Dynamic Partition Forest: An Efficient and Distributed Indexing Scheme for Similarity Search based on Hashing

Yangdi Lu[†], Yang Bo[†], Wenbo He[†] and Amir Nabatchian[‡]
[†]*Department of Computing and Software*
McMaster University, Hamilton, Canada
Email: {luy100, boy2, hew11}@mcmaster.ca
[‡]*Huawei Canada, Toronto, Canada*
Email: amir.nabatchian@huawei.com

***Abstract*—The similarity search over large-scale feature-rich data(e.g. image, video or text) is a fundamental problem and has become increasingly important in data mining research. Hashing based methods, especially Locality Sensitive Hashing(LSH), have been widely used for fast Approximate Nearest Neighbor search(ANNs). Various hash families, distribution schemes and space efficient search strategies have been proposed during the last decade. However, there are still two flaws in existing methods: (1) The state-of-the-art distribution scheme sacrificed too much accuracy for speeding up the query in practice. (2) Most LSH-based index approaches directly used the static number of compound hash values without considering the data distribution, resulting in system performance degradation. In this paper, a distribution-aware index structure called *Dynamic Partition Forest*(DPF) is designed to dynamize the used bits of hash values, which leads itself to auto-adapt various data distributions and incremental data. To improve the performance, a multiple-step search strategy is integrated with DPF to mitigate the accuracy loss with distributed scheme. The experiment results show that DPF increases the accuracy by 3% to 5% within the same timeframe compared to DPF without multiple-step search. Additionally, DPF with our partition scheme is 1.4 times faster than DPF without partition, which demonstrates the efficiency of our content-based distributed scheme. Experimental comparisons with other two state-of-the-art methods on three popular datasets show that DPF is 3.2 to 9 times faster to achieve the same accuracy with 17% to 78% decrease of index space.**

***Keywords*-content-based retrieval, distributed search strategy, index structure**

I. Introduction

The volume, velocity, variety, value and veracity of data make it very challenging to conduct big data analysis. Similarity search is the core technology to many applications(e.g. content-based image retrieval [1], recommendation system [2], semantic document retrieval [3]). Generally, data objects are represented as high dimensional points and similarity search problem is usually formulated as the k nearest neighbor search(KNNs) problem: given a set of query points Q and a set of data points D, it returns k closest points of D to each query $q \in Q$ under the distance function $d(\cdot, \cdot)$. A straightforward way for KNNs is linearly comparing the query with each point in D. The complexity of this approach is $O(dn)$, where d is the dimension and n is the number of points in D. This approach is clearly not applicable to large-scale data with high dimensionality. Precursive methods like KD tree [4] and Ball tree [5], perform well in low dimensionality(i.e. less than 20). When the dimensionality is over 20, these tree-based methods all suffer from the curse of dimensionality [6], where query performance declines exponentially with the increasing number of dimensionality.

To break this curse, the approximate nearest neighbor search(ANNs) has been proposed, which aims to increase efficiency by sacrificing accuracy. Various classes of algorithms are proposed to solve the ANNs problem. One of them is Permutation Indexes(PI) [7], which is an efficient algorithm to predict the similarity between objects. By introducing a distinguished set of pivots, each object can be represented by a permutation defined by the ranked distance towards these pivots. Thus, the distance between the objects now is described by the distance between their permutations. However, PI is a non-distributed algorithm [8] because the distance calculation to all the pivots has to be computed at once. Another algorithm is Navigable Small World Graphs(NSWG) [8]. Its basic concept resides on "The neighbor of my neighbor is also to be my neighbor". The transitive relation between any two nodes on a navigable small world network is of polylogarithmic time complexity, which makes it suitable for ANNs problem. The restriction of NSWG is that the query points should be inserted in the graph before search, which is, however, a resource-consuming operation. Besides these two methods, hashing-based methods like Locality Sensitive Hashing(LSH) [9], are also good candidates. The basic idea of LSH is using the distance-preserving hash functions to project the high dimensional object into a hash code space, and similar points in the original feature space should be hashed into close points in the hashcode space, results in pruning the search space to accelerate the query speed.

Over the last decade, hashing-based index methods have been improved in the following aspects: (1) Distributed Design: to make the methods applicable for large-scale data, the state-of-the-art distribution scheme called Distributed LSH

[10] uses layered-LSH to content-based partition the data into a large number of small partitions. However, in practice, we find that the accuracy decreases dramatically with the number of partitions increasing. A very detailed explanation of this phenomenon is presented in Section II-C. (2) Index Structure: E2LSH [9], [11] uses a conventional hash to map the hash values into linear index. Locality Sensitive B-tree(LSB) [12] transfers the hash values into Z-order values to build a B-tree structure index. LSH Forest [13] is a tree-based generation of LSH, where heavily load hash buckets are recursively partitioned with embedded hash tables. SKLSH [14] defines a new distance measure of hash values, and sort them to build a B^{+}-tree. However, none of these index structures uses $dynamic$ number of hash functions and considers data distribution with incremental data, resulting in performance degradation. (3) Space Efficiency: Multi-probes LSH [15] is proposed to generate high quality "perturbed" query based on the hash values, which not only reduces the memory usage of index, but also achieves high accuracy. (4) Hash Families: P-stable hash family [9] is designed for ℓ_p norm, where $p \in (0, 2]$. Sign Random Projection(SRP) hash family [16] is used for cosine distance. Orthogonal hash family [17] improves the SRP by using the orthogonal basis to do the projection. (5) Data dependent hashing: the hash functions are learned from data, like Isotropic Hash [18], it is try to learn an orthogonal matrix to rotate the principal component analyis(PCA) projection matrix which has the equal variances for different dimensions.

