Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search

Qin Lv William Josephson Zhe Wang Moses Charikar Kai Li
Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ 08540 USA
{qlv,wkj,zhewang,moses,li}@cs.princeton.edu

ABSTRACT

Similarity indices for high-dimensional data are very desirable for building content-based search systems for feature-rich data such as audio, images, videos, and other sensor data. Recently, locality sensitive hashing (LSH) and its variations have been proposed as indexing techniques for approximate similarity search. A significant drawback of these approaches is the requirement for a large number of hash tables in order to achieve good search quality. This paper proposes a new indexing scheme called *multi-probe* LSH that overcomes this drawback. Multi-probe LSH is built on the well-known LSH technique, but it intelligently probes multiple buckets that are likely to contain query results in a hash table. Our method is inspired by and improves upon recent theoretical work on entropy-based LSH designed to reduce the space requirement of the basic LSH method. We have implemented the multi-probe LSH method and evaluated the implementation with two different high-dimensional datasets. Our evaluation shows that the multi-probe LSH method substantially improves upon previously proposed methods in both space and time efficiency. To achieve the same search quality, multi-probe LSH has a similar time-efficiency as the basic LSH method while reducing the number of hash tables by an order of magnitude. In comparison with the entropy-based LSH method, to achieve the same search quality, multi-probe LSH uses less query time and 5 to 8 times fewer number of hash tables.

1. INTRODUCTION

Similarity search in high-dimensional spaces has become increasingly important in databases, data mining, and search engines, particularly for content-based search of feature-rich data such as audio recordings, digital photos, digital videos, and other sensor data. Since feature-rich data objects are typically represented as high-dimensional feature vectors, similarity search is usually implemented as K-Nearest Neighbor (KNN) or Approximate Nearest Neighbors (ANN) search in high-dimensional feature-vector space.

VLDB '07, September 23-28, 2007, Vienna, Austria.

An ideal indexing scheme for similarity search should have the following properties:

- *Accurate:* A query operation should return desired results that are very close to those of the brute-force, linear-scan approach.
- *Time efficient:* A query operation should take $O(1)$ or $O(\log N)$ time where N is the number of data objects in the dataset.
- *Space efficient:* An index should require a very small amount of space, ideally linear in the dataset size, not much larger than the raw data representation. For reasonably large datasets, the index data structure may even fit into main memory.
- *High-dimensional:* The indexing scheme should work well for datasets with very high intrinsic dimensionalities (e.g. on the order of hundreds).

In addition, the construction of the index data structure should be quick and it should deal with various sequences of insertions and deletions conveniently.

Current approaches do not satisfy all of these requirements. Previously proposed tree-based indexing methods for KNN search such as R-tree [14], K-D tree [4], SR-tree [18], navigating-nets [19] and cover-tree [5] return accurate results, but they are not time efficient for data with high (intrinsic) dimensionalities. It has been shown in [27] that when the dimensionality exceeds about 10, existing indexing data structures based on space partitioning are slower than the brute-force, linear-scan approach.

For high-dimensional similarity search, the best-known indexing method is locality sensitive hashing (LSH) [17]. The basic method uses a family of locality-sensitive hash functions to hash nearby objects in the high-dimensional space into the same bucket. To perform a similarity search, the indexing method hashes a query object into a bucket, uses the data objects in the bucket as the candidate set of the results, and then ranks the candidate objects using the distance measure of the similarity search. To achieve high search accuracy, the LSH method needs to use multiple hash tables to produce a good candidate set. Experimental studies show that this basic LSH method needs over a hundred [13] and sometimes several hundred hash tables [6] to achieve good search accuracy for high-dimensional datasets. Since the size of each hash table is proportional to the number of data objects, the basic approach does not satisfy the space-efficiency requirement.

In a recent theoretical study [22], Panigrahy proposed an entropy-based LSH method that generates randomly "perturbed" objects near the query object, queries them in addi-

tion to the query object, and returns the union of all results as the candidate set. The intention of the method is to trade time for space requirements. To explore the practicality of this approach, we have implemented it and conducted an experimental study. We found that although the entropy-based method can reduce the space requirement of the basic LSH method, significant improvements are possible.

This paper presents a new indexing scheme, called *multi-probe* LSH, that satisfies all the requirements of a good similarity indexing scheme. The main idea 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. We have developed and analyzed two schemes to compute the probing sequence: step-wise probing and query-directed probing. By probing multiple buckets in each hash table, the method requires far fewer hash tables than previously proposed LSH methods. By picking the probing sequence carefully, it also requires checking far fewer buckets than entropy-based LSH.

We have implemented the basic LSH, entropy-based LSH, and the multi-probe LSH methods and evaluated them with two datasets. The first dataset contains 1.3 million web images, each represented by a 64-dimensional feature vector. The second is an audio dataset that contains 2.6 million words, each represented by a 192-dimensional feature vector. Our evaluation shows that the multi-probe LSH method substantially improves over the basic and entropy-based LSH methods in both space and time efficiency. To achieve over 0.9 recall, the multi-probe LSH method reduces the number of hash tables of the basic LSH method by a factor of 14 to 18 while achieving similar time efficiencies. In comparison with the entropy-based LSH method, multi-probe LSH reduces the space requirement by a factor of 5 to 8 and uses less query time, while achieving the same search quality.

We emphasize that our focus in this paper is on improving the space and time efficiency of LSH, already established as an attractive technique for high-dimensional similarity search. We compare our new method to previously proposed LSH methods – a detailed comparison with other indexing techniques is outside the scope of this work.

2. SIMILARITY SEARCH PROBLEM

The problem of similarity search refers to finding objects that have similar characteristics to the query object. When data objects are represented by d-dimensional feature vectors, the goal of similarity search for a given query object q, is to find the K objects that are closest to q according to a distance function in the d-dimensional space. The search quality is measured by the fraction of the nearest K objects we are able to retrieve.

In this paper, we also consider the similarity search problem as solving the approximate nearest neighbors problem, where the goal is to find K objects whose distances are within a small factor $(1+\epsilon)$ of the true K-nearest neighbors' distances. With this viewpoint, we also measure search quality by comparing the distances to the query for the K objects retrieved to the corresponding distances of the K nearest objects. Our goal is to design a good indexing method for similarity search of large-scale datasets that can achieve high search quality with high time and space efficiency.

3. LSH INDEXING

The basic idea of locality sensitive hashing (LSH) is to use hash functions that map similar objects into the same hash buckets 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 distances to q. This section provides a brief overview of LSH functions, the basic LSH indexing method and a recently proposed entropy-based LSH indexing method.

3.1 Locality Sensitive Hashing (LSH)

The notion of *locality sensitive hashing* (LSH) was first introduced by Indyk and Motwani in [17]. LSH function families have the property that objects that are close to each other have a higher probability of colliding than objects that are far apart. Specifically, let S be the domain of objects, and D be the distance measure between objects.

DEFINITION 1. *A function family $\mathcal{H} = \{h : S \to U\}$ is called (r, cr, p_1, p_2)-sensitive for D if for any $q, p \in S$*

- *If $D(q,p) \leq r$ then $Pr_{\mathcal{H}}[h(q) = h(p)] \geq p_1$,*
- *If $D(q,p) > cr$ then $Pr_{\mathcal{H}}[h(q) = h(p)] \leq p_2$.*

To use LSH for approximate nearest neighbor search, we pick $c > 1$ and $p_1 > p_2$. With these choices, nearby objects (those within distance r) have a greater chance (p_1 *vs.* p_2) of being hashed to the same value than objects that are far apart (those at a distance greater than cr away).

