Similarity Search in High Dimensions via Hashing

ARISTIDES GIONIS [*] PIOTR INDYK [†] RAJEEV MOTWANI [‡]
Department of Computer Science
Stanford University
Stanford, CA 94305
{gionis,indyk,rajeev}@cs.stanford.edu

Abstract

The nearest- or near-neighbor query problems arise in a large variety of database applications, usually in the context of similarity searching. Of late, there has been increasing interest in building search/index structures for performing similarity search over high-dimensional data, e.g., image databases, document collections, time-series databases, and genome databases. Unfortunately, all known techniques for solving this problem fall prey to the "curse of dimensionality." That is, the data structures scale poorly with data dimensionality; in fact, if the number of dimensions exceeds 10 to 20, searching in k-d trees and related structures involves the inspection of a large fraction of the database, thereby doing no better than brute-force linear search. It has been suggested that since the selection of features and the choice of a distance metric in typical applications is rather heuristic, determining an approximate nearest neighbor should suffice for most practical purposes. In this paper, we examine a novel scheme for approximate similarity search based on hashing. The basic idea is to hash the points from the database so as to ensure that the probability of collision is much higher for objects that are close to each other than for those that are far apart. We provide experimental evidence that our method gives significant improvement in running time over other methods for searching in high-dimensional spaces based on hierarchical tree decomposition. Experimental results also indicate that our scheme scales well even for a relatively large number of dimensions (more than 50).

1 Introduction

A similarity search problem involves a collection of objects (e.g., documents, images) that are characterized by a collection of relevant features and represented as points in a high-dimensional attribute space; given queries in the form of points in this space, we are required to find the nearest (most similar) object to the query. The particularly interesting and well-studied case is the d-dimensional Euclidean space. The problem is of major importance to a variety of applications; some examples are: data compression [20]; databases and data mining [21]; information retrieval [11, 16, 38]; image and video databases [15, 17, 37, 42]; machine learning [7]; pattern recognition [9, 13]; and, statistics and data analysis [12, 27]. Typically, the features of the objects of interest are represented as points in $\Re^d$ and a distance metric is used to measure similarity of objects. The basic problem then is to perform indexing or similarity searching for query objects. The number of features (i.e., the dimensionality) ranges anywhere from tens to thousands. For example, in multimedia applications such as IBM's QBIC (Query by Image Content), the number of features could be several hundreds [15, 17]. In information retrieval for text documents, vector-space representations involve several thousands of dimensions, and it is considered to be a dramatic improvement that dimension-reduction techniques, such as the Karhunen-Loéve transform [26, 30] (also known as principal components analysis [22] or latent semantic indexing [11]), can reduce the dimensionality to a mere few hundreds!

[*]Supported by NAVY N00014-96-1-1221 grant and NSF Grant IIS-9811904.

[†]Supported by Stanford Graduate Fellowship and NSF NYI Award CCR-9357849.

[‡]Supported by ARO MURI Grant DAAH04-96-1-0007, NSF Grant IIS-9811904, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

The low-dimensional case (say, for d equal to 2 or 3) is well-solved [14], so the main issue is that of dealing with a large number of dimensions, the so-called "curse of dimensionality." Despite decades of intensive effort, the current solutions are not entirely satisfactory; in fact, for large enough d, in theory or in practice, they provide little improvement over a linear algorithm which compares a query to each point from the database. In particular, it was shown in [45] that, both empirically and theoretically, *all* current indexing techniques (based on space partitioning) degrade to linear search for sufficiently high dimensions. This situation poses a serious obstacle to the future development of large scale similarity search systems. Imagine for example a search engine which enables content-based image retrieval on the World-Wide Web. If the system was to index a significant fraction of the web, the number of images to index would be at least of the order tens (if not hundreds) of million. Clearly, no indexing method exhibiting linear (or close to linear) dependence on the data size could manage such a huge data set.

The premise of this paper is that in many cases it is not necessary to insist on the *exact* answer; instead, determining an *approximate* answer should suffice (refer to Section 2 for a formal definition). This observation underlies a large body of recent research in databases, including using random sampling for histogram estimation [8] and median approximation [33], using wavelets for selectivity estimation [34] and approximate SVD [25]. We observe that there are many applications of nearest neighbor search where an approximate answer is good enough. For example, it often happens (e.g., see [23]) that the relevant answers are *much* closer to the query point than the irrelevant ones; in fact, this is a desirable property of a good similarity measure. In such cases, the approximate algorithm (with a suitable approximation factor) will return the same result as an exact algorithm. In other situations, an approximate algorithm provides the user with a time-quality tradeoff — the user can decide whether to spend more time waiting for the exact answer, or to be satisfied with a much quicker approximation (e.g., see [5]).

