Dynamic Multi-probe LSH: An I/O Efficient Index Structure for Approximate Nearest Neighbor Search

Shaoyi Yin[*], Mehdi Badr, and Dan Vodislav

ETIS, Univ. of Cergy-Pontoise / CNRS, France
yinshaoyi@gmail.com, {mehdi.badr,dan.vodislav}@u-cergy.fr

Abstract. Locality-Sensitive Hashing (LSH) is widely used to solve approximate nearest neighbor search problems in high-dimensional spaces. The basic idea is to map the "nearby" objects into a same hash bucket with high probability. A significant drawback is that LSH requires a large number of hash tables to achieve good search quality. Multi-probe LSH was proposed to reduce the number of hash tables by looking up multiple buckets in each table. While optimized for a main memory database, it is not optimal when multi-dimensional vectors are stored in a secondary storage, because the probed buckets may be randomly distributed in different physical pages. In order to optimize the I/O efficiency, we propose a new method called Dynamic Multi-probe LSH which groups small hash buckets into a single bucket by dynamically increasing the number of hash functions during the index construction. Experimental results show that our method is significantly more I/O efficient.

Keywords: Locality sensitive hashing, indexing, high-dimensional database, approximate nearest neighbor search.

1 Introduction

Nearest neighbor search (NNS), also known as similarity search, consists in finding, for a given point in a high-dimensional space, the closest points from a given set. The nearest neighbor search problem arises in many application fields, such as pattern recognition, computer vision, multimedia databases (e.g. content-based image retrieval), recommendation systems and DNA sequencing. Various indexing structures have been proposed to speed up the nearest neighbor search. Early proposed tree-based indexing methods such as R-tree [10], K-D-tree [2], SR-tree [14], X-tree [3] and M-tree [5] return exact query results, but they all suffer the "curse of dimensionality": it has been shown in [19] that they exhibit linear complexity at high dimensionality, and that they are outperformed on average by a simple sequential scan of the database if the number of dimensions exceeds even moderate values, e.g. around 10.

[*] Shaoyi YIN is currently an associate professor at IRIT Laboratory, Paul Sabatier University, France.

In fact, for most of the applications, exact nearest neighbors are not more meaningful than the so-called ε-approximate nearest neighbors, because the feature vectors used to represent the objects are usually already imprecise. This phenomenon, called semantic gap is inherent e.g. in content-based image retrieval (CBIR), where feature vectors expressing low-level image properties are used to answer high-level user queries. Formally, the goal of the ε-approximate NNS is to find the data object within the distance $\varepsilon \times R$ from a query object, where R is the distance from the query point to its exact nearest neighbor. The most well-known methods for solving the ε-approximate NNS problem in high-dimensional spaces are Locality Sensitive Hashing (LSH) [9], [12] and its variants [1], [15].

The basic LSH method uses a family of locality-sensitive hash functions to hash nearby objects into the same bucket with a high probability. Several hash functions are combined to produce a compound hash signature corresponding to finer hash buckets. For a given query object, the indexing method hashes this object into a bucket, takes the objects in the same bucket as ε-approximate NNS candidates, and then ranks the candidates according to their distances to the query object. A side effect of combining several hash functions is that some "nearby" points may be hashed into different buckets. In order to increase the probability of finding all the nearest neighbors, the LSH method usually creates multiple hash tables and each hash table is built by using independent locality-sensitive hash functions. The number of hash tables is usually over a hundred [9] and sometimes several hundred [4]. This becomes a problem in terms of space consumption. To reduce the number of hash tables needed, the Multi-probe LSH method [15] has been proposed by Lv et al.

The main idea of Multi-probe LSH is to build on the basic LSH indexing method, but to use a carefully derived probing sequence to look up multiple buckets that have a high probability of containing the nearest neighbors of a query object. By probing multiple buckets in each hash table, the method requires far fewer hash tables than the previously proposed LSH methods. As an in-memory algorithm, Multi-probe LSH method is very efficient; however, if the feature vectors cannot be stored in main memory, the query cost becomes rather high, because the probed buckets may be randomly distributed in different disk pages. In this paper, we consider the case where the feature vectors are stored in a secondary storage, even though we suppose that the index structure itself is still in main memory.

In order to improve the query efficiency of the Multi-probe LSH, we present in this paper a new method called Dynamic Multi-probe LSH (DMLSH). The main modification to Multi-probe LSH is that we dynamically adapt the number of hash functions for each bucket in order to produce buckets whose size fits a disk page. With the same parameter setting, our method always requires less I/O cost and provides higher query accuracy than Multi-probe LSH. The experimental results have shown that the gain is significant.

The rest of this paper is organized as follows. We first review the background knowledge and the related work in Section 2, and then present our DMLSH method in Section 3. We describe experimental studies in Section 4 and conclude in Section 5.

2 Background and Related Work

Approximate Nearest Neighbor Search. Definition (ε-Nearest Neighbor Search (ε-NNS)) [9]. Let us consider the normed space l_p^d representing the d-dimensional Euclidian space R^d with the l_p norm and the distance $D(\cdot,\cdot)$ induced by this norm. Given a set P of points in l_p^d, solving the ε-NNS problem in P consists in preprocessing P so as to efficiently return a point $p \in P$ for any given query point q, such that $D(p,q) \leq (1+ \varepsilon)\, D(q,P)$, where $D(q,P)$ is the distance of q to its closest point in P.