Owning to the difficulty to pre-analyze the distribution of incremental data, almost all of LSH-based methods have applied the static number of compound hash functions. However, static number of compound hash functions is likely to impact the performance if the data is not uniformly distributed. As shown in Figure 1, the data has two main clusters: cluster 1 and cluster 2. The points in cluster 1 is denser than cluster 2. The dashed line is hyperplane h, which represents a hash function. For querying the top 5 nearest neighbors of the green point in cluster 1, it is crucial to add the hyperplane h, because high resolution is required to exclude irrelevant red points. However, for querying the top 5 nearest neighbors of green point in cluster 2, the hyperplane h is not necessary because it does not need hyperplane h to differentiate good candidates(i.e. red points in cluster 2). Reflected to LSH, whether we can use the *dynamic* number of hash functions to distinguish unpredictable distribution of incremental data is important to improve the system performance. Here, we present a summary of our contributions in this paper:

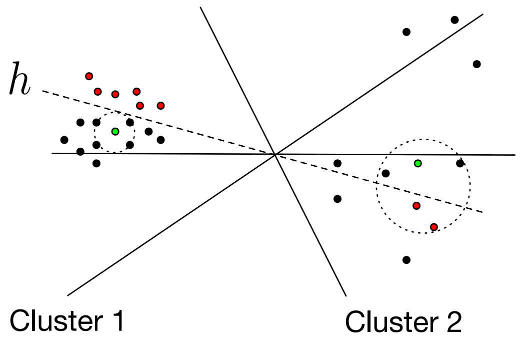


Figure 1: The dilemma of static number of compound hash functions.

1. **Mitigate the Accuracy Loss after Distributed LSH.** We find the accuracy loss after implementing distributed LSH scheme. By exploring the ground truth distribution over different partitions, we discover that only part of partitions is likely to contain the mistakenly partitioned similar objects to query. Thus, we design a hierarchical Δ-step search strategy on partition search with detailed proof to mitigate the accuracy loss in minimum time. The results show that 1-step search strategy with a large number of partitions can significantly improve the accuracy of ANNs.
2. **Dynamic Bits Extension with Collision Overflow Design.** We take the data distribution into account and design an index structure for binary hash code called *Dynamic Partition Forest*(DPF). The threshold overflow design helps DPF automatically change the involved bits of hash values in constructing it in a suitable way for incremental data. With this design, DPF intelligently eliminates irrelevant points and includes good candidates to improve the system performance.
3. **Practical Implementation and Comprehensive Experiments.** We have implemented our method to evaluate the efficiency of the hierarchical Δ-step search strategy and DPF. Three widely used datasets are utilized to examine the practical performance of DPF. The results demonstrate 3% to 5% accuracy improvements through 1-step search strategy. To be more convincing, our system is 3.2 to 9 times faster to achieve the same accuracy compared with two state-of-the-art methods, E2LSH [9] and LSH Forest [13]. What's more, the space overhead of DPF is reduced by 17% to 78%.

The rest of this paper is organized as follows. The quantization, partition step and hierarchical Δ-step search strategy are described in Section II. Section III presents the approach to construct the DPF and search algorithm in DPF. The experiments are illustrated in Section IV. Finally, we conclude the paper in Section V.

II. Quantization and Partition

Some frequently used notations in this paper are given in Table I. The feature-rich data are generally high-dimensional vectors. As shown in Figure 2, given a set of data objects, the first component of our method is to quantize these objects

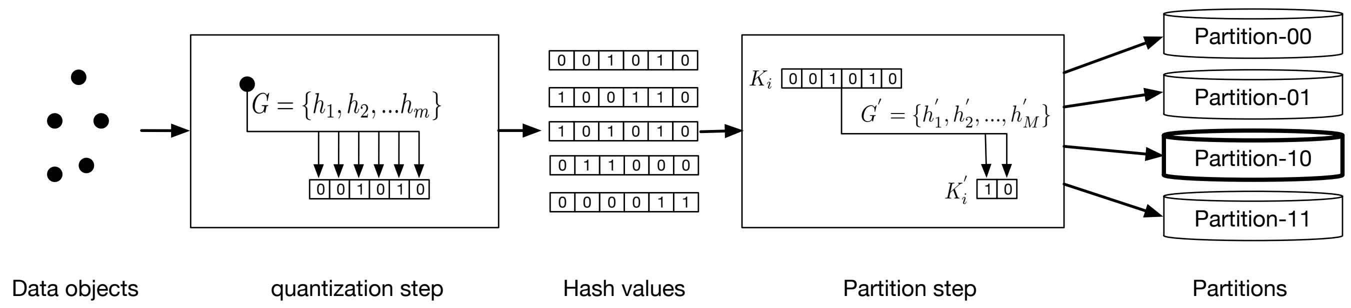


Figure 2: The flow of quantization and partition process. $m = 6$, $M = 2$ and $2^M = 4$ partitions for distribution.

Table I: Summary of Notations

Notation	Description
$D = [p_1, ..., p_n]$	Dataset consists of n d-dimensional objects
$Q = [q_1, ...q_u]$	Query dataset
L	Number of hash tables
F_s	Size of hash family
m	Maximum number of hash functions
$G(\cdot) = [h_1(\cdot), ..., h_m(\cdot)]$	m hash functions in quantization step
$K = \langle K_1, K_2, ...K_n \rangle$	Hash values (i.e. K_1=$G(p_1)$)
$G^{'}(\cdot) = [h_1^{'}(\cdot), ..., h_M^{'}(\cdot)]$	M hash functions in partition step
$K^{'} = \langle K_1^{'}, K_2^{'}, ...K_n^{'} \rangle$	Partition ID (i.e. $K_1^{'}$=$G^{'}(K_1)$)
Δ	The Hamming distance of partition-IDs.
$l = \{l_1, l_2, l_3, \ldots, l_{max}\}$	The length of d-node in each level
$T = \{T_1, T_2, \ldots, T_{max}\}$	The threshold of k-nodes in each level

into binary hash values. Based on the hash values, the second component is a content-based partition approach to make the system distributed for large-scale data. The quantization step for generating hash functions is described in detail as follows.

