: The recall and ART on CC_WEB_VIDEO

Methods	Feature	highest recall	lowest recall	average recall	ART(ms)
LSHForest	HSV_blue	0.939	0.248	0.536	7.886
	HSV_green	0.917	0.258	0.518	6.871
	HSV_red	0.918	0.305	0.584	8.176
	combine feature	0.976	0.421	0.690	19.816
FALCONN	HSV_blue	0.949	0.25	0.539	2.970
	HSV_green	0.917	0.261	0.526	3.014
	HSV_red	0.939	0.312	0.587	**2.853**
	combine feature	**0.997**	0.418	0.661	7.61
PHF	HSV_blue	0.918	0.175	0.495	3.498
	HSV_green	0.960	0.217	0.524	3.498
	HSV_red	0.960	0.216	0.514	3.470
	combine feature	0.984	0.389	0.670	10.126
RDF	HSV_blue	0.960	0.286	0.594	3.418
	HSV_green	0.960	0.303	0.619	3.508
	HSV_red	0.971	0.310	0.629	3.509
	combine feature	0.980	0.486	**0.744**	10.379

Chapter 6

Conclusions and Future Work

In this thesis, we present a LSH-based distributed index called Random Draw Forest(RDF), to achieve the efficient and high-quality similarity search over large-scale multimedia data. We use the multiple shuffling permutations to avoid the MSB problem in the traditional index structure. The structure of RDT is adaptive for different dataset with different distribution or magnitude. In the meanwhile, to reduce the search range for large-scale data, the Layered Hash provides the good solution for distribution of large-scale data. To gain the mistakenly partitioned similar objects in other sub-indexes, we design a Δ-step sub-indexes lookup strategy. We also combine the multi-probes and multi-index storage to overcome index storage overhead. The comprehensive experiments show the Layered Hash has the stable data distribution and RDF outperforms the other LSH-based state-of-the-art algorithms.

In the future, there are four points in LSH and similarity search can be optimized:

- The way to pick up the adaptive hash functions for certain dataset by considering data distribution. Sun presented an approach to explore and exploit the data distribution by using principal component analysis(PCA) first, then adjust

the hash functions which are most suitable for the data [28].

- Index structure improvement. For example, by optimizing the step to redistribute the objects in RDF, which improves the speed of index construction.
- Learn the intrinsic dimensionality of data, which is usually much lower than the appearing dimensionality.
- The way to efficiently filter the candidates, approximation is acceptable.

We envision that more fruitful directions to solve the similarity search problem based on practical large-scale dataset in the future.

Bibliography

[1] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal lsh for angular distance. In *Advances in Neural Information Processing Systems*, pages 1225–1233, 2015.

[2] B. Bahmani, A. Goel, and R. Shinde. Efficient distributed locality sensitive hashing. In *Proceedings of CIKM '12*, CIKM '12, 2012.

[3] M. Bawa, T. Condie, and P. Ganesan. Lsh forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, pages 651–660. ACM, 2005.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[5] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[6] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of SCG '04*.

[7] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 541–552. ACM, 2012.

[8] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6):1115–1145, 1995.

[9] E. C. Gonzalez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(9):1647–1658, 2008.

[10] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[11] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

[12] J. Ji, S. Yan, J. Li, G. Gao, Q. Tian, and B. Zhang. Batch-orthogonal locality-sensitive hashing for angular similarity. *IEEE transactions on pattern analysis and machine intelligence*, 36(10):1963–1974, 2014.

[13] A. Khwileh, D. Ganguly, and G. J. Jones. Utilisation of metadata fields and query expansion in cross-lingual search of user-generated internet video. *Journal of Artificial Intelligence Research*, 55:249–281, 2016.

[14] J. Kotek. Mapdb. http://www.mapdb.org/.

[15] I. Laptev, M. Marszalek, C. Schmid, and B. Rozenfeld. Learning realistic human actions from movies. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[16] M. Lichman. UCI machine learning repository, 2013.

[17] 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 Endow.*, 7(9):745–756, May 2014.

[18] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[19] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of VLDB 2007*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[20] Y. A. Malkov and D. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*, 2016.

[21] X. Mao, B.-S. Feng, Y.-J. Hao, L. Nie, H. Huang, and G. Wen. S2jsd-lsh: A locality-sensitive hashing schema for probability distributions. In *AAAI*, pages 3244–3251, 2017.

[22] B. Naidan, L. Boytsov, and E. Nyberg. Permutation search methods are efficient, yet faster search is possible. *Proceedings of the VLDB Endowment*, 8(12):1618–1629, 2015.

[23] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1186–1195. Society for Industrial and Applied Mathematics, 2006.

[24] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

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

[26] P. Scovanner, S. Ali, and M. Shah. A 3-dimensional sift descriptor and its application to action recognition. In *Proceedings of the 15th ACM international conference on Multimedia*, pages 357–360. ACM, 2007.

[27] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal processing magazine*, 25(2):128–131, 2008.

[28] Y. Sun, Y. Hua, X. Liu, S. Cao, and P. Zuo. Dlsh: a distribution-aware lsh scheme for approximate nearest neighbor query in cloud computing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 242–255. ACM, 2017.

[29] 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 Endow.*, 6(14):1930–1941, Sept. 2013.

[30] 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. *Proceedings of the VLDB Endowment*, 6(14):1930–1941, 2013.

[31] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of SIGMOD 2009*, SIGMOD '09, pages 563–576. ACM, 2009.

[32] E. Tuncel, H. Ferhatosmanoglu, and K. Rose. Vq-index: An index structure for similarity searching in multimedia databases. In *Proceedings of the tenth ACM international conference on Multimedia*, pages 543–552. ACM, 2002.

[33] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

[34] X. Wu, A. G. Hauptmann, and C.-W. Ngo. Practical elimination of near-duplicates from web video search. In *Proceedings of the 15th ACM international conference on Multimedia*, pages 218–227. ACM, 2007.

[35] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[36] Yury, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[37] N. Zhu, Y. Lu, W. He, and Y. Hua. A content-based indexing scheme for large-scale unstructured data. In *Multimedia Big Data (BigMM), 2017 IEEE Third International Conference on*, pages 205–212. IEEE, 2017.