Note that the above definition generalizes to any metric space. It also generalizes naturally to finding K>1 approximate nearest neighbors. In the approximate K-NNS problem, we wish to find K points $p_1, \ldots , p_k$, such that the distance of p_i to the query point q is at most $(1+ \varepsilon)$ times the distance from q to the i-th nearest point in P.

Locality Sensitive Hashing. The basic idea of LSH is to use hash functions that map similar objects into the same hash bucket with high probability. Performing a similarity search query on an LSH index consists of two steps: 1) using LSH functions to select "candidate" objects for a given query q, and 2) ranking the candidate objects according to their distance to q.

Definition (Locality-Sensitive Hash Family) [9], [12]. A family $H = \{h: S \rightarrow U\}$ is called (r, ε, p_1, p_2)-sensitive, with $p_1 > p_2 > 0$, $\varepsilon > 0$, if for any $p, q \in S$, the following conditions hold:

- If $D(p, q) \leq r$ then $Pr_H[h(p)=h(q)] \geq p_1$;
- If $D(p, q) > (1+ \varepsilon)r$ then $Pr_H[h(p)=h(q)] \leq p_2$.

Here S is a set of objects and $D(\cdot,\cdot)$ is the distance function of elements in the set S.

Different LSH families can be used for different distance functions. Families for Jaccard distance, Hamming distance, l_1 and l_2 distances are known. The most widely used one is the LSH family for Euclidean distance proposed by Datar et al.[7]. Each function is defined on R^d as follows: $h_{a,b}(v) = \lfloor (a \cdot v + b)/W \rfloor$, where a is a random d-dimensional vector and b is a real number chosen uniformly from the range [$0, W$]. $a \cdot v$ is the dot product of vectors a and v. Each hash function maps a d-dimensional vector v onto into an integer value.

Given a locality-sensitive hash family $H = \{h: S \rightarrow U\}$, an LSH index is constructed as follows. (1) For an integer $M > 0$, define a family $G = \{g: S \rightarrow U^M\}$ of compound hash functions; for any $g \in G$, $g(v) = (h_1(v), h_2(v), \ldots , h_M(v))$, where $h_j \in H$ for $1 \leq j \leq M$. (2) For an integer L>0, choose $g_1, g_2, \ldots g_L$ from G, independently and uniformly at random. Each of the L functions g_i ($1 \leq i \leq L$) is used to construct one hash table, resulting in L hash tables. (3) Insert each vector v into the hash bucket to which $g_i(v)$ points to, for $i = 1, \ldots, L$. A K-NNS query for vector q is processed in two steps. (1) Compute the hash value $g_i(q)$ and retrieve all the vectors in bucket $g_i(q)$ for $i = 1, \ldots, L$ as candidates. (2) Rank the candidates according to their distances to the query object q, and then return the top K objects. Note that compound hash functions reduce the probability that distant vectors belong to a same bucket, but increase the risk of nearby points separated into different buckets. Merging candidates from several hash tables reduces the risk of missing close objects.

Fig. 1 presents an example of 2-NNS, where $L = 3$. Objects $p_1, p_2, \ldots, p_5$ are inserted into three hash tables corresponding to hash functions g_1, g_2, g_3. For a given query q, by checking the three buckets $g_1(q), g_2(q), g_3(q)$, we get the candidates p_1, p_3, p_4, p_5. By ranking their distances to q, we return p_1 and p_3 as the 2-NN objects.

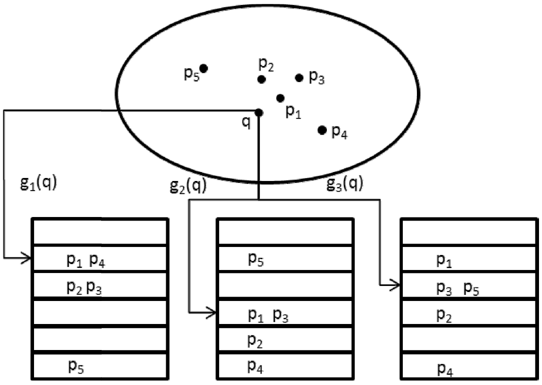

Fig. 1. LSH index structure

The main drawback of the above index structure is that it may require a large number of hash tables to cover most nearest neighbors. For example, over 100 hash tables are needed to achieve 1.1-approximation in [9], and as many as 583 hash tables are used in [4]. Thus, the whole data structure takes too much space. In order to reduce the number of hash tables while keeping a good approximation ratio, Multi-probe LSH [15] was proposed.

Multi-probe LSH. The main idea of the Multi-probe LSH method is to use a carefully derived probing sequence to check multiple buckets that are likely to contain the nearest neighbors of a query object. Since each hash table provides more candidates, the number of needed hash tables could be reduced.