Different LSH families can be used for different distance functions D. Families for Jaccard measure, Hamming distance, ℓ_1 and ℓ_2 are known [17]. Datar *et al.*[8] have proposed LSH families for l_p norms, based on p-stable distributions [28, 16]. Here, each hash function is defined as:

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{W} \right\rfloor$$

where a is a d-dimensional random vector with entries chosen independently from a p-stable distribution and b is a real number chosen uniformly from the range $[0, W]$. Each hash function $h_{a,b} : \mathbb{R}^d \to \mathbb{Z}$ maps a d-dimensional vector v onto the set of integers. The p-stable distribution used in this work is the Gaussian distribution, which is 2-stable and works for the Euclidean distance.

3.2 Basic LSH Indexing

Using a family of LSH functions $\mathcal{H}$, we can construct indexing data structures for similarity search. The basic LSH indexing method works as follows [17, 13, 8]:

- For an integer M, define a function family $\mathcal{G} = \{g : S \to U^M\}$, and for $g \in \mathcal{G}$, $g(v) = (h_1(v), \ldots, h_M(v))$, where $h_j \in \mathcal{H}$ for $1 \leq j \leq M$ (*i.e.*, g is the concatenation of M LSH functions).
- For an integer L, choose $g_1, \ldots, g_L$ from $\mathcal{G}$, 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[1].

[1]The optimal M and L values depend on the nearest neighbors' distance R. In practice, multiple sets of hash tables are used in order to cover different R values (e.g., $r, 2r, 4r, \ldots$), or use the LSH Forest [3] method.

By concatenating multiple LSH functions, the collision probability of far away objects becomes very small (p_2^M), but it also reduces the collision probability of nearby objects (p_1^M). As a result, multiple hash tables are needed in order to find most of the nearby objects.

LSH-based indices are constructed and maintained using the following three operations:

- *init* (L, M, W) constructs L hash tables, each containing M LSH functions in the form of $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{W} \rfloor$ with randomly chosen a and b.
- *insert* (v) computes the hash values $g_i(v)$ for the i-th hash table and places v into the hash bucket[2] to which $g_i(v)$ points, for $i = 1, \ldots, L$.
- *delete* (v) removes v from the hash bucket that $g_i(v)$ points to in the i-th table, for $i = 1, \ldots, L$.

The basic LSH indexing method processes a similarity search, for a given query q, in two steps. The first step is to generate a candidate set by the union of all buckets that query q is hashed to. The second step ranks the objects in the candidate set according to their distances to query object q, and then returns the top K objects.

The main drawback of the basic LSH indexing method 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 [13], and as many as 583 hash tables are used in [6]. The size of each hash table is proportional to the dataset size, since each table has as many entries as the number of data objects in the dataset. When the space requirement for the hash tables exceeds the main memory size, looking up a hash bucket may require a disk I/O, causing substantial delay to the query process.

3.3 Entropy-Based LSH Indexing

Recent theoretical work by Panigrahy [22] proposed an *entropy-based* LSH scheme, which constructs its indices in a similar manner as the basic scheme, but uses a different query procedure. This scheme works as follows. Assuming we know the distance R_p from the nearest neighbor p to the query q. In principle, for every hash bucket, we can compute the probability that p lies in that hash bucket (call this the success probability of the hash bucket). Note that this distribution depends only on the distance R_p. Given this information, it would make sense to query the hash buckets which have the highest success probabilities. However, performing this calculation is cumbersome. Instead, Panigrahy proposes a clever way to sample buckets from the distribution given by these probabilities. Each time, a random point p' at distance R_p from q is generated and the bucket that p' is hashed to is checked. This ensures that buckets are sampled with exactly the right probabilities. Performing this sampling multiple times will ensure that all the buckets with high success probabilities are probed.

However, this approach has some drawbacks: the sampling process is inefficient because perturbing points and computing their hash values are slow, and it will inevitably generate duplicate buckets. In particular, buckets with high success probability will be generated multiple times and much of the computation is wasteful. Although it is possible to remember all buckets that have been checked previously, the overhead is high when there are many concurrent queries. Further, buckets with small success probabilities will also be generated and this is undesirable. Another drawback is that the sampling process requires knowledge of the nearest neighbor distance R_p, which is difficult to choose in a data-dependent way. If R_p is too small, perturbed queries may not produce the desired number of objects in the candidate set. If R_p is too large, it would require many perturbed queries to achieve good search quality.

We have implemented the entropy-based LSH indexing method. With hand-tuned R_p according to our specific datasets, the entropy-based LSH scheme can reduce the number of hash tables by a factor of 2 or 3, but increases the query time by 30% - 210%. However, this is optimistic result since we did not know how to choose R_p independently from the datasets.

[2]Since the total number of hash buckets may be large, only non-empty buckets are retained using regular hashing.

4. MULTI-PROBE LSH INDEXING

To address the issues associated with the basic and entropy-based LSH methods, we propose a new method called multi-probe LSH, which uses a more systematic approach to explore hash buckets. Ideally, we would like to examine the buckets with the highest success probabilities. We develop a simple approximation for these success probabilities and use it to order the hash buckets for exploration. Moreover, our ordering of hash buckets does not depend on the nearest neighbor distance as in the entropy-based approach. Our experiments demonstrate that our approximation works quite well. In using this technique, we are able to achieve high recall with substantially fewer hash tables. It is plausible that Panigrahy's analysis of entropy-based LSH can be adapted to give theoretical bounds on the performance of our multi-probe LSH scheme.

4.1 Algorithm Overview

The key 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. Given the property of locality sensitive hashing, we know that if an object is close to a query object q but not hashed to the same bucket as q, it is likely to be in a bucket that is "close by" (*i.e.*, the hash values of the two buckets only differ slightly). So our goal is to locate these "close by" buckets, thus increasing the chance of finding the objects that are close to q.

We define a *hash perturbation vector* to be a vector $\Delta = (\delta_1, \ldots, \delta_M)$. Given a query q, the basic LSH method checks the hash bucket $g(q) = (h_1(q), \ldots, h_M(q))$. When we apply the perturbation Δ, we will probe the hash bucket $g(q) + \Delta$.

Recall that the LSH functions we use are of the form $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{W} \rfloor$. If we pick W to be reasonably large, with high probability, similar objects should hash to the same or adjacent values (*i.e.* differ by at most 1). Hence we restrict our attention to perturbation vectors Δ with $\delta_i \in \{-1, 0, 1\}$.

Each perturbation vector is directly applied to the hash values of the query object, thus avoiding the overhead of point perturbation and hash value computations associated with the entropy-based LSH method. We will design a sequence of perturbation vectors such that each vector in this sequence maps to a unique set of hash values so that we never probe a hash bucket more than once.

Figure 1 shows how the multi-probe LSH method works. In the figure, $g_i(q)$ is the hash value of query q in the i-th table, $(\Delta_1, \Delta_2, \ldots)$ is a probing sequence, and $g_i(q) + \Delta_1$ is

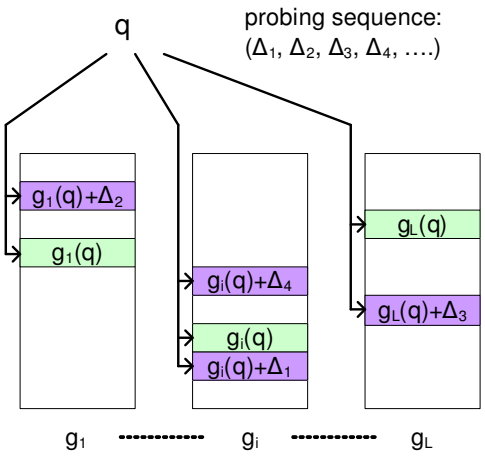

Figure 1: Multi-probe LSH uses a sequence of hash perturbation vectors to probe multiple hash buckets. $g_i(q)$ is the hash value of query q in the i-th table. Δ_j is a hash perturbation vector.

the new hash value after applying perturbation vector Δ_1 to $g_i(q)$; it points to another hash bucket in the table. By using multiple perturbation vectors we locate more hash buckets which are likely to be close to the query object's buckets and may contain q's nearest neighbors. Next, we address the issue of generating a sequence of perturbation vectors.