A. Generating Hash Functions for Quantization

As shown in *Definition 1*, LSH has the property that close objects in high-dimensional space will collide with a higher possibility than distant ones.

Definition 1: ***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, both $c > 1$ and $\mathcal{P}_1 > \mathcal{P}_2$ hold. To define the maximum distinguishing capacity, 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 hash functions from a designed hash family. Specifically, the **compound hash value** of a point p_i under G is $K_i = G(p_i) = [h_1(p_i), h_2(p_i), \ldots, h_m(p_i)]$. For simplicity, we call the compound hash values as hash values in the rest of the paper.

The hash functions G is the key in quantization step. We aim to map the $D = [p_1, p_2, \ldots, p_n]^T \in R^{n \times d}$ to a hashcode space to get the compact representations $K = [K_1, K_2, \ldots, K_n]^T \in R^{n \times m}$. We denote $G = [h_1, h_2, \ldots, h_m] \in R^{d \times m}$. In this paper, we first initialize the orthogonal angle hash family, as shown in *Definition 2*.

Definition 2: ***Orthogonal angle hash family***. 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 function $sign(z) = 1$ if $z \geq 0$ and 0 otherwise. It means our method uses one bit to quantize each projected dimension. When m hash functions are used, our method actually projects the original feature space into 2^m parts. More specifically, $K_i = \underbrace{01001\ldots001}_{m}$. The way to generate orthogonal angle hash functions for each hash table is data independent, which causes hashing-based method requires numerous hash tables for high accuracy in practice. To minimize quantization loss as much as possible, we find that the closer $sign(z)$ and z are, the better the locality property of the projected data will be preserved [19], [20], which can be formulated as the following optimization problem:

$$\min_{z_i} \sum_{i=1}^{n} \| sign(z_i) - z_i \|$$

Our method uses an iterative optimization method proposed in [19], [21]. Algorithm 1 shows the detailed quantization step for one hash table. However, the optimization method requires a pair-wised similarity matrix of the data. It consumes both computation overhead and memory space overhead when the data size is large(e.g. 1M dataset need 1M× 1M similarity matrix). Thus, in the experiment, we only apply the optimization method for Fashion-MNIST [22] dataset.

B. Distributed through Content-based Partition

For large-scale data, there are two challenges need to be considered. (1) Single machine doesn't have enough memory space for the whole index. Thus, a distributed scheme is required to partition our data into smaller partitions. However, if we directly divide the data into partitions like [23], the system needs to parallelly search all partitions

Algorithm 1: Quantization(d, F_s, I_{num}, m, D)

```
Input: Dimension $d$; Hash family size $F_s (F_s \leq d)$; Number
       of Iteration $I_{num}$; Number of hash functions $m$; Data
       set $D$;
Output: Hash values $K$;
1  $O = \emptyset$, $G = \emptyset$, $K = \emptyset$;
2  Generate a random matrix $H$ with each element $x$ being
     sampled independently from the normal distribution
     $\mathcal{N}(0, 1)$. Denote $H = [x_{i,j}]_{d \times d}$;
3  Compute the $QR$ decomposition of $H$, such that $H = Q \cdot R$;
4  Get the first $F_s$ column of Q into $O$ as orthogonal hash
     family;
5  Randomly pick $m$ hash functions from $O$ to $G$;
6  Calculate the pair-wised similarity matrix $A$
       $A_{i,j} = \begin{cases} 1 & \text{if } p_i \in N_k(p_j) \text{ or } p_j \in N_k(p_i) \\ 0 & \text{otherwise} \end{cases}$
     where $N_k(p)$ denotes the $k$-nearest neighbors of $p$;
7  Optimize the hash functions $G$ as follows;
8  $F = diag(A \cdot I)$;
9  $L = F - A$;
10 for $i = 1; i \leq I_{num}$ do
11     $K = sign(D \cdot G)$;
12     $\mathbb{G} = D^T \cdot L \cdot D \cdot G + D^T \cdot D \cdot G - D^T \cdot K$;
13     $\mathbb{M} = \mathbb{G} \cdot G^T - G \cdot \mathbb{G}^T$;
14     $Q = (I + \frac{1}{2}\mathbb{M})^{-1}(I - \frac{1}{2}\mathbb{M})$;
15     $G = Q \cdot G$;
16 $K = sign(D \cdot G)$;
17 return $K$;
```

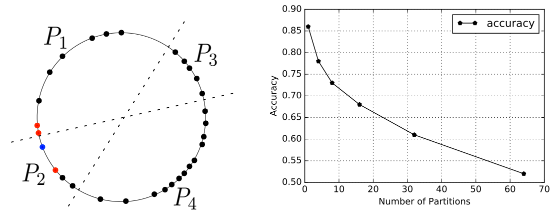
for each query, which involves network overhead. (2) Another challenge is how to make sure that each query only needs to access a single machine. It means the objects are required to be similar with each other within a partition. Our content-based partition strategy used an idea which is similar to Layered LSH [10], the difference is we use the orthogonal hash family, while Layered LSH uses the P-stable hash family. Specifically, another set of hash functions $G^{'} = [h_1^{'}, h_2^{'}, \ldots, h_M^{'}] \in R^{m \times M}$ are used. As we can see in Fig. 2, for each hash value K_i, the result $K_i^{'}$ is an M long binary hash value, which indicates the partition-ID that the object belongs to. The principle of our content-based partition approach resides here:

1) Similar objects have high possibility to have the similar hash values after quantization.
2) Similar hash values have high possibility to have the similar hash values(partition-ID).

Algorithm 2 illustrates the content-based partition step for one hash table. Through this, there is no overhead network delay between the work nodes of the system, which improves the network efficiency. The parameters M influences the concurrency handling capacity of the system. Specifically, 2^M partitions are generated.