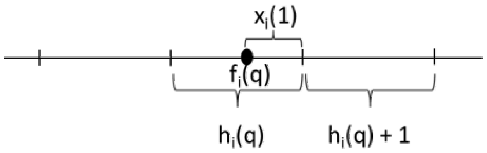

Fig. 2. Distance from q to the boundary of its neighbor bucket

Multi-probe LSH is based on the family of locality-sensitive functions of Datar et al.[7] described above, returning integer values. To derive the probing sequence, the authors first define a hash perturbation vector $\Delta = \{\delta_1, \ldots, \delta_M\}$ where $\delta_i \in \{-1, 0, 1\}$ and M is the number of hash functions for each hash table. Given a query q, the basic LSH method checks the hash bucket $g(q) = (h_1(q), h_2(q), \ldots, h_M(q))$, while

Multi-probe LSH checks also the buckets $g(q) + \Delta_1$, $g(q) + \Delta_2$, ..., $g(q) + \Delta_T$. These buckets are ordered according to their "success probability" which is a score estimated using formula score(Δ) = $\sum_{i=1}^{M} x_i\,(\delta_i)^2$, where $x_i(\delta_i)$ is the distance from q to the boundary of the bucket $h_i(q) + \delta_i$. For example, in Fig. 2, $x_i(1)$ is the distance from q to the boundary of the bucket $h_i(q) + 1$, where $h_i(q) = (a_i \cdot q + b_i)/W$ and $f_i(q) = a_i \cdot q + b_i$.

Multi-probe LSH is originally designed as an in-memory algorithm. In this paper, however, we consider that the multidimensional vectors are stored in a secondary storage such as hard disk. The problem with Multi-probe LSH in this new context is that many probes are needed for each hash table and the probed buckets are randomly stored in the disk, so a lot of I/Os are required for each query. Our objective in this paper is to reduce the number of I/Os for a K-NN search.

Other Related Work. LSH Forest indexing method [1] represents each hash table by a prefix tree to eliminate the need of finding the optimal number of hash functions per table. However, this method does not help reduce the number of hash tables, so the space consumption and query time are not improved. There exists some other work which tends to estimate optimal parameters with sample datasets [8], use improved hash functions [11], [13], [17], [18] or divide the dataset into clusters before building LSH indexes [16]. All these methods are complementary to the Multi-probe LSH and our improved structure and could be combined with our method in order to achieve better performance and quality.

3 Dynamic Multi-probe LSH

3.1 Overview

The main idea of DMLSH is to dynamically vary the granularity of buckets in order to adapt the number of objects they contain to the size of a disk page. We use the same locality-sensitive hash functions and the same probing sequence as Multi-probe LSH. More precisely: 1) Instead of directly building a hash table by using all M functions, we first build a hash table by using only one LSH function. If a bucket contains more than l objects (where l is the number of objects contained in a disk page), we add a second LSH function to this bucket in order to split it into several small buckets. If some small bucket still contains more than l objects, we continue adding LSH functions until each bucket contains less than l objects or the number of functions used becomes to be M. We store the signatures of all these buckets in to a B+ tree. Note that these signatures have different lengths, so the keys in the B+ tree have variable size. 2) We use the sequence probing algorithm of Multi-probe LSH to generate the signatures of the buckets to be probed. If the bucket signature exists in the B+ tree index, we will take the objects in the corresponding bucket as candidates; otherwise, we will check the bucket whose signature is a prefix of the generated signature.

Let us explain these principles through an example. In Fig. 3(a), a basic LSH table has been built using 2 hash functions h_1 and h_2. For a given query q, if we use Multi-probe algorithm to generate 6 probes, the buckets chosen are those of signatures 11, 01, 10, 00, 21 and 20. It means that, we need to access 6 pages on the disk to get 7 candidates. In Fig. 3(b), instead of using directly 2 hash functions, we use one hash function h_1 first, then only if the number of objects in a bucket exceeds a threshold l (in this example l=2), we add a second hash function for this bucket. For the same query q, we use the algorithm of the Multi-probe LSH to generate the same probing sequence, i.e. 11, 01, 10, 00, 21 and 20. For a signature that corresponds to no bucket (e.g. 11), we take the bucket whose signature is its prefix (i.e. bucket 1 for signature 11). Thus, the probing sequence becomes 1, 01, 00 and 2. Only 4 disk pages need to be accessed.

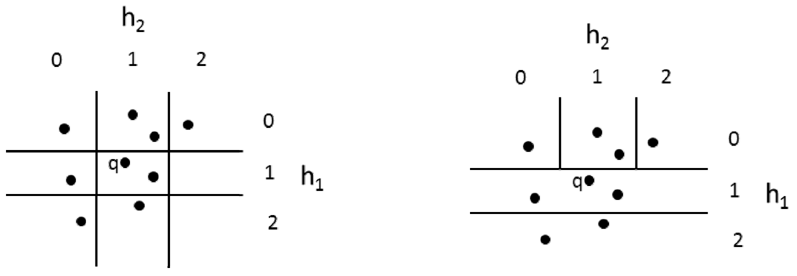

Fig. 3. Dynamically adding hash functions

3.2 Index Construction

As in the Multi-probe LSH, we use L hash tables, and for each hash table we randomly generate M LSH functions. The difference between our method and Multi-probe LSH is that not every inserted object needs to be hashed with all the M functions, and the hash results are stored in a B+ tree rather than a normal hash table. However, the search that we do in the B+ tree is not an exact match. Instead, we may return a prefix of the searched key, for example, when searching for 001010, we may return 001 as a result. Thus, we have slightly modified the search algorithm of the traditional B+ tree. Note that signatures (keys) stored in the B+ tree are sequences of "digits", where a digit is an integer value returned by an LSH function. For simplicity, our examples only consider binary digits.