The above arguments rely on the assumption that approximate similarity search can be performed much faster than the exact one. In this paper we show that this is indeed the case. Specifically, we introduce a new indexing method for approximate nearest neighbor with a truly sublinear dependence on the data size even for high-dimensional data. Instead of using space partitioning, it relies on a new method called *locality-sensitive hashing (LSH).* The key idea is to hash the points using several hash functions so as to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point. We provide such locality-sensitive hash functions that are simple and easy to implement; they can also be naturally extended to the *dynamic* setting, i.e., when insertion and deletion operations also need to be supported. Although in this paper we are focused on Euclidean spaces, different LSH functions can be also used for other similarity measures, such as dot product [5].

Locality-Sensitive Hashing was introduced by Indyk and Motwani [24] for the purposes of devising main memory algorithms for nearest neighbor search; in particular, it enabled us to achieve worst-case $O(dn^{1/\epsilon})$-time for approximate nearest neighbor query over an n-point database. In this paper we improve that technique and achieve a significantly improved query time of $O(dn^{1/(1+\epsilon)})$. This yields an approximate nearest neighbor algorithm running in sublinear time for any $\epsilon > 0$. Furthermore, we generalize the algorithm and its analysis to the case of *external memory.*

We support our theoretical arguments by empirical evidence. We performed experiments on two data sets. The first contains 20,000 histograms of color images, where each histogram was represented as a point in d-dimensional space, for d up to 64. The second contains around 270,000 points representing texture information of blocks of large aerial photographs. All our tables were stored on disk. We compared the performance of our algorithm with the performance of the Sphere/Rectangle-tree (SR-tree) [28], a recent data structure which was shown to be comparable to or significantly more efficient than other tree-decomposition-based indexing methods for spatial data. The experiments show that our algorithm is significantly faster than the earlier methods, in some cases even by several orders of magnitude. It also scales well as the data size and dimensionality increase. Thus, it enables a new approach to high-performance similarity search — fast retrieval of approximate answer, possibly followed by a slower but more accurate computation in the few cases where the user is not satisfied with the approximate answer.

The rest of this paper is organized as follows. In Section 2 we introduce the notation and give formal definitions of the similarity search problems. Then in Section 3 we describe locality-sensitive hashing and show how to apply it to nearest neighbor search. In Section 4 we report the results of experiments with LSH. The related work is described in Section 5. Finally, in Section 6 we present conclusions and ideas for future research.

2 Preliminaries

We use l_p^d to denote the Euclidean space $\Re^d$ under the l_p norm, i.e., when the length of a vector $(x_1, \ldots x_d)$ is defined as $(|x_1|^p + \ldots + |x_d|^p)^{1/p}$. Further, $d_p(p, q) =$

$||p-q||_p$ denotes the distance between the points p and q in l_p^d. We use H^d to denote the *Hamming metric space* of dimension d, i.e., the space of binary vectors of length d under the standard Hamming metric. We use $d_H(p,q)$ denote the *Hamming distance*, i.e., the number of bits on which p and q differ.

The nearest neighbor search problem is defined as follows:

Definition 1 (Nearest Neighbor Search (NNS)) *Given a set P of n objects represented as points in a normed space l_p^d, preprocess P so as to efficiently answer queries by finding the point in P closest to a query point q.*

The definition generalizes naturally to the case where we want to return $K > 1$ points. Specifically, in the *K-Nearest Neighbors Search (K-NNS)*, we wish to return the K points in the database that are closest to the query point. The approximate version of the NNS problem is defined as follows:

Definition 2 (ϵ-Nearest Neighbor Search (ϵ-NNS)) *Given a set P of points in a normed space l_p^d, preprocess P so as to efficiently return a point $p \in P$ for any given query point q, such that $d(q,p) \leq (1+\epsilon)d(q,P)$, where $d(q,P)$ is the distance of q to the its closest point in P.*

Again, this definition 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 q is at most $(1+\epsilon)$ times the distance from the ith nearest point to q.

3 The Algorithm

In this section we present efficient solutions to the approximate versions of the NNS problem. Without significant loss of generality, we will make the following two assumptions about the data:

1. the distance is defined by the l_1 norm (see comments below),
2. all coordinates of points in P are positive integers.