4.2 Step-Wise Probing

An n-step perturbation vector Δ has exactly n coordinates that are non-zero. This corresponds to probing a hash bucket which differs in n coordinates from the hash bucket of the query. Based on the property of locality sensitive hashing, buckets that are one step away (*i.e.*, only one hash value is different from the M hash values of the query object) are more likely to contain objects that are close to the query object than buckets that are two steps away.

This motivates the *step-wise* probing method, which first probes all the 1-step buckets, then all the 2-step buckets, and so on. For an LSH index with L hash tables and M hash functions per table, the total number of n-step buckets is $L \times \binom{M}{n} \times 2^n$ and the total number of buckets within s steps is $L \times \sum_{n=1}^{s} \binom{M}{n} \times 2^n$.

Figure 2 shows the distribution of bucket distances of K nearest neighbors. The plot on the left shows the difference of a single hash value (δ_i) and the plot on the right shows the number of hash values (out of M) that differ from the hash values of the query object (n-step buckets). As we can see from the plots, almost all of the individual hash values of the K nearest neighbors are either the same ($\delta_i = 0$) as that of the query object or differ by just -1 or $+1$. Also, most K nearest neighbors are hashed to buckets that are within 2 steps of the hashed bucket of the query object.

4.3 Success Probability Estimation

Using the step-wise probing method, all coordinates in the hash values of the query q are treated identically, *i.e.*, all have the same chance of being perturbed, and we consider both the possibility of adding 1 and subtracting 1 from each coordinate to be equally likely. In fact, a more refined construction of a probing sequence is possible by considering how the hash value of q is computed. Note that each hash function $h_{a,b}(q) = \lfloor \frac{a \cdot q + b}{W} \rfloor$ first maps q to a line. The line is divided into slots (intervals) of length W numbered from left to right and the hash value is the number of the slot that q falls into. A point p close to q is likely to fall in either the same slot as q or an adjacent slot. In fact, the probability that p falls into the slot to the right (left) of q depends on how close q is to the right (left) boundary of its slot. Thus the position of q within its slot for each of the M hash functions is potentially useful in determining perturbations worth considering. Next, we describe a more sophisticated method to construct a probing sequence that takes advantage of such information. We mention that the idea of considering the position of q within its slot for each hash function originated in Panigrahy's analysis for his entropy-based LSH scheme.

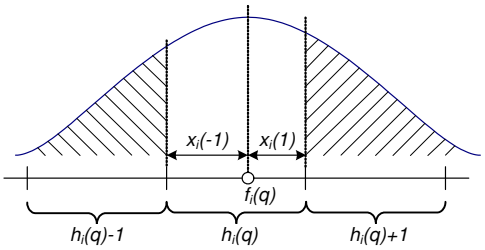

Figure 3: Probability of q's nearest neighbors falling into the neighboring slots.

Figure 3 illustrates the probability of q's nearest neighbors falling into the neighboring slots. Here, $f_i(q) = a_i \cdot q + b_i$ is the projection of query q on to the line for the i-th hash function and $h_i(q) = \lfloor \frac{a_i \cdot q + b_i}{W} \rfloor$ is the slot to which q is hashed. For $\delta \in \{-1, +1\}$, let $x_i(\delta)$ be the distance of q from the boundary of the slot $h_i(q) + \delta$, then $x_i(-1) = f_i(q) - h_i(q) \times W$ and $x_i(1) = W - x_i(-1)$. For convenience, define $x_i(0) = 0$. For any fixed point p, $f_i(p) - f_i(q)$ is a Gaussian random variable with mean 0 (here the probability distribution is over the random choices of a_i). The variance of this random variable is proportional to $\|p - q\|_2^2$. We assume that W is chosen to be large enough so that for all points p of interest, p falls with high probability in one of the three slots numbered $h_i(q)$, $h_i(q) - 1$ or $h_i(q) + 1$. Note that the probability density function of a Gaussian random variable is $e^{-x^2/2\sigma^2}$ (scaled by a normalizing constant). Thus the probability that point p falls into slot $h_i(q) + \delta$ can be estimated by:

$$Pr[h_i(p) = h_i(q) + \delta] \approx e^{-C x_i(\delta)^2}$$

where C is a constant depending on $\|p - q\|_2$.

We now estimate the success probability (finding a p that is close to q) of a perturbation vector $\Delta = (\delta_1, \ldots, \delta_M)$.

$$\begin{aligned} Pr[g(p) = g(q) + \Delta] &= \prod_{i=1}^{M} Pr[h_i(p) = h_i(q) + \delta_i] \\ &\approx \prod_{i=1}^{M} e^{-C x_i(\delta_i)^2} = e^{-C \sum_i x_i((\delta_i)^2)} \end{aligned}$$

This suggests that the likelihood that perturbation vector

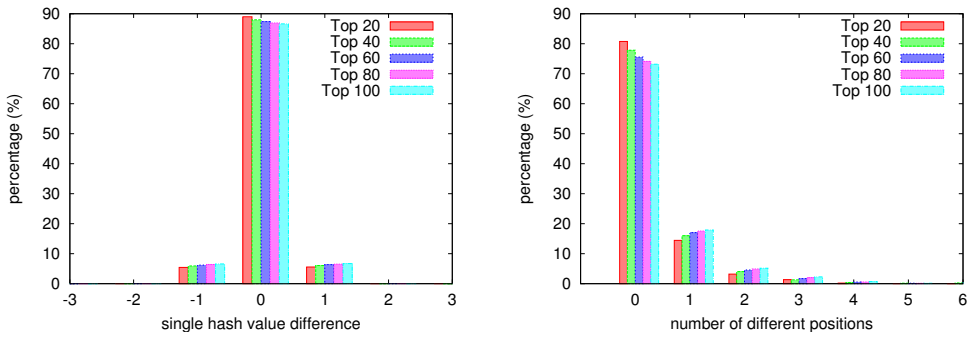

Figure 2: Bucket distance distribution of K nearest neighbors. Image dataset. $W = 0.7, M = 16, L = 15$.

Δ will find a point close to q is related to

$$score(\Delta) = \sum_{i=1}^{M} x_i(\delta_i)^2$$

Perturbation vectors with smaller scores have higher probability of yielding points near to q. Note that the score of Δ is a function of both Δ and the query q. This is the basis for our new *query-directed* probing method, which orders perturbation vectors in increasing order of their (query dependent) scores.

4.4 Query-Directed Probing Sequence

A naive way to construct the probing sequence would be to compute scores for all possible perturbation vectors and sort them. However, there are $L \times (2^M - 1)$ perturbation vectors and we expect to actually use only a small fraction of them. Thus explicitly generating all perturbation vectors seems unnecessarily wasteful. We describe a more efficient way to generate perturbation vectors in increasing order of their scores.

First note that the score of a perturbation vector Δ depends only on the non-zero coordinates of Δ (since $x_i(\delta) = 0$ for $\delta = 0$). We expect that perturbation vectors with low scores will have a few non-zero coordinates. In generating perturbation vectors, we will represent only the non-zero coordinates as a set of (i, δ_i) pairs. An (i, δ) pair represents adding δ to the i-th hash value of q.