DMLSH Tree. We call DMLSH tree the specific (in memory) B+ tree that stores the signatures of the buckets produced by the DMLSH method, together with some extra information about these buckets. Note that only *non-empty buckets* are stored in a DMLSH tree.

The only modification that we have made to the B+ tree algorithm described in [6] is that we overloaded the comparison operators of the keys. The new definitions of the operators "==", "<" and ">" are as follows:

- Given two keys k_1 and k_2, we say that $k_1 == k_2$ if one of k_1 or k_2 is a prefix of the other one (including the case where $k_1 = k_2$ as sequences).

- If $k_1 == k_2$ is not true, it is easy to show that k_1 and k_2 have the form $k_1 = k + d_1 + s_1$ and $k_2 = k + d_2 + s_2$, where k, s_1, s_2 are digit sequences (possibly empty) and d_1, d_2 are single digits, with $d_1 != d_2$. We say that $k_1 < k_2$ (respectively $k_1 > k_2$) if $d_1 < d_2$ (respectively $d_1 > d_2$).

For example, we have 001 == 00101, 0101 < 011 and 10110 > 100010, etc.

The following property holds for DMLSH bucket signatures: *For any two distinct signatures k_1 and k_2 produced by DMLSH, we have either $k_1 < k_2$ or $k_1 > k_2$.*

Proof. Let us suppose that for two distinct signatures produced by DMLSH we have $k_1 == k_2$ with e.g. k_1 being a prefix of k_2. In this case, the bucket of signature k_2 is included into the one of signature k_1. But this is in contradiction with the fact that buckets issued from DMLSH produce a partitioning of the multidimensional space. Since it is not possible to have $k_1 == k_2$, we deduce from the properties of the "==", "<" and ">" operators presented above that $k_1 < k_2$ or $k_1 > k_2$. Consequently, signatures produced by DMLSH respect a strict total order induced by the "<" operator.

Each hash table is maintained as a DMLSH tree as follows: the bucket signature (hash value) of each non-empty bucket is treated as a key and the keys are inserted into the tree. Each key in the leaf node is followed by (Fig. 4):

1. a counter for the number of hash functions used for getting this signature, *nb_hashes_used*;
2. a counter for the number of vectors in the corresponding hash bucket, *nb_vectors*;
3. the address of the page containing these vectors, @;
4. a set *HV* containing the hash values of these vectors computed by using all the *M* functions.

Bucket number	nb_hashes_used	nb_vectors	@	HV

Fig. 4. Data structure of each item in the leaf nodes

Construction of the DMLSH Index. Algorithm 1 shows the process of the index construction. For each of the *L* hash tables, we generate *M* random LSH functions and an initial empty DMLSH tree. Then, for each vector *v* in the database *S*, we insert *v* in each of the DMLSH trees with *InsertVector* (Algorithm 2).

Algorithm 1. Construction of the index structure

```
for i = 1 to L do
    for j = 1 to M do
        Generate a random LSH function $h_{i,j}$
    end for
    Create an empty DMLSH tree $Tree_i$
end for
for each v ∊ S do
    InsertVector(v) (Algorithm 2)
end for
```

As shown below, vector insertion respects the DMLSH strategy to generate new buckets that use as few hash functions as possible. Only when the size of a bucket exceeds a threshold, we add a new hash function for this bucket and distribute the objects into smaller buckets. When the number of hash functions used becomes to be *M*, we stop adding new hash functions and we will store the newly inserted objects in the overflow pages of the full bucket.

Object Insertion. As shown in Algorithm 2, when inserting a new vector *v*, we insert it into each of the *L* hash tables. For each hash table, we compute the hash value *g(v)* of vector *v*, using the *M* functions, and search the key *g(v)* in the corresponding B+ tree. If a prefix of *g(v)* exists in the tree, the function *Tree$_i$.find(g(v))* will return the item in the leaf node which corresponds to the prefix key. Note that a prefix may not exist in the tree, because empty buckets are not stored in the B+ tree. If there is no prefix of the searched key, item is NULL and we will add into the tree a new item with the shortest prefix of *g(v)* which is not a prefix of any other existing key. This is done by function *Tree$_i$.insert(g'(v), item)*. The length of this prefix is computed by function *Nb_hash* (Algorithm 3). Finally, we insert the vector into the bucket page linked to the found (or inserted) prefix key and update the other fields of the leaf item with *AddVectorToItem* (see Algorithm 4).

Algorithm 2. *InsertVector*(v): insert a vector v

```
for i = 1 to L do
    g(v) = ($h_{i,1}$(v), $h_{i,2}$(v), …, $h_{i,M}$(v))
    item = Tree$_i$.find(g(v))
    if item == NULL then
        item = New_leaf_item()
        nb_hashes_used = Nb_hash(v, i) (Algorithm 3)
        g'(v) = ($h_{i,1}$(v), $h_{i,2}$(v), …, $h_{i,nb\_hashes\_used}$(v))
        Tree$_i$.insert(g'(v), item)
    AddVectorToItem(v, item, i) (Algorithm 4)
end for
```

Algorithm 3. *Nb_hash*(v, i): determine nb_hashes_used for vector v in Tree$_i$

```
g(v) = ($h_{i,1}$(v), $h_{i,2}$(v), …, $h_{i,M}$(v))
pred = Tree$_i$.find_pred(g(v))
succ = Tree$_i$.find_succ(g(v))
lcc_pred = length(longest_common_prefix(pred, g(v)))
lcc_succ = length(longest_common_prefix(succ, g(v)))
lcc = max(lcc_pred, lcc_succ)
return lcc+1;
```