The first assumption is not very restrictive, as usually there is no clear advantage in, or even difference between, using l_2 or l_1 norm for similarity search. For example, the experiments done for the Webseek [43] project (see [40], chapter 4) show that comparing color histograms using l_1 and l_2 norms yields very similar results (l_1 is marginally better). Both our data sets (see Section 4) have a similar property. Specifically, we observed that a nearest neighbor of an average query point computed under the l_1 norm was also an ϵ-approximate neighbor under the l_2 norm with an average value of ϵ less than 3% (this observation holds for both data sets). Moreover, in most cases (i.e., for 67% of the queries in the first set and 73% in the second set) the nearest neighbors under l_1 and l_2 norms were *exactly* the same. This observation is interesting in its own right, and can be partially explained via the theorem by Figiel et al (see [19] and references therein). They showed analytically that by simply applying scaling and random rotation to the space l_2, we can make the distances induced by the l_1 and l_2 norms almost equal up to an arbitrarily small factor. It seems plausible that real data is already randomly rotated, thus the difference between l_1 and l_2 norm is very small. Moreover, for the data sets for which this property does not hold, we are guaranteed that after performing scaling and random rotation our algorithms can be used for the l_2 norm with arbitrarily small loss of precision.

As far as the second assumption is concerned, clearly all coordinates can be made positive by properly translating the origin of $\Re^d$. We can then convert all coordinates to integers by multiplying them by a suitably large number and rounding to the nearest integer. It can be easily verified that by choosing proper parameters, the error induced by rounding can be made arbitrarily small. Notice that after this operation the minimum interpoint distance is 1.

3.1 Locality-Sensitive Hashing

In this section we present locality-sensitive hashing (LSH). This technique was originally introduced by Indyk and Motwani [24] for the purposes of devising main memory algorithms for the ϵ-NNS problem. Here we give an improved version of their algorithm. The new algorithm is in many respects more natural than the earlier one: it does not require the hash buckets to store only one point; it has better running time guarantees; and, the analysis is generalized to the case of secondary memory.

Let C be the largest coordinate in all points in P. Then, as per [29], we can embed P into the Hamming cube $H^{d'}$ with $d' = Cd$, by transforming each point $p = (x_1, \ldots x_d)$ into a binary vector

$$v(p) = \text{Unary}_C(x_1) \ldots \text{Unary}_C(x_d),$$

where $\text{Unary}_C(x)$ denotes the unary representation of x, i.e., is a sequence of x ones followed by $C-x$ zeroes.

Fact 1 *For any pair of points p, q with coordinates in the set $\{1 \ldots C\}$,*

$$d_1(p,q) = d_H(v(p), v(q)).$$

That is, the embedding preserves the distances between the points. Therefore, in the sequel we can concentrate on solving ϵ-NNS in the Hamming space $H^{d'}$. However, we emphasize that we *do not* need to actually *convert* the data to the unary representation, which could be expensive when C is large; in fact, all our algorithms can be made to run in time *independent* on C. Rather, the unary representation provides us with a convenient framework for description of the algorithms which would be more complicated otherwise.

We define the hash functions as follows. For an integer l to be specified later, choose l subsets $I_1, \ldots, I_l$ of $\{1, \ldots, d'\}$. Let $p_{|I}$ denote the projection of vector p on the coordinate set I, i.e., we compute $p_{|I}$ by selecting the coordinate positions as per I and concatenating the bits in those positions. Denote $g_j(p) = p_{|I_j}$. For the preprocessing, we store each $p \in P$ in the bucket $g_j(p)$, for $j = 1, \ldots, l$. As the total number of buckets may be large, we compress the buckets by resorting to standard hashing. Thus, we use two levels of hashing: the LSH function maps a point p to bucket $g_j(p)$, and a standard hash function maps the contents of these buckets into a hash table of size M. The maximal bucket size of the latter hash table is denoted by B. For the algorithm's analysis, we will assume hashing with chaining, i.e., when the number of points in a bucket exceeds B, a new bucket (also of size B) is allocated and linked to and from the old bucket. However, our implementation does not employ chaining, but relies on a simpler approach: if a bucket in a given index is full, a new point cannot be added to it, since it will be added to some other index with high probability. This saves us the overhead of maintaining the link structure.

The number n of points, the size M of the hash table, and the maximum bucket size B are related by the following equation:

$$M = \alpha \frac{n}{B},$$

where α is the memory utilization parameter, i.e., the ratio of the memory allocated for the index to the size of the data set.

To process a query q, we search all indices $g_1(q), \ldots, g_l(q)$ until we either encounter at least $c \cdot l$ points (for c specified later) or use all l indices. Clearly, the number of disk accesses is always upper bounded by the number of indices, which is equal to l. Let $p_1, \ldots, p_t$ be the points encountered in the process. For Approximate K-NNS, we output the K points p_i closest to q; in general, we may return fewer points if the number of points encountered is less than K.