Algorithm 2: Partition(K, $G^{'}$)

```
Input: Hash values $K$; Hash functions $G^{'}$;
Output: Each objects' partition ID $K^{'}$;
1  $K^{'} = \emptyset$;
2  $Y = sign(K \cdot G^{'})$ /*Now $Y$ is a $n$ by $M$ matrix*/ ;
3  for $i = 1; i \leq Y.length$ do
4      $pID = 0$;
5      for $j = 1; j \leq Y[i].length$ do
6          $pID = (pID << 1) \mid Y[i, j]$;
7      $K^{'}[i] = pID$;
8  return $K^{'}$;
```



(a) 2-d hashcode example (b) Accuracy loss

Figure 3: An example of accuracy loss after distributed LSH scheme in 2-d hashcode space and the accuracy loss plot with the increasing number of partitions. Blue point is the query and red points represent good candidates. P_1, P_2, P_3, P_4 are four partitions.

C. Mitigate Accuracy Loss by Δ-step Search Strategy

The ideal partition strategy is dividing all the similar objects into one partition. However, due to the approximate property of LSH, we find the similar objects are still likely to be divided into different partitions, which results in performance degradation. As shown in Fig. 3(a), after applying the partition step, the query point falls into P_2. The traditional search strategy only retrieves the similar objects in P_2, results in the loss of good candidates in P_1. In Fig. 3(b), after implementing the partition strategy in practice, the accuracy dramatically declines with the increment of partitions.

To mitigate the accuracy loss, we need to find out where the lost good candidates located. In another word, the search strategy should explore more "valuable" partitions, which are most likely to contain the lost good candidates of the query. Thus, we design a Δ-step search approach based on another LSH property: the partitions that are one step away are most likely to contain objects that are close to the query object than partitions that are two steps away. There are only two possible values 0 or 1 in each bit of a partition-ID. The Hamming distance between two partition-IDs is denoted as Δ, and $\Delta_{max} = M$. To prove that the Δ has an unbiased estimate of the similarity between the corresponding hash values, we have *Lemma 1* [24](*Lemma* 3.2) as follows.

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

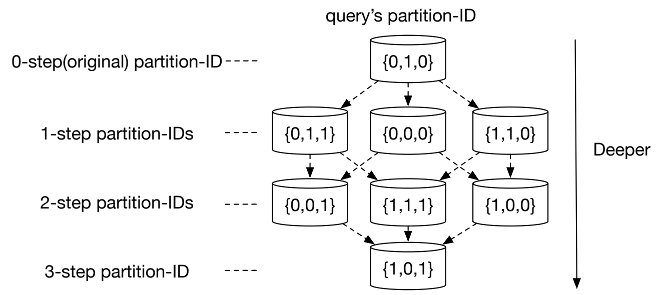


Figure 4: Different Δ-step partition-ID of query with $M = 3$. Each dotted line represents one step.

from S^{d-1}, given $h_r(v) = sign(v \cdot r)$, we have

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

where $\theta_{v_i,v_j} = cos^{-1}(\frac{\langle v_i, v_j \rangle}{\|v_i\|\|v_j\|})$. It reveals the relation between Hamming distance and angular similarity. Through the former proof in [17], we further have

Theorem 1: Given M orthorgonal vectors $h_1, h_2, \dots, h_M$ from the orthogonal angle hash family, then for any two normalized binary vectors $p, q \in S^{m-1}$ after quantization step, by defining M indicator random variables $X_1^{p,q}$, $X_2^{p,q}$, $\dots$, $X_M^{p,q}$ 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 $C = M/\pi$. It explains smaller Δ-step partitions are more likely to contain the hash values close to the query's hash value. To hierarchically generate the Δ-step partitions, we apply +1(for bit=0) or -1(for bit=1) on the Δ number of bits in original partition-ID. The total number of Δ-step partitions is $\binom{M}{\Delta}$. As the example shown in Figure 4, suppose $M = 3$, the original sub-index-ID is 010, then the 1-step partition-IDs are 110, 000, 011, the 2-step partition-IDs are 100, 111, 001, the 3-step partition-IDs is 101. The *Theorem 1* reveals larger Δ results in more dissimilar objects. Thus, our hierarchical Δ-step search strategy is to search the original(0-step) partition first, if the accuracy is low, then search the 1-step partitions to increase the accuracy. The comprehensive evaluation of content-based partition strategy and Δ-step search strategy is in Section IV.

III. Distribution-aware Index

After quantization and partition steps, each partition consists of a sufficient number of similar objects. An efficient index structure is required to facilitate the query speed. However, as we described in Section I, directly using the static number of bits(hyperplanes) to build index will cause performance degradation. Our idea is to design an index structure which can detect high collision area and then dynamically use adequate bits of hash values to hierarchically divide the area. In practice, we set a objects threshold to each sub-space. When detecting objects overflow, our system then includes more bits(hyperplanes) to divide the sub-space into more smaller sub-space.

A. Construction of Dynamic Partition Forest

To meet our requirements, we design a distribution-aware index structure called *dynamic partition forest*(DPF), which consists of multiple *dynamic partition trees*(DPT). The structure of DPT is similar to the R-tree which is formed by hierarchically partitioning the hash values in each sub-index. The difference is when the tree level goes deeper, the more bits of hash values are evaluated to determine the position, which is inspired by [25].