Algorithm 3 shows how to determine the number of hash functions to use for a newly inserted vector whose complete hash value does not have a prefix key in the tree. We first compute the key *g(v)* using all *M* hash functions, then we search in the tree for *pred* which is the greatest key smaller than *g(v)* and *succ* which is the smallest key larger than *g(v)*. The next step is to compute *lcc* which is the maximum length of the longest common prefix between *g(v)* and *pred*/*succ*. Note that if *pred* or *succ* do not exist, the corresponding length of the common prefix is 0. At the end, we return *lcc*+1 as the number of hash functions to be used.

Algorithm 4 shows how to add a vector *v* into an item. Insertion is possible only if the counter *nb_vectors* is below the threshold l ($l = B/sizeof(v)$, where *B* is the size of a disk page), or if the maximum number *M* of hash functions is reached. Insertion adds *v* into the page @ (or into an overflow page) and the full signature *g(v)* to *HV*. If insertion is not possible, the bucket is "split" as follows: its item is removed from the B+ tree and all its vectors are reinserted in buckets using one more hash function (Algorithm 5). If the bucket containing the reinserted vector already exists in the tree, the vector is directly inserted; otherwise, a new bucket is created.

Algorithm 4. *AddVectorToItem*(v, item, i): add a vector v into a leaf entry item of Tree$_i$

```
if item.nb_vectors < $l$ or item.nb_hashes_used==M then
    item.nb_vectors++
    $g(v) = (h_{i,1}(v), h_{i,2}(v), \ldots, h_{i,M}(v))$
    Add g(v) into item.HV
    Add v into the page at address item.@ or into an overflow page
else
    Tree$_i$.remove(item)
    for each $v_j \in$ item do
        ReinsertVector($v_j$, item.nb_hashes_used+1, i)
    end for
```

Algorithm 5. *ReinsertVector*(v, k, i): reinsert vector v into Tree$_i$ with k hash functions

```
$g(v) = (h_{i,1}(v), h_{i,2}(v), \ldots, h_{i,k}(v))$
item = Tree$_i$.find(g(v))
if item == NULL then
    item = New_leaf_item()
    item.nb_hashes_used = k
    Tree$_i$.insert(g(v), item)
end if
AddVectorToItem(v, item, i)   (Algorithm 4)
```

Example. Since the insertion algorithm is the same for all the hash tables, we only consider one hash table as an example. For simplicity, we assume the threshold $l = 2$ and the maximum number of hash functions $M = 2$. Initially, we have four objects p_1, p_2, p_3 and p_4, with $h_1(p_1) = 0$, $h_1(p_2) = 1$, $h_1(p_3) = 1$ and $h_1(p_4) = 0$. Their complete hash values with function $g = (h_1, h_2)$ are: $g(p_1) = 00$, $g(p_4) = 01$, $g(p_2) = g(p_3) = 11$. These values are stored in the set *HV* following the prefix keys. The index is shown in Fig. 5(a). The format of the element in the leaf nodes is defined by Fig. 4. When we insert object p_5 with $h_1(p_5) = 1$, $g(p_5) = 10$, the counter *nb_vectors* for the bucket 1 becomes 3 which is larger than *l*, so we need to split this hash bucket (i.e. add one hash function h_2). We reinsert all the objects of bucket 1. Fig. 5(b) shows the result: new entries with keys 10 and 11 are inserted in the B+ tree and the old entry with key 1 is deleted; for key 0, the bucket is not split.

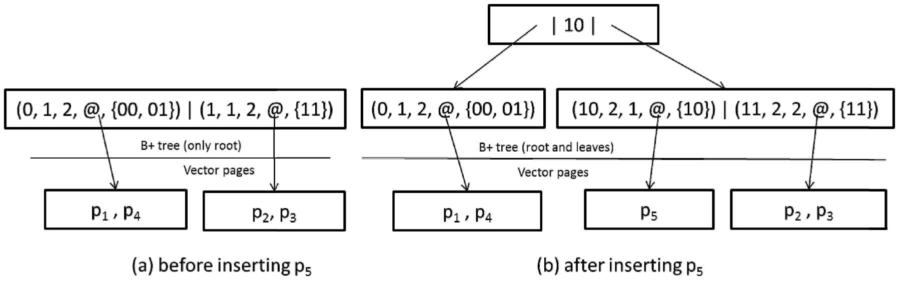

Fig. 5. Example of a DMLSH index

3.3 Approximate K Nearest Neighbor Search

Algorithm. For a given query *q*, we repeat the following process for each of the hash tables. We build an empty B+ tree *Probed_Tree* in memory, used to memorize the signatures of the accessed buckets. This tree has the same properties as a DMLSH tree (Section 3.2.1), but its items only contain a bucket signature. We compute the hash value of *q* using all *M* hash functions, i.e. $g(q) = (h_1(q), ..., h_M(q))$ and generate the probing sequence for $g(q)$. The algorithm of generating the probing sequence can be found in [15]. We note *T* the number of probes.