It remains to specify the choice of the subsets I_j. For each $j \in \{1, \ldots, l\}$, the set I_j consists of k elements from $\{1, \ldots, d'\}$ sampled uniformly at random with replacement. The optimal value of k is chosen to maximize the probability that a point p "close" to q will fall into the same bucket as q, and also to minimize the probability that a point p' "far away" from q will fall into the same bucket. The choice of the values of l and k is deferred to the next section.

```
**Algorithm** Preprocessing
**Input** A set of points $P$,
  $l$ (number of hash tables),
**Output** Hash tables $\mathcal{T}_i$, $i = 1, \ldots, l$
**Foreach** $i = 1, \ldots, l$
  Initialize hash table $\mathcal{T}_i$ by generating
  a random hash function $g_i(\cdot)$
**Foreach** $i = 1, \ldots, l$
  **Foreach** $j = 1, \ldots, n$
    Store point $p_j$ on bucket $g_i(p_j)$ of hash table $\mathcal{T}_i$
```

Figure 1: Preprocessing algorithm for points already embedded in the Hamming cube.

```
**Algorithm** Approximate Nearest Neighbor Query
**Input** A query point $q$,
  $K$ (number of appr. nearest neighbors)
**Access** To hash tables $\mathcal{T}_i$, $i = 1, \ldots, l$
  generated by the preprocessing algorithm
**Output** $K$ (or less) appr. nearest neighbors
$S \leftarrow \emptyset$
**Foreach** $i = 1, \ldots, l$
  $S \leftarrow S \cup \{$points found in $g_i(q)$ bucket of table $\mathcal{T}_i\}$
Return the $K$ nearest neighbors of $q$ found in set $S$
/* Can be found by main memory linear search */
```

Figure 2: Approximate Nearest Neighbor query answering algorithm.

Although we are mainly interested in the I/O complexity of our scheme, it is worth pointing out that the hash functions can be efficiently computed if the data set is obtained by mapping l_1^d into d'-dimensional Hamming space. Let p be any point from the data set and let p' denote its image after the mapping. Let I be the set of coordinates and recall that we need to compute $p'_{|I}$. For $i = 1, \ldots, d$, let $I_{|i}$ denote, in sorted order, the coordinates in I which correspond to the ith coordinate of p. Observe, that projecting p' on $I_{|i}$ results in a sequence of bits which is monotone, i.e., consists of a number, say o_i, of ones followed by zeros. Therefore, in order to represent p'_I it is sufficient to compute o_i for $i = 1, \ldots, d$. However, the latter task is equivalent to finding the number of elements in the sorted array $I_{|i}$ which are smaller than a given value, i.e., the ith coordinate of p. This can be done via binary search in $\log C$ time, or even in constant time using a precomputed array of C bits. Thus, the total time needed to compute the function is either $O(d \log C)$ or $O(d)$, depending on resources used. In our experimental section, the value of C can be made very small, and therefore we will resort to the second method.

For quick reference we summarize the preprocessing and query answering algorithms in Figures 1 and 2.

3.2 Analysis of Locality-Sensitive Hashing

The principle behind our method is that the probability of collision of two points p and q is closely related to the distance between them. Specifically, the larger the distance, the smaller the collision probability. This intuition is formalized as follows [24]. Let $D(\cdot,\cdot)$ be a distance function of elements from a set S, and for any $p \in S$ let $\mathcal{B}(p,r)$ denote the set of elements from S within the distance r from p.

Definition 3 *A family $\mathcal{H}$ of functions from S to U is called* (r_1, r_2, p_1, p_2)-sensitive *for $D(\cdot,\cdot)$ if for any $q, p \in S$*

- *if $p \in \mathcal{B}(q, r_1)$ then $\Pr_{\mathcal{H}}[h(q) = h(p)] \geq p_1$,*
- *if $p \notin \mathcal{B}(q, r_2)$ then $\Pr_{\mathcal{H}}[h(q) = h(p)] \leq p_2$.*

In the above definition, probabilities are considered with respect to the random choice of a function h from the family $\mathcal{H}$. In order for a locality-sensitive family to be useful, it has to satisfy the inequalities $p_1 > p_2$ and $r_1 < r_2$.