We introduce two types of nodes in DPT: (1) k-node: contains two fields KEY and $POINT$, KEY is the objectID, 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. The value in each slot saves the reference to the first k-node in the slot or the first d-node in the slot. In addition, we set a threshold array T for each level to indicate the maximum number of similar objects under each slot. The detail steps of inserting a hash value $\hbar$ from K into a DPT are explained as follows:

- **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 position 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 DPT.
- **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 hash value is $\hbar$ in storage space and terminate the insert process. The *Insert k1* and *Insert k4* in Figure 5 shows this step.
- **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 $T[level]$, we insert

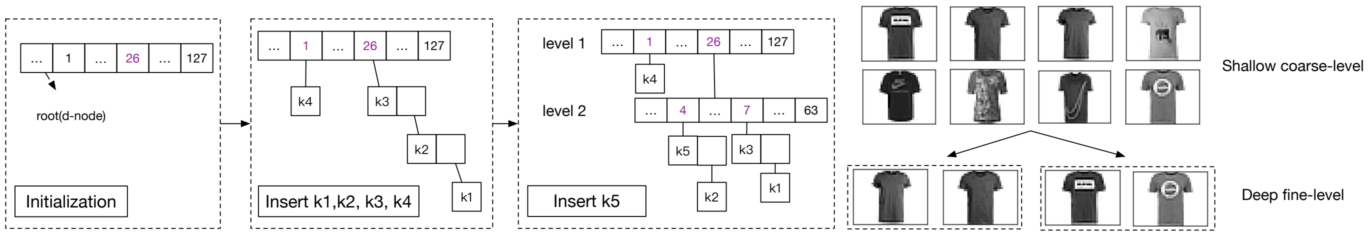


Figure 5: Example of DPT construction. Here, T = {3, 2, ...} and $l = \{128, 64, \ldots\}$. The decimal representation of hash values are k1 = {26, 7, ...}, k2 = {26, 4, ...}, k3 = {26, 7 , ...}, k4 = {1, 4, ...}, k5 = {26, 4, ...}. In left part, shirts are more similar in deep fine-level than the shallow coarse level.

$\hbar$ under this slot, then terminate, as *Insert k2* and *Insert k3* shown in Figure 5. If the number of objects under this slot is larger than $T[level]$, 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 5. If the number of objects in the new d-node under one slot is still larger than $T[level]$, we do **Step 5** repeatedly until less than $T[level]$ or reaching to the max level, then terminate. So, it means at the max level, there is no T limitation.

By following the above steps, the pseudocode of DPF construction for one hash table is shown in Algorithm 3. Each hash table contains 2^M DPT in total to make up the DPF. Compared with traditional index structure, the advantages of DPF reside in three points: (1) The length of used bits to build index is dependent on data distribution. When the collision of similar objects overflows in a slot, DPF will hierarchically partition similar objects into next levels to achieve higher differentiation ability. As depicted in the left part of Figure 5, images are classified from shallow coarse-level to deep fine-level with increasing differentiation. (2) The threshold T determines the discrimination ability of DPT. A large threshold in deep level enlarges the search range which requires more candidates to be evaluated, leading to computation overhead. A small threshold in shallow level may exclude too many objects while it increases the false negative rate. By decreasing the threshold from shallow level to deep level, our method not only decreases the computation overhead but also reduces the false negative rate. (3) The parameter l has the opposite effect against T. By setting l variable in different level, DPT captures enough good candidates even for incremental data.

Algorithm 3: Index(K, $K^{'}$, l, T)

```
Input: Hash values $K$, Partition-ID $K^{'}$, Length of d-node in
       each level $l = l_1, l_2, \ldots, l_{max}$, Threshold array $T$
Output: Index I
I = initialize $2^M$ number of empty DPT;
max = $l.length$;
mask = Array($l_1 - 1$, $l_2 - 1$,..., $l_{max} - 1$);
for $i = 0; i < K.length$ do
    curDPT = I[$K^{'}[i]$];
    $level$ = 1;
    while true do
        slot = ($K[i] >>> (m - \sum_{w=1}^{level} log_2(l_w))$) &
          mask[$level$]);
        (valueInSlot, nodeType) = curDPT.find(slot, $level$);
        if valueInSlot is 0(empty) then
            curDPT.addKNode($i$, $level$, slot);
            break;
        else
            if nodeType is d-node then
                $level$ = $level$ + 1;
                continue;
            else
                collideNum = curDPT.addKNode($i$, level,
                  slot);
                if collideNum $>$ $T[level]$ and level $<$ max
                  then
                    curDPT.addDNode(slot, $level$);
                    curDPT.redistributeObjects(slot, $level$);
                break;
return I;
```

B. Approximate Nearest Neighbor Query

The ANN query operation is as follows:

- **Calculate query's hash value and partition-ID:** The operation is the same as the quantization step.
- **Calculate Δ-step partition-IDs:** The choice of Δ leverages the ANN query's accuracy and efficiency. The way to calculate Δ-step partition-IDs is described in Section II-C.
- **Cenerate probes of query:** To achieve index space efficiency and make the best use of DPT. We integrate our search algorithm with Multi-probes LSH search strategy [15], which not only considers the main slot

where the query falls but also the slots that are "close" to the main slot.

- **Search the DPT:** the search algorithm starts from the root level, then move down to deeper level and terminate until reaching a slot containing k-nodes.

Algorithm 4 shows the details of query operation in one hash table.

```
Algorithm 4: Search($q$, I, $G$, $G^{'}$, $\Delta$, $l$)
Input: Query $q$; Index I; Quantization hash functions $G$;
       Partition hash functions $G^{'}$; $\Delta$-step search; Length of
       d-node in each level $l = l_1, l_2, \dots, l_{max}$;
Output: Result $R$
1  $R = \emptyset$, $K_q = sign(q \cdot G)$, $K_q^{'}$ = Partition($K_q$, $G'$);
2  max = $l.length$;
3  mask = Array($l_1 - 1$, $l_2 - 1$,..., $l_{max} - 1$);
4  multiProbes = GenerateMultiProbes($K_q$);
5  $\Delta$-Partitions = GenerateDeltaPartitions($K_q^{'}$, $\Delta$);
6  for $i = 0; i < \Delta$-Partitions.length do
7      curDPT = I[$\Delta$-Partitions[$i$]];
8      for $j = 0; j < multiProbes.length$ do
9          $level$ = 1;
10         curR = $\emptyset$;
11         probes = multiProbes[$j$];
12         while true do
13             slot = (probes >>> $(m - \sum_{w=1}^{level} log_2(l_w))$) & mask($level$);
14             (valueInSlot, nodeType) = curDPT.find(slot, $level$);
15             if valueInSlot is 0(empty) then
16                 R.add(curR);
17                 break;
18             else
19                 if nodeType is d-node then
20                     $level$ = $level$ + 1;
21                     continue;
22                 else
23                     /*If it is k-node, retrieve the all k-nodes under this slot*/;
24                     curR = curDPT.getKnodes($level$, slot);
25                     R.add(curR);
26                     break;
27 return $R$;
```