For each probe, we search the probed key in *Probed_Tree*. If its prefix is found, it means that the bucket has already been checked, so we skip this probe. Otherwise, we search the probed key in the DMLSH tree. If a prefix key p_k is found and $g(q)$ exists in the set *HV*, this means that the bucket of signature $g(q)$ is not empty and is included into the "physical" bucket of signature p_k. Therefore, we add the vectors in the page(s) linked to p_k into the candidate set and insert p_k into *Probed_Tree*.

After retrieving all the candidates, we compute the distances from them to the query *q*, rank them in increasing order of their distances and return the top-k results.

Example. Let us consider the example in Fig. 5(b). Suppose $g(q) = (h_1(q), h_2(q)) = 10$, $T = 3$ and the generated probing sequence is 10,00,01. For the first probe 10, we find the signature in the tree, we load the linked page, add p_5 as a candidate and insert 10 into *Probed_Tree*. For the second probe 00, we find its prefix 0. Since 00 is in the *HV* set ($00 \in \{00, 01\}$), we load the linked page, add p_1 and p_4 as candidates and insert 0 into *Probed_Tree*. For the last probe 01, we find its prefix 0 in the tree *Probed_Tree*, meaning that the bucket has already been probed, so we don't need an extra I/O for this probe. In this example, instead of loading 3 pages for the 3 probed buckets, we only loaded 2 pages.

Properties. The DMLSH method proposed in this paper has two important properties compared to the original Multi-Probe LSH. *P1: Under the same parameter setting, the number of I/Os made by DMLSH for a given query q is no more than that made by Multi-probe LSH. P2: Under the same parameter setting, the accuracy of the K-NN search made by DMLSH for a given query q is not lower than that of Multi-probe LSH.* They could be easily proved theoretically, since in DMLSH, 1) several non-full probed buckets may share the same prefix key and they are stored in a single disk page; 2) the candidate set is a superset of that produced by MLSH.

4 Experimental Evaluation

4.1 Methods under Evaluation

DMLSH is an I/O efficient version of Multi-probe LSH, so we will compare these two methods by varying the different parameters: the number of hash functions *M*, the number of hash tables *L* and the number of probes *T*.

Our method could be also combined with basic LSH and its variants mentioned in related work, by organizing each hash table as a DMLSH tree. However, the impact in this case is less important than with Multi-probe LSH, because a single bucket is accessed in each table; also these methods have less practical utility because of the high number of tables. Consequently, we limit our study to the more effective Multi-probe LSH method.

4.2 Dataset

We choose two datasets for our experimental evaluation, widely used in the related work. They are: **Color Data**. The Color dataset contains 68040 vectors of 32 dimensions, which are the color histograms of images in the Corel collection[1]. The dimension values are real numbers with at most 6 decimal digits ranging from 0 to 1.We randomly choose 100 vectors as query examples. **Audio Data**. The audio dataset contains 54387 vectors of 192 dimensions. It is extracted from the LDC SWITCHBOARD-1 collection[2]. The values are real numbers between -1 and 1. *We increase the size of both datasets to be 1 million by inserting noise vectors for the following experiments.* We randomly choose 100 vectors as query examples.

4.3 Evaluation Metrics

We adopt two metrics to measure our method: query efficiency and query accuracy. Since the space consumption of our method is about the same with Multi-probe LSH, we do not consider this metric.

Query Efficiency. Since the vectors are stored in the secondary storage, we evaluate the query efficiency in terms of I/O cost. In the experiments, we set the page size as

[1] `http://kdd.ics.uci.edu/databases/CorelFeatures/`

[2] `http://www.cs.princeton.edu/cass/audio.tar.gz`

the size of 100 vectors. Note that DMLSH introduces a CPU overhead for distance computation, since the number of candidates it produces is larger than for MLSH. However, the measures in this case indicate only a small difference (5%), not significant compared with the I/O saving.

Query Accuracy. We measure the average recall ratio of the 100 *K*-NN queries for *K*=20. Given a query object *q*, let *E(q)* be the set of exact *K*-NN objects, and *F(q)* the set of found *K*-NN objects. Then the recall ratio is defined as follows:

$$\text{Recall} = \frac{|E(q) \cap F(q)|}{|E(q)|} \quad (1)$$

4.4 Experimental Results

In this section, we compare DMLSH and MLSH by varying the number of hash functions *M*, the number of hash tables *L*, respectively the number of probes *T*.

Impact of the Number of Hash Functions *M*. We measured the I/O cost and the recall ratio of the first two methods by varying the maximum number of hash functions (*M*) used for each hash table. For both datasets, the number of hash tables *L* is set to 3 and the number of probes *T* is set to 100. The results are shown in Fig. 6 and Fig. 7.