Given the query object q and the hash functions h_i for $i = 1, \ldots, M$ corresponding to a single hash table, we first compute $x_i(\delta)$ for $i = 1, \ldots, M$ and $\delta \in \{-1, +1\}$. We sort these $2M$ values in increasing order. Let z_j denote the jth element in this sorted order. Let $\pi_j = (i, \delta)$ if $z_j = x_i(\delta)$. This represents the fact that the value $x_i(\delta)$ is the jth smallest in the sorted order. Note that since $x_i(-1) + x_i(+1) = W$, if $\pi_j = (i, \delta)$, then $\pi_{2M+1-j} = (i, -\delta)$. We now represent perturbation vectors as a subset of $\{1, \ldots, 2M\}$, referred to as a perturbation set[3]. For each such perturbation set A, the corresponding perturbation vector Δ_A is obtained by taking the set of coordinate perturbations $\{\pi_j | j \in A\}$. Every perturbation set A can be associated with a score $score(A) = \sum_{j \in A} z_j^2$, which is exactly the same as the score of the corresponding perturbation vector Δ_A. Given the sorted order π of (i, δ_i) pairs and the values $z_j, j = 1, \ldots, 2M$, the problem of generating perturbation vectors now reduces to the problem of generating perturbation sets in increasing order of their scores.

[3]Each perturbation set corresponds to one perturbation vector, while a probing sequence contains multiple perturbation vectors.

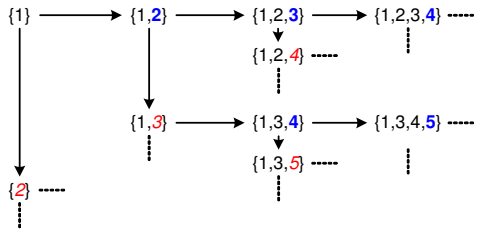

Figure 4: Generate perturbation sequences. Vertical arrows represent *shift* operations, and horizontal arrows represent *expand* operations.

Algorithm 1 Generate T perturbation sets

```
A_0 = {1}
minHeap_insert(A_0, score(A_0))

for i = 1 to T do
  repeat
    A_i = minHeap_extractMin()
    A_s = shift(A_i)
    minHeap_insert(A_s, score(A_s))
    A_e = expand(A)
    minHeap_insert(A_e, score(A_e))
  until valid(A_i)
  output A_i
end for
```

We define two operations on a perturbation set:

- $shift(A)$: This operation replaces $\max(A)$ by $1+\max(A)$. *E.g.* $shift(\{1, 3, 4\}) = \{1, 3, 5\}$.
- $expand(A)$: This operation adds the element $1 + \max(A)$ to the set A. *E.g.* $expand(\{1, 3, 4\}) = \{1, 3, 4, 5\}$.

Algorithm 1 shows how to generate the first T perturbation sets. A min-heap is used to maintain the collection of candidate perturbation sets such that the score of a parent set is not larger than the score of its child set. The heap is initialized with the set $\{1\}$. Each time we remove the top node (set A_i) and generate two new sets $shift(A_i)$ and $expand(A_i)$ (see Figure 4). Only the valid top node (set A_i) is output. Note, for every $j = 1, \ldots, M$, π_j and π_{2M+1-j}

represent opposite perturbations on the same coordinate. Thus, a valid perturbation set A must have at most one of the two elements $\{j, 2M+1-j\}$ for every j. We also consider any perturbation set containing value greater than 2M to be invalid.

We mention two properties of the *shift* and *expand* operations which are important for establishing the correctness of the above procedure.

1. For a perturbation set A, the scores for $shift(A)$ and $expand(A)$ are greater than the score for A.
2. For any perturbation set A, there is a unique sequence of *shift* and *expand* operations which will generate the set A starting from $\{1\}$.

Based on these two properties, it is easy to establish the following correctness property by induction on the sorted order of the sets (by score).

CLAIM 1. *The procedure described correctly generates all valid perturbation sets in increasing order of their score.*

CLAIM 2. *The number of elements in the heap at any point of time is one more than the number of min-heap_extract-min operations performed.*

To simplify the exposition, we have described the process of generating perturbation sets for a single hash table. In fact, we will need to generate perturbation sets for each of the L hash tables. For each hash table, we maintain a separate sorted order of (i, δ) pairs and z_j values, represented by π_j^t and z_j^t respectively. However we can maintain a single heap to generate the perturbation sets for all tables simultaneously. Each candidate perturbation set in the heap is associated with a table t. Initially we have L copies of the set $\{1\}$, each associated with a different table. For a perturbation set A for table t, the score is a function of the z_j^t values and the corresponding perturbation vector Δ_A is a function of the π_j^t values. When set A associated with table t is removed from the heap, the newly generated sets $shift(A)$ and $expand(A)$ are also associated with table t.

4.5 Optimized Probing Sequence Construction

The query-directed probing approach described above generates the sequence of perturbation vectors at query time by maintaining a heap and querying this heap repeatedly. We now describe a method to avoid the overhead of maintaining and querying such a heap at query time. In order to do this, we precompute a certain sequence and reduce the generation of perturbation vectors to performing lookups instead of heap queries and updates.

Note that the generation of the sequence of perturbation vectors can be separated into two parts: (1) generating the sorted order of perturbation sets, and (2) mapping each perturbation set into a perturbation vector. The first part requires the z_j values while the second part requires the mapping π from $\{1, \ldots, 2M\}$ to (i, δ) pairs. Both these are functions of the query q.

As we will explain shortly, it turns out that we know the distribution of the z_j values precisely and can compute $E[z_j^2]$ for each j. This motivates the following optimization: We approximate the z_j^2 values by their expectations. Using this approximation, the sorted order of perturbation sets can be precomputed (since the score of a set is a function of the z_j^2 values). The generation process is exactly the same as described in the previous subsection, but uses the $E[z_j^2]$ values instead of their actual values. This can be done independently of the query q. At query time, we compute the mapping π_j^t as a function of query q (separately for each hash table t). These mappings are used to convert each perturbation set in the precomputed order into L perturbation vectors, one for each of the L hash tables. This precomputation reduces the query time overhead of dynamically generating the perturbation sets at query time.

To complete the description, we need to explain how to obtain $E[z_j^2]$. Recall that the z_j values are the $x_i(\delta)$ values in sorted order. Note $x_i(\delta)$ is uniformly distributed in $[0, W]$ and further $x_i(-1) + x_i(+1) = W$. Since each of the M hash functions is chosen independently, the $x_i(\delta)$ values are independent of the $x_j(\delta')$ values for $j \neq i$. The joint distribution of the z_j values for $j = 1, \ldots, M$ is then the following: pick M numbers uniformly and at random from the interval $[0, W/2]$. z_j is the j-th largest number in this set. This is a well studied distribution, referred to as the order statistics of the uniform distribution in $[0, W]$. Using known facts about this distribution, we get that for $j \in \{1, \ldots, M\}$, $E[z_j] = \frac{j}{2(M+1)}W$ and $E[z_j^2] = \frac{j(j+1)}{4(M+1)(M+2)}W^2$. Further, for $j \in \{M+1, \ldots, 2M\}$, $E[z_j^2] = E[(W - z_{2M+1-j})^2] = W^2\left(1 - \frac{2M+1-j}{M+1} + \frac{(2M+1-j)(2M+2-j)}{4(M+1)(M+2)}\right)$. These values are used in determining the precomputed order of perturbation sets as described earlier.

5. EXPERIMENTAL SETUP

This section describes the configurations of our experiments, including the evaluation datasets, evaluation benchmarks, evaluation metrics, and some implementation details.