IV. Experiment

A. Setup and Dataset

We implement our method on a Linux Intel(R) Xeon(R) server(2.20GHz, 32.0GB memory) and evaluate the performance by using three widely used datasets for ANNs as follows: **Fashion-MNIST** [22] It is a dataset of Zalando's article images consists of 60,000 examples. Each example is a 784-dimensional vector. Compared to MNIST [26], Fashion-MNIST is more difficult because the images are more complicated. **SIFT** [27] It is an images dataset contains 1M objects. Each data object is a 128-dimensional SIFT feature which is extracted from Caltech-256 by using open source VLFeat library. **GloVe** [28] It consists of 1.2M 100-dimensional word embedding in vector space trained from tweets. For each dataset, we sample a subset of 1000 data objects as query set.

B. Evaluation Protocols

- $Recall$ is widely used to evaluate the accuracy of the return objects in many ANNs work [8], [29]. Given a query q, let $R^* = \{o_1^*, o_2^*, o_3^* \dots, o_k^*\}$ be the ground truth of top k nearest neighbors to q, our method also returns k objects $R = \{o_1, o_2, o_3 \dots, o_k\}$. Both results are ranked by the increasing order of their distance to q. The $recall$ with repect to q is computed as $recall(q) = |R^* \bigcap R|/|R^*|$.
- *Time* mainly consists of two parts: 1) The searching time in each DPT to find the closest objects to the query objects; 2) The calculating time to verify all candidates to get top k nearest neighbors. We use it to evaluate the time performance of our method.

C. Performance of Partition Strategy

To see whether our content-based partition strategy can efficiently divide the similar objects into one partition. We analyze the distribution of top k ground truth nearest neighbors in different Δ-step partitions. Given a query q, we first calculate the q's original partition-ID. Then we follow the approach in Section II-C to generate q's different Δ-step partitions. We also calculate q's top k ground truth nearest neighbors' partition-ID to see where these ground truth nearest neighbors fall into. Since each query's top k nearest neighbors will fall into different partitions, we sample 1000 queries and get the average distribution. As shown in the Figure 6, when $M = 2$, 92% to 97% of the top k nearest neighbors are partitioned into the original(0-step) partition even with the increment of k. When M rises, the top k nearest neighbors are lightly decentralized into different step partitions, but the original partition still maintains a high percentage around 90%, which proves the efficiency of content-based partition strategy used in this paper. In the meanwhile, we find the missing top k nearest neighbors are always in 1-step partitions, it is the reason why we adopt the 1-step search approach to improve the accuracy. When M reaches to 5 and 6, the percentage in original partition drop to 77% to 81% for all of three datasets, which causes the accuracy loss dramatically. However, the 1-step partitions still contain the most part of the missing top k nearest neighbors. In Figure 6(d),(e) on Fashion-MNIST, the 2-step partitions contains more top k nearest neighbor than Glove and SIFT, it is because the data of Fashion-MNIST is denser than other two datasets.

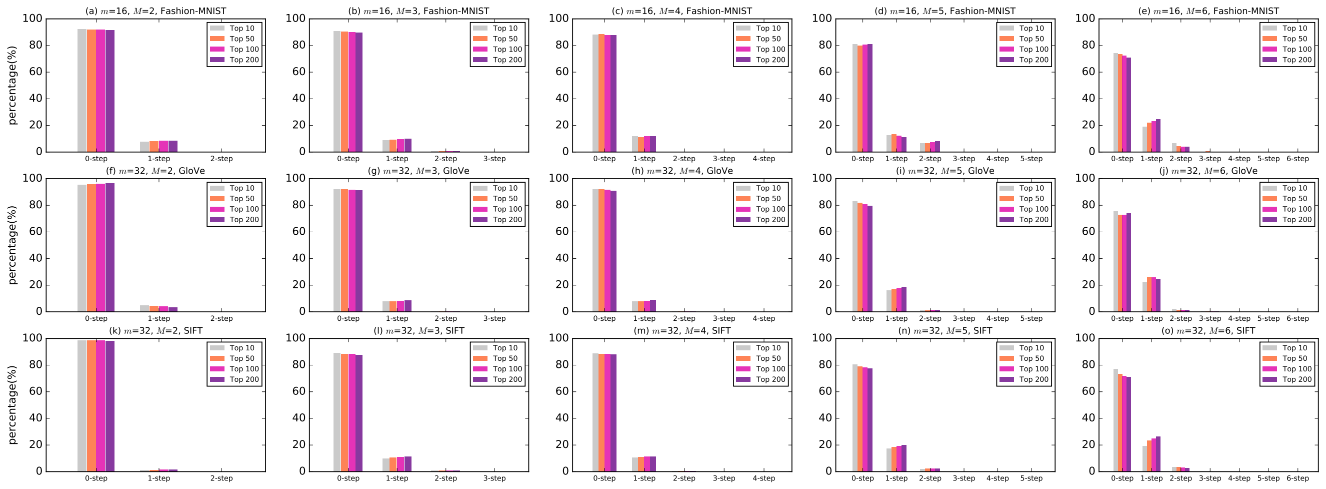


Figure 6: Top k ground truth distribution in different Δ-step partitions.

Table II: Different Δ-step search performance with different M. The Δ means the depth of searched partitions. For example, Δ = 1 means it searches the 0-step partitions and 1-step partitions.