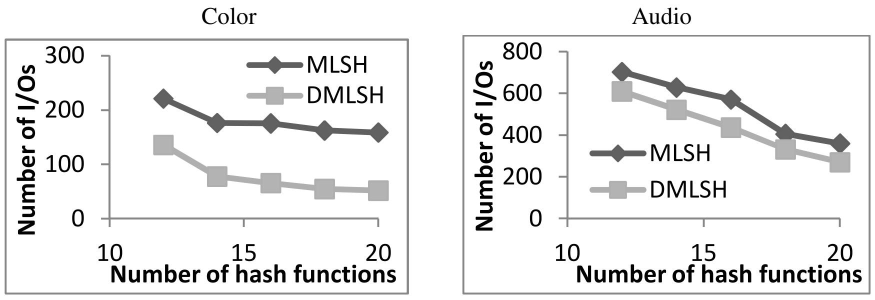

Fig. 6. Impact of M on the I/O cost

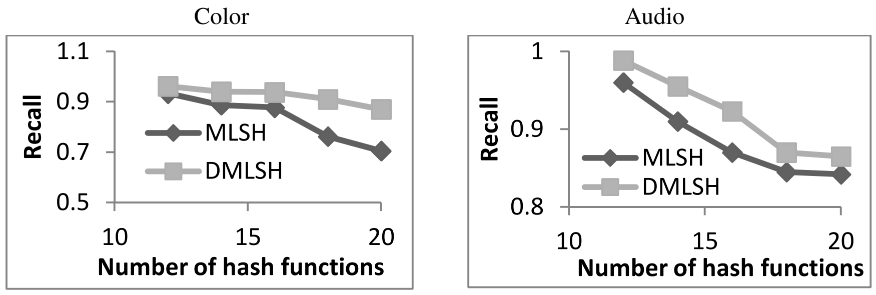

Fig. 7. Impact of M on the recall ratio

For the Color dataset, we set $W = 0.6$. DMLSH reduces the I/O cost by 39% - 67% and increases the recall ratio by 3% - 23%. For the Audio dataset, we set $W = 3.5$. DMLSH reduces the I/O cost by 13% - 25% and increases the recall ratio by 3% - 6%.

We can see that the overall trend is that, when the number of hash functions grows, both the I/O cost and the recall ratio decrease. This is because when we add a new hash function, 1) the average size of each bucket is decreased and 2) more empty buckets are probed.

Impact of the Number of Hash Tables L. Fig. 8 and Fig. 9 show the impact of the number of hash tables L on the I/O cost and on the recall ratio. The number of probes T is set to 100.

For the Color dataset, we set $M = 14$ and $W = 0.6$. DMLSH reduces the I/O cost by 53% - 62% and increases the recall ratio by 3% - 10%. For the Audio dataset, we set $M = 18$ and $W = 3.5$. DMLSH reduces the I/O cost by 16% - 33% and increases the recall ratio by 1% - 9%.

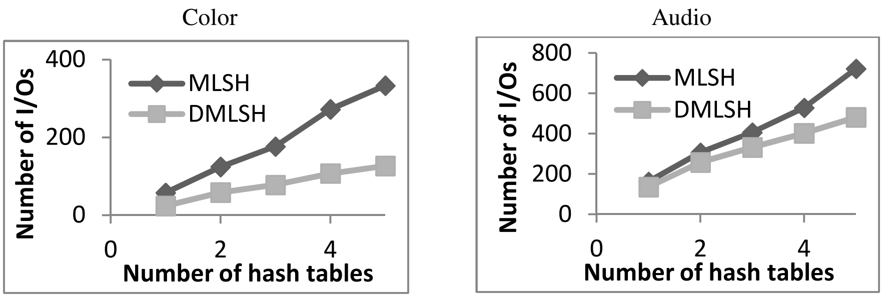

Fig. 8. Impact of L on the I/O cost

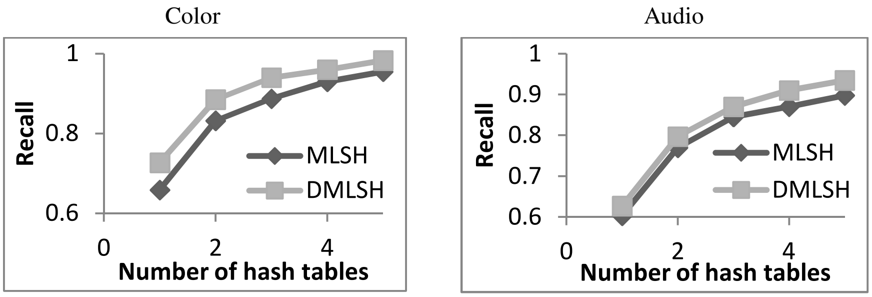

Fig. 9. Impact of L on the recall ratio

When the number of hash tables grows, both the I/O cost and the recall ratio increase. This is normal, because when we use L+1 hash tables, the set of candidates is always a superset of that produced by using L hash tables. The choice of the number of hash tables is a trade-off between query efficiency, query accuracy and space consumption.

We observed that, to achieve the same recall ratio, our method DMLSH needs fewer hash tables than MLSH, hence consumes less space.

Impact of the Number of Probes *T*. In Fig. 10 and Fig. 11, we vary the number of probes from 10 to 170. For both datasets, the number of hash tables L is set to 3.

For the Color dataset, we set $M = 14$ and $W = 0.6$. DMLSH reduces the I/O cost by 33% - 60%. The bigger the number of probes, the higher the reduction of I/O cost. With the same number of probes, DMLSH increases the recall ratio by 4% - 16%. For the Audio dataset, we set $M = 18$ and $W = 3.5$. DMLSH reduces the I/O cost by 2% - 24% and increases the recall ratio by 3% - 5%. To achieve the same recall ratio, our method DMLSH needs fewer probes than MLSH.