5.1 Evaluation Datasets

We have used two datasets in our evaluation. The dataset sizes are chosen such that the index data structure of the basic LSH method can entirely fit into the main memory. Since the entropy-based and multi-probe LSH methods require less memory than the basic LSH method, we will be able to compare the in-memory indexing behaviors of all three approaches. The two datasets are:

Image Data: The image dataset is obtained from Stanford's WebBase project [24], which contains images crawled from the web. We only picked images that are of JPEG format and are larger than 64×64 in size. The total number of images picked is 1.3 million. For each image, we use the *extractcolorhistogram* tool from the FIRE image search engine [11, 9] to extract a 64-dimensional color histogram.

Audio Data: The audio dataset comes from the LDC SWITCHBOARD-1 [25] collection. It is a collection of about 2400 two-sided telephone conversations among 543 speakers from all areas of the United States. The conversations are split into individual words based on the human transcription. In total, the audio dataset contains 2.6 million words. For each word segment, we then use the Marsyas library [26] to extract feature vectors by taking a 512-sample sliding window with variable stride to obtain 32 windows for each word. For each of the 32 windows, we extract the first six MFCC parameters, resulting in a 192-dimensional feature vector for each word.

Table 1 summarizes the number of objects in each dataset

Dataset	#Objects	#Dimension	Total Size
Image	1,312,581	64	336 MB
Audio	2,663,040	192	2.0 GB

Table 1: Evaluation Datasets.

and the dimensionality of the feature vectors.

5.2 Evaluation Benchmarks

For each dataset, we created an evaluation benchmark by randomly picking 100 objects as the query objects, and for each query object, the ground truth (*i.e.*, the ideal answer) is defined to be the query object's K nearest neighbors (not including the query object itself), based on the Euclidean distance of their feature vectors. Unless otherwise specified, K is 20 in our experiments.

5.3 Evaluation Metrics

The performance of a similarity search system can be measured in three aspects: search quality, search speed, and space requirement. Ideally, a similarity search system should be able to achieve high-quality search with high speed, while using a small amount of space.

Search quality is measured by *recall*. Given a query object q, let $I(q)$ be the set of ideal answers (*i.e.*, the k nearest neighbors of q), let $A(q)$ be the set of actual answers, then

$$recall = \frac{|A(q) \cap I(q)|}{|I(q)|}$$

In the ideal case, the recall score is 1.0, which means all the k nearest neighbors are returned. Note that we do not need to consider precision here, since all the candidate objects (*i.e.*, objects found in one of the checked hash buckets) will be ranked based on their Euclidean distances to the query object and only the top k candidates will be returned.

For comparison purposes, we will also present search quality results in terms of *error ratio* (or effective error), which measures the quality of approximate nearest neighbor search. As defined in [13]:

$$\text{error ratio} = \frac{1}{|Q|K} \sum_{q \in Q} \sum_{k=1}^{K} \frac{d_{LSH_k}}{d_k^*}$$

where d_{LSH_k} is the k-th nearest neighbor found by a LSH method, and d_k^* is the true k-th nearest neighbor. In other words, it measures how close the distances of the K nearest neighbors found by LSH are compared to the exact K nearest neighbors' distances.

Search speed is measured by query time, which is the time spent to answer a query. Space requirement is measured by the total number of hash tables needed, and the total memory usage.

All performance measures are averaged over the 100 queries. Also, since the hash functions are randomly picked, each experiment is repeated 10 times and the average is reported.

5.4 Implementation Details

We have implemented the three different LSH methods as discussed in previous sections: *basic, entropy*, and *multi-probe*. For the multi-probe LSH method, we have implemented both step-wise probing and query-directed probing. The default probing method for multi-probe LSH is query-directed probing. For all the hash tables, only the object ids are stored in the hash buckets. A separate data structure stores all the vectors, which can be accessed via object ids. We use an object id bitmap to efficiently union objects found in different hash buckets. As a baseline comparison, we have also implemented the *brute-force* method, which linearly scans through all the feature vectors to find the k nearest objects. All methods are implemented using the C programming language. Also, each method reads all the feature vectors into main memory at startup time.

We have experimented with different parameter values for the LSH methods and picked the ones that give best performance. In the results, unless otherwise specified, the default values are $W = 0.7, M = 16$ for the image dataset and $W = 24.0, M = 11$ for the audio dataset. For the entropy-based LSH method, the perturbation distance $R_p = 0.04$ for the image dataset and $R_p = 4.0$ for the audio dataset.

The evaluation is done on a PC with one dual-processor Intel Xeon 3.2GHz CPU with 1024KB L2 cache. The PC system has 6GB of DRAM and a 160GB 7,200RPM SATA disk. It runs the Linux operating system with a 2.6.9 kernel.

6. EXPERIMENTAL RESULTS

In this section, we report the evaluation results of the three LSH methods using the image dataset and the audio dataset. We are interested in answering the question about the space requirements, search time and search quality trade-offs for different LSH methods.

6.1 Main Results

The main result is that the multi-probe LSH method is much more space efficient than the basic LSH and entropy-based LSH methods to achieve various search quality levels and it is more time efficient than the entropy-based LSH method.

Table 2 shows the average results of the basic LSH, entropy-based LSH and multi-probe LSH methods using 100 random queries with the image dataset and the audio dataset. We have experimented with different number of hash tables L (for all three LSH methods) and different number of probes T (*i.e.*, number of extra hash buckets to check, for the multi-probe LSH method and the entropy-based LSH method). For each dataset, the table reports the query time, the error ratio and the number of hash tables required, to achieve three different search quality (recall) values. .

The results show that the multi-probe LSH method is significantly more space efficient than the basic LSH method. For both the image data set and the audio data set, the multi-probe LSH method reduces the number of hash tables by a factor of 14 to 18. In all cases, the multi-probe LSH method has similar query time to the basic LSH method.

The space efficiency implication is dramatic. Since each hash table entry consumes about 16 bytes in our implementation, 2 gigabytes of main memory can hold the index data structure of the basic LSH method for about 4-million images to achieve a 0.93 recall. On the other hand, when the same amount of main memory is used by the multi-probe LSH indexing data structures, it can deal with about 60-million images to achieve the same search quality.

The results in Table 2 also show that the multi-probe LSH method is substantially more space and time efficient than the entropy-based approach. For the image dataset, the

recall	method	error ratio	query time (s)	#hash tables	space ratio
0.96	basic	1.027	0.049	44	14.7
	entropy	1.023	0.094	21	7.0
	multi-probe	1.015	0.050	3	1.0
0.93	basic	1.036	0.044	30	15.0
	entropy	1.044	0.092	11	5.5
	multi-probe	1.053	0.039	2	1.0
0.90	basic	1.049	0.029	18	18.0
	entropy	1.036	0.078	6	6.0
	multi-probe	1.029	0.031	1	1.0

(a) image dataset

recall	method	error ratio	query time (s)	#hash tables	space ratio
0.94	basic	1.002	0.191	69	13.8
	entropy	1.002	0.242	44	8.8
	multi-probe	1.002	0.199	5	1.0
0.92	basic	1.003	0.174	61	15.3
	entropy	1.003	0.203	25	6.3
	multi-probe	1.002	0.163	4	1.0
0.90	basic	1.004	0.133	49	16.3
	entropy	1.003	0.181	19	6.3
	multi-probe	1.003	0.143	3	1.0

(b) audio dataset

Table 2: Search performance comparison of different LSH methods: multi-probe LSH is most efficient in terms of space usage and time while achieving the same recall score as other LSH methods.