		M=2			M=3				M=4					M=5						M=6						
	Δ	0	1	2	0	1	2	3	0	1	2	3	4	0	1	2	3	4	5	0	1	2	3	4	5	6
Glove	recall	0.87	0.93	0.94	0.84	0.91	0.94	0.95	0.78	0.91	0.93	0.94	0.94	0.76	0.90	0.92	0.93	0.93	0.93	0.72	**0.89**	0.91	0.92	0.93	0.93	0.93
	time(ms)	32.1	52.1	55.8	27.4	38.8	47.3	48.6	23.8	34.2	52.1	55.6	58.9	20.2	30.1	52.8	69.7	75.3	76.4	17.6	**26.8**	52.7	73.9	82.4	86.3	87.8
Fashion-MNIST	recall	0.88	0.92	0.93	0.84	0.90	0.92	0.92	0.82	**0.89**	0.93	0.93	0.93	0.80	0.88	0.89	0.90	0.92	0.92	0.78	0.89	0.90	0.92	0.92	0.93	0.93
	time(ms)	56.1	67.4	68.9	47.3	59.8	68.3	69.7	37.8	**53.6**	73.7	80.7	81.9	33.4	55.7	78.2	82.7	85.1	85.9	31.2	57.3	68.5	81.2	88.5	93.1	96.4
SIFT	recall	0.85	0.91	0.92	0.79	0.89	0.89	0.90	0.75	**0.88**	0.91	0.91	0.91	0.67	0.86	0.89	0.90	0.90	0.90	0.65	0.85	0.90	0.91	0.91	0.91	0.91
	time(ms)	19.4	24.1	26.1	14.7	20.3	23.6	25.0	11.2	**19.7**	33.9	34.8	35.2	9.8	22.2	44.3	49.2	50.7	51.2	9.7	21.6	45.1	55.5	61.2	63.7	64.1

D. *Performance of Δ-step Search Strategy*

From the previous analysis of ground truth k nearest neighbor distribution in Section IV-C, we learned that similar objects are still likely to be divided into different partitions. However, the original(0-step) partition keeps the highest rate of top k similar objects, and the 1-step partitions keep the second. To examine the different Δ-step search performance, we use the average query time and average recall to measure the efficiency and accuracy. We fix the parameters $L = 10$, $T = \{1000, 800, 600, 400\}$ and $l = \{128, 128, 128, 128\}$ for SIFT, $L = 15$, $T_h = \{1000, 800, 600, 400\}$ and $l = \{128, 128, 128, 128\}$ for Glove, $L = 25$ $T_h = \{200, 150, 100, 50\}$ and $l = \{128, 128, 128, 128\}$ for Fashion-MNIST. As the $\Delta_{max} = M$, we do experiments on various M and evaluate all possible Δ-step searches. The results are shown in Table II, we learned that (1) The 0-step search costs the shortest time with the lowest recall for different M on three datasets. The reason is that 0-step search only explores one partition, which roughly contains $100/2^M\%$ data. The 0-step search loses more accuracy with larger M, which is called accuracy loss phenomenon discussed in Section II-C. (2) Under the same M, with the increment of Δ, the recall and query time rise together as a result of searching more partitions. In addition, increasing Δ from 0 to 1 gains the recall most, which verifies the conclusion that 1-step partitions contain most of the similar objects as we tested in Section IV-C. (3) More partitions are created by increasing M. Thus, most of the time, more query time is required in large M compared to small M even to achieve the same recall. While $\Delta = 1$ is always the best trade-off. The best performance combinations are emphasized with bold font. For example, on Glove, When $M = 6$, we achieve recall = 0.89 with 26.8ms query time by doing 1-step search. Compared to the 0-step search, it mitigates the accuracy loss by increasing the recall from 0.72 to 0.89. In the meanwhile, the query time is relatively small compared with $M = 2$ and $\Delta = 0$ on Glove. (4) The performance of three datasets is improved by applying the 1-step search. In conclusion, if the system is more inclined to query speed, the 0-step search is fine. While the system considers a relatively high accuracy, 1-step search is strongly recommended. Besides, one of the main problems affecting the performance of LSH-based methods is limitation of hash tables, which is also the reason why we apply Multi-probes search strategy [15] on DPF, as discussed in Section III-B. As we can see in Figure 8, to achieve the same recall = 0.9, the number of searched DPT is reduced around 68% by using Multi-probes search.

E. *Compared with Other Methods*

To demonstrate the effectiveness of the proposed approach in this paper, we use the recall-time curve to measure the

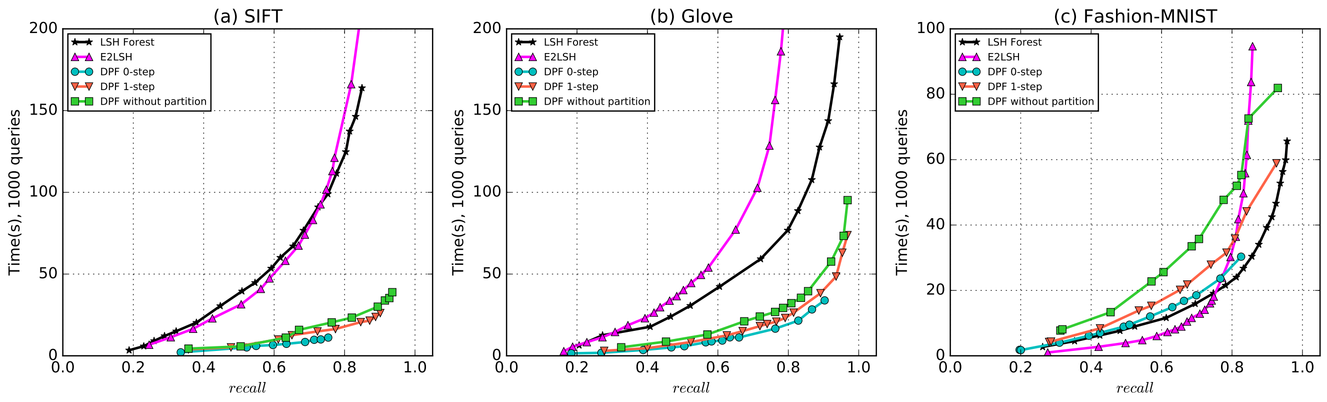


Figure 7: Top 10 ANNs results of 1000 queries on three datasets. We $k = 10$. Low and to the right is better.