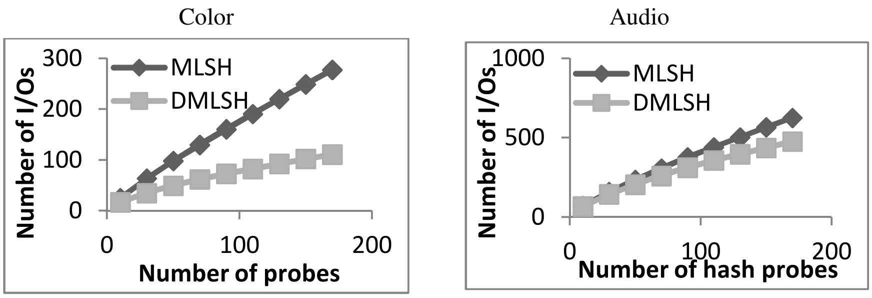

Fig. 10. Impact of T on the I/O cost

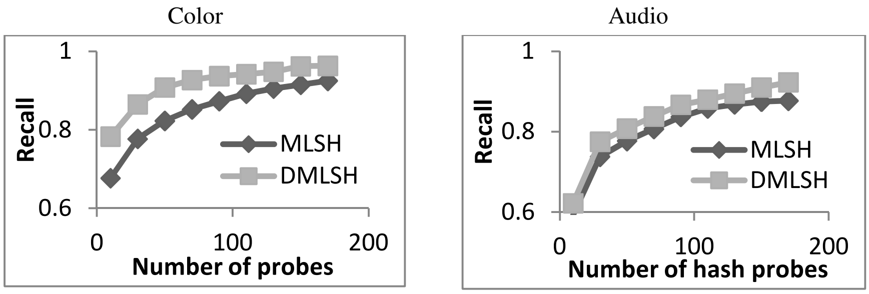

Fig. 11. Impact of T on the recall ratio

5 Conclusion

This paper presents the Dynamic Multi-probe LSH indexing method, which is a more I/O efficient version of the Multi-probe LSH. It dynamically varies the granularity of buckets in order to adapt the number of objects they contain to the size of a disk page. For the construction of the index, it uses initially one hash function and adds a new hash function only when the bucket size exceeds the page size. In the final hash table, the buckets are built by using a different number of hash functions; consequently they have signatures of different length. Bucket signatures are indexed by a slightly modified B+ tree to accelerate the search speed.

For a given query, we first generate the probing sequence and then we access the probed buckets. Since several probed buckets may share the same prefix key and are stored in the same physical page, we need only one single I/O to access these buckets. Thus, the total number of disk accesses is reduced. In addition, since the candidate set is a superset of that produced by Multi-probe LSH, the recall ratio of the approximate *K*-NN query results is always higher than or equal to that of the Multi-probe LSH.

References

1. Bawa, M., Condie, T., Ganesan, P.: Lsh forest: self-tuning indexes for similarity search. In: WWW, pp. 651–660 (2005)
2. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)
3. Berchtold, S., Keim, D.A., Kriegel, H.P.: The X-Tree: an index structure for high-dimensional data. In: Proceedings of the 22nd VLDB Conference, pp. 28–39 (1996)
4. Buhler, J.: Efficient large scale sequence comparison by locality-sensitive hashing. Bioinformatics 17, 419–428 (2001)
5. Ciaccia, P., Patella, M., Zezula, P.: M-tree an efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd VLDB Conference, pp. 426–435 (1997)
6. Comer, D.: The ubiquitous B-tree. ACM Computing Surveys 11(2), 121–137 (1979)
7. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the Twentieth Annual Symposium on Computational Geometry, pp. 253–262 (2004)
8. Dong, W., Wang, Z., Josephson, W., Charikar, M., Li, K.: Modeling LSH for performance tuning. In: CIKM 2008, pp. 669–678 (2008)
9. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proceedings of the 25th Very Large Database (VLDB) Conference, pp. 518–529 (1999)
10. Guttman, A.: R-Trees: A dynamic index structure for spatial searching. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 47–57 (1984)
11. He, J., Liu, W., Chang, S.: Scalable similarity search with optimized kernel hashing. In: ACM SIGKDD, pp. 1129–1138 (2010)
12. Indyk, P., Motwani, R.: Approximate nearest neighbor: towards removing the curse of dimensionality. In: Proceedings of STOC, pp. 604–613 (1998)
13. Jegou, H., Amsaleg, L., Schmid, C., Gros, P.: Query adaptative locality sensitive hashing. In: ICASSP 2008, pp. 825–828 (2008)
14. Katayama, N., Satoh, S.: The SR-tree: an index structure for high-dimensional nearest neighbor queries. In: SIGMOD Conference, pp. 369–380 (1997)
15. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe LSH: efficient indexing for high-dimensional similarity search. In: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), Vienna, Austria, pp. 950–961 (2007)
16. Pan, J., Manocha, D.: Bi-level locality sensitive hashing for k-Nearest Neighbor computation. In: ICDE, pp. 378–389 (2012)
17. Raginsky, M., Lazebnik, S.: Locality-sensitive binary codes from shift-invariant kernels. In: Advances in Neural Information Processing Systems, pp. 1509–1517 (2009)
18. Satuluri, V., Parthasarathy, S.: Bayesian locality sensitive hashing for fast similarity search. PVLDB 5(5), 430–441 (2012)
19. Weber, R., Schek, H., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: VLDB, pp. 194–205 (1998)