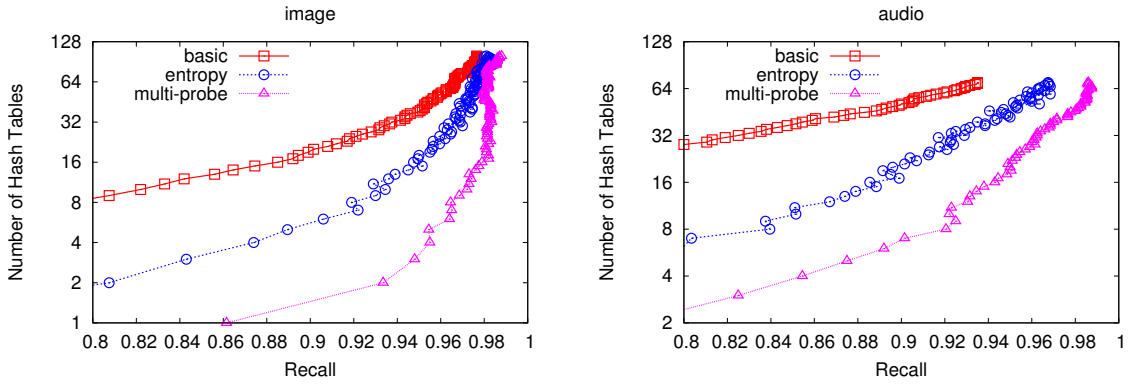

Figure 5: Number of hash tables (in log scale) required by different LSH methods to achieve certain search quality ($T = 100$ for both multi-probe LSH and entropy-based LSH): multi-probe LSH achieves higher recall with fewer number of hash tables.

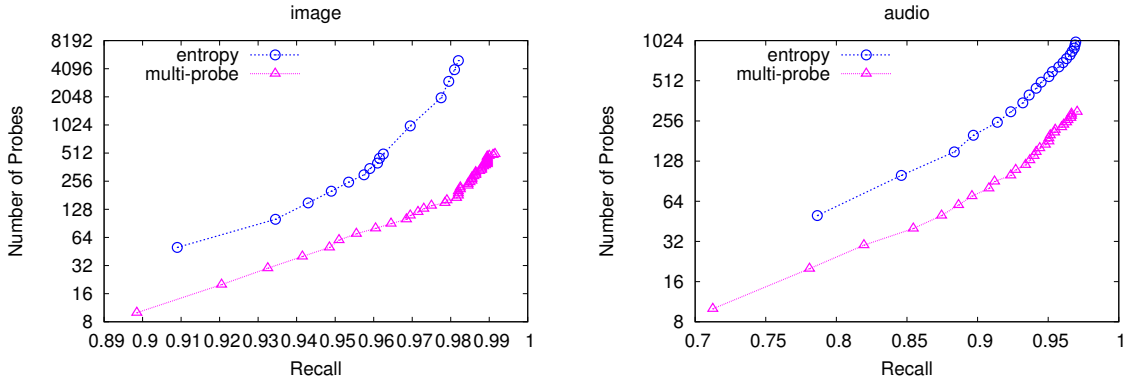

Figure 6: Number of probes (in log scale) required by multi-probe LSH and entropy-based LSH to achieve certain search quality ($L = 10$ for both image and audio): multi-probe LSH method uses much fewer number of probes.

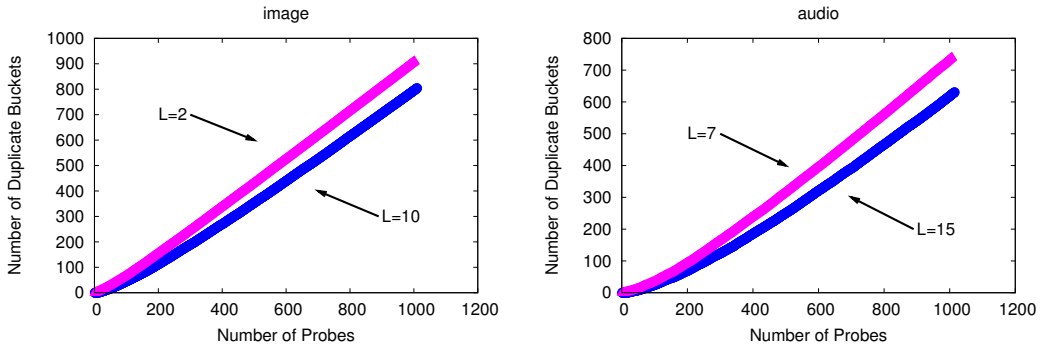

Figure 7: Number of duplicate buckets checked by the entropy-based LSH method: a large fraction of buckets checked by entropy-based LSH are duplicate buckets, especially for smaller L.

multi-probe LSH method reduces the number of hash tables required by the entropy-based approach by a factor of 7.0, 5.5, and 6.0 respectively for the three recall values, while reducing the query time by half. For the audio data set, multi-probe LSH reduces the number of hash tables by a factor of 8.8, 6.3, and 6.3 for the three recall values, while using less query time.

Figure 5 shows the detailed relationship between search quality and the number of hash tables for all three indexing approaches. Here, for easier comparison, we use the same number of probes ($T = 100$) for both multi-probe LSH and entropy-based LSH. It shows that for most recall values, the multi-probe LSH method reduces the number of hash tables required by the basic LSH method by an order of magnitude. It also shows that the multi-probe method is better than the entropy-based LSH method by a significant factor.

6.2 Multi-Probe vs. Entropy-Based Methods

Although both multi-probe and entropy-based methods visit multiple buckets for each hash table, they are very different in terms of how they probe multiple buckets. The entropy-based LSH method generates randomly perturbed objects and use LSH functions to hash them to buckets, whereas the multi-probe LSH method uses a carefully derived probing sequence based on the hash values of the query object. The entropy-based LSH method is likely to probe previously visited buckets, whereas the multi-probe LSH method always visits new buckets.

To compare the two approaches in detail, we are interested in answering two questions. First, when using the same number of hash tables, how many probes does the multi-probe LSH method need, compared with the entropy-based approach? As we can see in Figure 6 (note that the y axis is in log scale of 2), multi-probe LSH requires substantially fewer number of probes.

Second, how often does the entropy-based approach probe previously visited buckets (duplicate buckets)? As we can see in Figure 7, the number of duplicate buckets is over 900 for the image dataset and over 700 for the audio dataset, while the total number of buckets checked is 1000. Such redundancy becomes worse with fewer hash tables.

6.3 Query-Directed vs. Step-Wise Probing

This subsection presents the experimental results of the differences between the query-directed and step-wise probing sequences for the multi-probe LSH indexing method. The results show that query-directed probing sequence is far superior to the step-wise probing sequence.

First, with similar query times, the query-directed probing sequence requires significantly fewer hash tables than the step-wise probing sequence. Table 3 shows the space requirements of using the two probing sequences to achieve three recall precisions with similar query times. For the image dataset, the query-directed probing sequence reduces the number of hash tables by a factor of 5, 10 and 10 for the three cases. For the audio dataset, it reduces the number of hash tables by a factor of 5 for all three cases.

Second, with the same number of hash tables, the query-directed probing sequence requires far fewer probes than the step-wise probing sequence to achieve the same recall precisions. Figure 8 shows the relationship between the number of probes and recall precisions for both approaches when they use the same number of hash tables (10 for image data and 15 for audio data). The results indicate that the query-directed probing sequence can reduce the number of probes typically by an order of magnitude for various recall values.