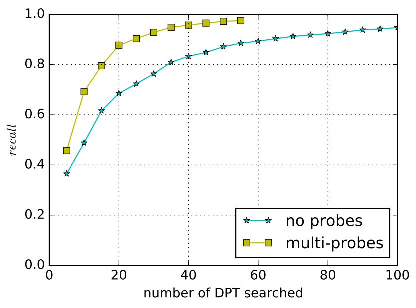


Figure 8: Impact of Multi-probes on Glove dataset

Table III: The maximum number of L.

	SIFT	Glove	Fashion-MNIST
E2LSH	20	40	80
LSH Forest	46	30	50
DPF	10	25	20

performance. Our method is compared with the following two state-of-the-art LSH-based methods. **E2LSH** [9] uses a conventional hash to map the hash values into linear index. We use 16-bit hash values for Fashion-MNIST and 20-bit for Glove and SIFT. **LSH Forest** [13] is a tree-based generation of LSH, where heavily loaded hash buckets are recursively partitioned with embedded hash tables. It uses 32-bit hash values of fixed length and applies SRP hash family to approximate the cosine distance. **Dynamic Partitioned Forest(DPF)** is the method proposed in this paper. We test the DPF with and without partitioning. Furthermore, integrated with the Δ search strategy, we test $\Delta = 0$ and $\Delta = 1$ with $M = 4$ for all three datasets.

The recall-time curves of these methods are shown in Figure 7. Several interesting conclusions are drawn as follows:

1) In SIFT and Glove datasets, DPF outperforms the other two methods. In SIFT dataset, the query speed of DPF 1-step reaching the recall = 0.8 is 6.5 times faster than LSH Forest, and 7.5 times faster than E2LSH. In Glove dataset, the query speed of DPF 1-step reaching the recall = 0.8 is 3.2 times faster than LSH Forest, and 9 times faster than E2LSH.
2) In SIFT dataset, DPF 0-step is the fastest way to get the top 10 NN. However, using the same index space, the maximum recall it can achieve is 0.77, while the recalls of DPF 1-step and DPF without partition can achieve over 0.9. In addition, DPF 1-step is 1.4 times faster than DPF without partition and 0.75 times slower than DPF 0-step with the same M. This finding is also presented in both Glove and Fashion-MNIST datasets.
3) In Glove dataset, using the same index space, the maximum recall DPF 0-step can achieve is 0.9, which is higher than SIFT and Fashion-MNIST datasets.
4) In Fashion-MNIST dataset, E2LSH performs the best before recall reaches 0.75, but then falls behind LSH Forest and DPF 1-step.
5) The Table III shows the space overhead in our experiments. The large L is, the more hash tables are created. Compared with E2LSH and LSH Forest, DPF significantly decreases 17% to 78% space overhead.

Overall, DPF dramatically improves the performance of ANNs in SIFT and Glove datasets. In Fashion-MNIST dataset, DPF also performs positively.

V. Conclusion

In this paper, we propose an efficient and distributed indexing scheme to support similarity search over large-scale data. We point out two flaws (dramatically accuracy loss after distributed scheme and performance degradation by using static number of bits to construct index) in existing methods. In order to solve them, we propose a hierarchical Δ-step search approach to mitigate the accuracy loss after applying distributed scheme. A distribution-aware index structure called dynamic partition forest is designed to dynamically used the bits(hyperplanes) to partition high collision space.

Compared with other two methods, experimental results demonstrate the efficiency and accuracy of our method.

REFERENCES

[1] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic *et al.*, "Query by image and video content: The qbic system," *computer*, vol. 28, no. 9, pp. 23–32, 1995.

[2] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: Scalable online collaborative filtering," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 271–280.

[3] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[4] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[5] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

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

[7] E. C. Gonzalez, K. Figueroa, and G. Navarro, "Effective proximity retrieval by ordering permutations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 9, pp. 1647–1658, 2008.

[8] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

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

[10] B. Bahmani, A. Goel, and R. Shinde, "Efficient distributed locality sensitive hashing," in *Proceedings of CIKM '12*, ser. CIKM '12, 2012.

[11] A. Andoni, "E2lsh 0.1 user manual," *http://www. mit. edu/andoni/LSH/*, 2005.

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

[13] 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*. ACM, 2005, pp. 651–660.

[14] 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.*, vol. 7, no. 9, pp. 745–756, May 2014.

[15] 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*, ser. VLDB '07. VLDB Endowment, 2007, pp. 950–961.

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

[17] 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*, vol. 36, no. 10, pp. 1963–1974, 2014.

[18] W. Kong and W.-J. Li, "Isotropic hashing," in *Advances in neural information processing systems*, 2012, pp. 1646–1654.

[19] K. Zhao, H. Lu, and J. Mei, "Locality preserving hashing." in *AAAI*, 2014, pp. 2874–2881.

[20] Y. Cao, M. Long, J. Wang, H. Zhu, and Q. Wen, "Deep quantization network for efficient image retrieval." in *AAAI*, 2016, pp. 3457–3463.

[21] X. He and P. Niyogi, "Locality preserving projections," in *Advances in neural information processing systems*, 2004, pp. 153–160.

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

[23] 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*, vol. 6, no. 14, pp. 1930–1941, 2013.

[24] M. X. Goemans and D. P. Williamson, "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming," *Journal of the ACM (JACM)*, vol. 42, no. 6, pp. 1115–1145, 1995.

[25] 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*. IEEE, 2017, pp. 205–212.

[26] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[27] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2011.

[28] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[29] B. Naidan, L. Boytsov, and E. Nyberg, "Permutation search methods are efficient, yet faster search is possible," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1618–1629, 2015.