The main reason for the big gap between the two sequences is that many similar objects are not in the buckets 1-step away from the hashed buckets. In fact, some are several steps away from the hashed buckets. The step-wise probing visits all 1-step buckets, then all 2-step buckets, and so on. The query-directed probing visits buckets with high success probability first. Figure 9 shows the number of n-step (n = 1, 2, 3, 4) buckets picked by the query-directed probing method, as a function of the total number of probes. The figure clearly shows that many 2,3,4-step buckets are picked before all the 1-step buckets are picked. For example, for the image dataset, of the first 200 probes, the number of 1-step, 2-step, 3-step and 4-step probes is 50, 90, 50, and 10, respectively.

6.4 Sensitivity Results

By probing multiple hash buckets per table, the multi-probe LSH method can greatly reduce the number of hash tables while finding desired similar objects. A sensitivity question is whether this approach generates a larger candidate set than the other approaches or not. Table 4 shows the ratio of the average candidate set size to the dataset size for the cases in Table 2. The result shows that the multi-probe LSH approach has similar ratios to the basic

	#probes	recall	error ratio	query time(s)	#hash tables
1-step	320	0.933	1.027	0.042	10
query-directed	400	0.937	1.020	0.040	1
1,2-step	5120	0.960	1.017	0.071	10
query-directed	450	0.960	1.024	0.060	2
1,2,3-step	49920	0.969	1.012	0.132	10
query-directed	600	0.969	1.019	0.064	2

(a) image dataset

	#probes	recall	error ratio	query time(s)	#hash tables
1-step	330	0.885	1.004	0.224	15
query-directed	160	0.885	1.004	0.193	3
1,2-step	3630	0.947	1.001	0.462	15
query-directed	450	0.947	1.001	0.323	3
1,2,3-step	23430	0.973	1.001	0.724	15
query-directed	900	0.974	1.001	0.444	3

(b) audio dataset

Table 3: Query-directed probing vs. step-wise probing in multi-probe LSH: query-directed probing uses fewer number of hash tables, shorter query time to achieve the same search quality as step-wise probing.

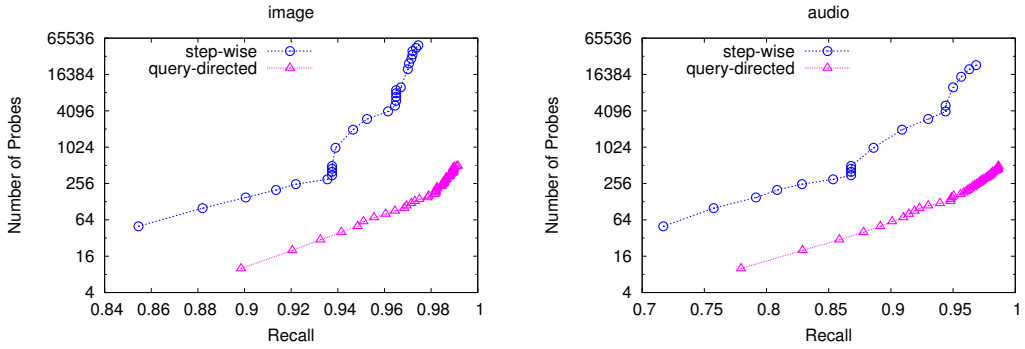

Figure 8: Number of probes required (in log scale) using step-wise probing and query-directed probing to achieve certain search quality: query-directed probing requires substantially fewer number of probes.

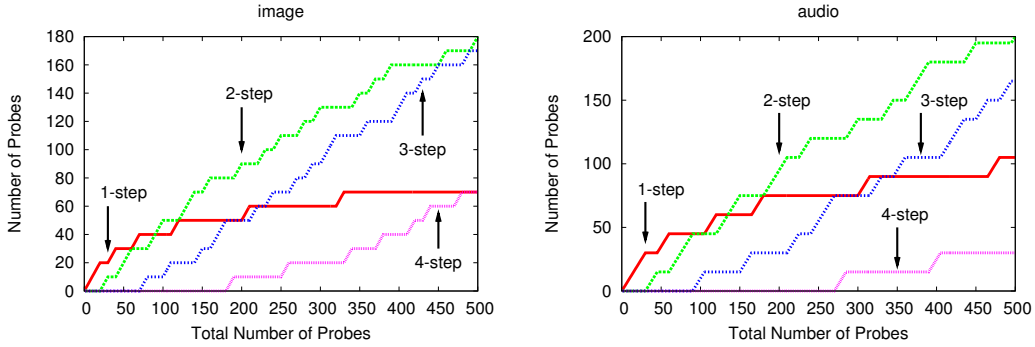

Figure 9: Number of n-step perturbation sequences picked by query-directed probing: many 2,3,4-step sequences are picked before all 1-step sequences are picked.

and entropy-based LSH approaches[4].

In all experiments presented above, we have used $K = 20$ (number of nearest neighbors). Another sensitivity question is whether the search quality of the multi-probe LSH method is sensitive to different K values. Figure 10 shows that the search quality is not so sensitive to different K values. For the image dataset, there are some differences with different K values when the number of probes is small. As the number of probes increases, the sensitivity reduces. For the audio dataset, the multi-probe LSH achieves similar search qualities for different K values.

We suspect that the different sensitivity results in the two datasets are due to the characteristics of the datasets. As shown in Table 2, for the image data, a 0.90 recall corresponds to a 1.049 error ratio, while for the audio data, the same 0.90 recall corresponds to a 1.004 error ratio. This means that the audio objects are much more densely populated in the high-dimensional space. In other words, if a query object q's nearest neighbor is at distance r, there are many objects that lie within cr distance from q. This makes the approximate nearest neighbor search problem easier, but makes high recall values more difficult. However, for a given K, the multi-probe LSH method can effectively reduce the space requirement while achieving desired search quality with more probes.

[4]Since multi-probe LSH achieves higher search quality with fewer hash tables, it can use "tighter" buckets, thus reducing the candidate set size even further.

7. RELATED WORK

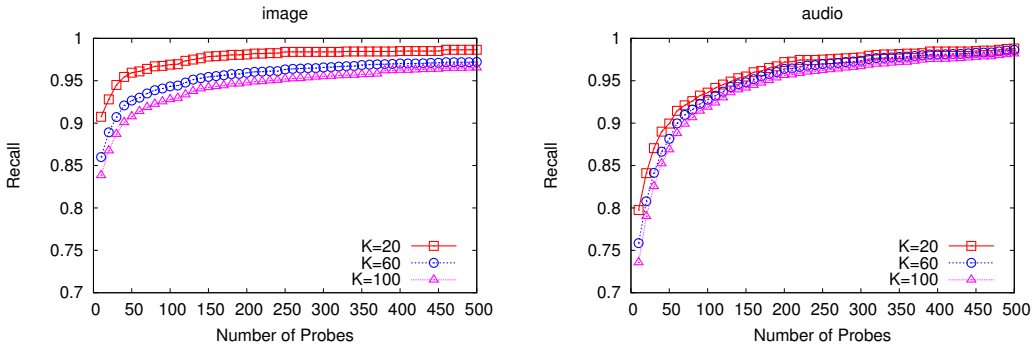

Figure 10: Recall of multi-probe LSH for different K (number of nearest neighbors): multi-probe LSH achieves similar search quality for different K values.

method	image		audio	
	recall	C/N (%)	recall	C/N (%)
basic	0.96	4.4	0.94	6.3
entropy	0.96	4.9	0.94	6.8
multi-probe	0.96	5.1	0.94	7.1
basic	0.93	3.3	0.92	5.7
entropy	0.93	3.9	0.92	5.9
multi-probe	0.93	4.1	0.92	6.0
basic	0.90	2.6	0.90	5.0
entropy	0.90	3.1	0.90	5.6
multi-probe	0.90	3.0	0.90	5.3

Table 4: Percentage of objects examined using different LSH methods (C is candidate set size, N is dataset size): multi-probe LSH has similar filter ratio as other LSH methods.

The similarity search problem is closely related to the nearest neighbor search problem, which has been studied extensively. A number of indexing data structures have been devised for nearest neighbor search; examples include R-tree [14], K-D tree [4], and SR-tree [18]. These data structures are capable of supporting similarity queries, but do not scale satisfactorily to large, high-dimensional datasets. The exact nearest neighbor problem suffers from the "curse of dimensionality" – *i.e.* either the search time or the search space is exponential in the number of dimensions, d [10, 20]. Several approximation-based indexing techniques have been proposed in the literature, such as VA-file [27], A-tree [23], and AV-tree [2]. These techniques use vector approximations or bounding rectangle approximations to prune search space. Much progress has been made on solving the Approximate Nearest-Neighbor (ANN) problem. The objective is to find points whose distance from the query point is at most $1 + \epsilon$ times the exact nearest neighbor's distance. Due to the limited space, we can not give an extensive review of previous searching and indexing techniques. Please see [7, 15, 12] for some survey. Here, we focus on locality sensitive hashing techniques that are most relevant to our work.

Locality sensitive hashing (LSH), introduced by Indyk and Motwani, is the best-known indexing method for ANN search. Theoretical lower bounds for LSH have also been studied [21, 1]. The basic LSH indexing method [17] only checks the buckets to which the query object is hashed and usually requires a large number of hash tables (hundreds) to achieve good search quality. Bawa *et al.* proposed the LSH Forest indexing method [3] which represents each hash table by a prefix tree so the number of hash functions per table can be adapted for different approximation distances. However, this method does not help reduce the number of hash tables for a given approximation distance. In a theoretical study, Panigrahy recently proposed an entropy-based LSH scheme [22], which tries to reduce the number of hash tables by using multiple perturbed queries. In practice, it is difficult to generate perturbed queries in a data-independent way and most hashed buckets by the perturbed queries are redundant. The multi-probe LSH method proposed in this paper is inspired by but quite different from the entropy-based LSH method. Instead of generating perturbed queries, our method computes a non-overlapped bucket sequence, according to the probability of containing similar objects.

8. CONCLUSIONS

This paper presents the multi-probe LSH indexing method for high-dimensional similarity search, which uses carefully derived probing sequences to probe multiple hash buckets in a systematic way. Our experimental results show that the multi-probe LSH method is much more space efficient than the basic LSH and entropy-based LSH methods to achieve desired search accuracy and query time. The multi-probe LSH method reduces the number of hash tables of the basic LSH method by a factor of 14 to 18 and reduces that of the entropy-based approach by a factor of 5 to 8.

We have also shown that although both multi-probe and entropy-based LSH methods trade time for space, the multi-probe LSH method is much more time efficient when both approaches use the same number of hash tables. Our experiments show that the multi-probe LSH method can use ten times fewer number of probes than the entropy-based approach to achieve the same search quality.

We have developed two probing sequences for the multi-probe LSH method. Our results show that the query-directed probing sequence is far superior to the simple, step-wise sequence. By estimating success probability, the query-directed probing sequence typically uses an order-of-magnitude fewer probes than the step-wise probing approach. Although the analysis presented in this paper is for a specific LSH function family, the general technique applies to other LSH function families as well.

This paper focuses on comparing the basic, entropy-based

and multi-probe LSH methods in the case that the index data structure fits in main memory. Our results indicate that 2GB memory will be able to hold a multi-probe LSH index for 60 million image data objects, since the multi-probe method is very space efficient. However, since our dataset sizes in the experiments are chosen to fit the index data structure of each of the three methods (basic, entropy-based and multi-probe) into main memory, we have not experimented the multi-probe LSH indexing method with a 60-million image dataset. For even larger datasets, an out-of-core implementation of the multi-probe LSH method may be worth investigating. Although the multi-probe LSH method can use the LSH forest method to represent its hash table data structure to exploit its self-tuning features, our implementation in this paper uses the basic LSH data structure for simplicity. A comparison of multi-probe LSH and other indexing techniques would also be helpful. We plan to study these issues in the near future.

Acknowledgments

This work is supported in part by NSF grants EIA-0101247, CCR-0205594, CCR-0237113, CNS-0509447, DMS-0528414 and by research grants from Google, Intel, Microsoft, and Yahoo!. William Josephson is supported by a National Science Foundation Graduate Research Fellowship.

9. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for near neighbor problem in high dimensions. In *Proc. of the 47th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2006.

[2] S. Balko, I. Schmitt, and G. Saake. The active vertice method: A performance filtering approach to high-dimensional indexing. *Elsevier Data and Knowledge Engineering (DKE)*, 51(3):369–397, 2004.

[3] M. Bawa, T. Condie, and P. Ganesan. LSH forest: Seft-tuning indexes for similarity search. In *Proc. of the 14th Intl. World Wide Web Conf. (WWW)*, pages 651–660, 2005.

[4] J. L. Bentley. K-D trees for semi-dynamic point sets. In *Proc. of the 6th ACM Symposium on Computational Geometry (SCG)*, pages 187–197, 1990.

[5] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proc. of the 23rd Intl. Conf. on Machine Learning*, pages 97–104, 2006.

[6] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17:419–428, 2001.

[7] E. Chávez, G. Navarro, R. A. Baeza-Yates, and J. L. M. roqun. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[8] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. of the 20th Symposium on Computational Geometry(SCG)*, pages 253–262, 2004.

[9] T. Deselaers. *Features for Image Retrieval*. PhD thesis, RWTH Aachen University. Aachen, Germany, December 2003.

[10] D. Dobkin and R. J. Lipton. Multidimensional search problems. *SIAM J. on Computing*, 5(2):181–186, 1976.

[11] FIRE: Flexible Image Retrieval Engine. `http://www-i6.informatik.rwth-aschen.de/~deselaers/fire.html`.

[12] I. K. Fodor. A survey of dimension reduction techniques. Technical Report UCRL-ID-148494, Lawrence Livermore National Laboratory, 2002.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of 25th Intl. Conf. on Very Large Data Bases(VLDB)*, pages 518–529, 1999.

[14] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM Conf. on Management of Data (SIGMOD)*, pages 47–57, 1984.

[15] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[16] P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. In *Proc. of the 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 189–197, 2000.

[17] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimens ionality. In *Proc. of the 30th ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[18] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. of ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 369–380, 1997.

[19] R. Krauthgamer and J. R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proc.s of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 798–807, 2004.

[20] S. Meiser. Point location in arrangements of hyperplanes. *Information and Computation*, 106(2):286–303, 1993.

[21] R. Motwani, A. Naor, and R. Panigrahy. Lower bounds on locality sensitive hashing. In *Proc. of the 22nd ACM Symposium on Computational Geometry (SCG)*, pages 154–157, 2006.

[22] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms(SODA)*, 2006.

[23] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *Proc. of the 26th Intl. Conf. on Very Larg e Data Bases (VLDB)*, pages 516–526, 2000.

[24] The Stanford WebBase Project. `http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/`.

[25] SWITCHBOARD-1 Release 2. `http://www.ldc.upenn.edu/Catalog/docs/switchboard/`.

[26] G. Tzanetakis and P. Cook. *MARSYAS: A Framework for Audio Analysis*. Cambridge University Press, 2000.

[27] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity search methods in high dimensional spaces. In *Proc. of the 24th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 194–205, 1998.

[28] V. Zolotarev. One-dimensional stable distributions. *Translations of Mathematical Monographs*, 65, 1986.