Observe that if $D(\cdot,\cdot)$ is the Hamming distance $d_H(\cdot,\cdot)$, then the family of projections on one coordinate is locality-sensitive. More specifically:

Fact 2 *Let S be $H^{d'}$ (the d'-dimensional Hamming cube) and $D(p,q) = d_H(p,q)$ for $p, q \in H^{d'}$. Then for any r, $\epsilon > 0$, the family $\mathcal{H}_{d'} = \{h_i : h_i((b_1, \ldots, b_{d'})) = b_i, \text{ for } i = 1, \ldots, d'\}$ is $\left(r, r(1+\epsilon), 1 - \frac{r}{d'}, 1 - \frac{r(1+\epsilon)}{d'}\right)$-sensitive.*

We now generalize the algorithm from the previous section to an *arbitrary* locality-sensitive family $\mathcal{H}$. Thus, the algorithm is equally applicable to other locality-sensitive hash functions (e.g., see [5]). The generalization is simple: the functions g are now defined to be of the form

$$g_i(p) = (h_{i_1}(p), h_{i_2}(p), \ldots, h_{i_k}(p)),$$

where the functions $h_{i_1}, \ldots, h_{i_k}$ are randomly chosen from $\mathcal{H}$ with replacement. As before, we choose l such functions $g_1, \ldots, g_l$. In the case when the family $\mathcal{H}_{d'}$ is used, i.e., each function selects one bit of an argument, the resulting values of $g_j(p)$ are essentially equivalent to $p_{|I_j}$.

We now show that the LSH algorithm can be used to solve what we call the (r, ϵ)-Neighbor problem: determine whether there exists a point p within a fixed distance $r_1 = r$ of q, or whether all points in the database are at least a distance $r_2 = r(1+\epsilon)$ away from q; in the first case, the algorithm is required to return a point p' within distance at most $(1+\epsilon)r$ from q. In particular, we argue that the LSH algorithm solves this problem for a proper choice of k and l, depending on r and ϵ. Then we show how to apply the solution to this problem to solve ϵ-NNS.

Denote by P' the set of all points $p' \in P$ such that $d(q, p') > r_2$. We observe that the algorithm correctly solves the (r, ϵ)-Neighbor problem if the following two properties hold:

P1 If there exists p^* such that $p^* \in \mathcal{B}(q, r_1)$, then $g_j(p^*) = g_j(q)$ for some $j = 1, \ldots, l$.

P2 The total number of blocks pointed to by q and containing only points from P' is less than cl.

Assume that $\mathcal{H}$ is a (r_1, r_2, p_1, p_2)-sensitive family; define $\rho = \frac{\ln 1/p_1}{\ln 1/p_2}$. The correctness of the LSH algorithm follows from the following theorem.

Theorem 1 *Setting $k = \log_{1/p_2}(n/B)$ and $l = \left(\frac{n}{B}\right)^\rho$ guarantees that properties* **P1** *and* **P2** *hold with probability at least $\frac{1}{2} - \frac{1}{e} \geq 0.132$.*

Remark 1 *Note that by repeating the LSH algorithm $O(1/\delta)$ times, we can amplify the probability of success in at least one trial to $1 - \delta$, for any $\delta > 0$.*

Proof: Let property **P1** hold with probability P_1, and property **P2** hold with probability P_2. We will show that both P_1 and P_2 are large. Assume that there exists a point p^* within distance r_1 of q; the proof is quite similar otherwise. Set $k = \log_{1/p_2}(n/B)$. The probability that $g(p') = g(q)$ for $p \in P - \mathcal{B}(q, r_2)$ is at most $p_2^k = \frac{B}{n}$. Denote the set of all points $p' \notin \mathcal{B}(q, r_2)$ by P'. The expected number of blocks allocated for g_j which contain *exclusively* points from P' does not exceed 2. The expected number of such blocks allocated for all g_j is at most $2l$. Thus, by the Markov inequality [35], the probability that this number exceeds $4l$ is less than $1/2$. If we choose $c = 4$, the probability that the property P2 holds is $P_2 > 1/2$.

Consider now the probability of $g_j(p^*) = g_j(q)$. Clearly, it is bounded from below by

$$p_1^k = p_1^{\log_{1/p_2} n/B} = (n/B)^{-\frac{\log 1/p_1}{\log 1/p_2}} = (n/B)^{-\rho}.$$

By setting $l = \left(\frac{n}{B}\right)^\rho$, we bound from above the probability that $g_j(p^*) \neq g_j(q)$ for all $j = 1, \ldots, l$ by $1/e$. Thus the probability that one such g_j exists is at least $P_1 \geq 1 - 1/e$.

Therefore, the probability that both properties P1 and P2 hold is at least $1 - [(1 - P_1) + (1 - P_2)] = P_1 + P_2 - 1 \geq \frac{1}{2} - \frac{1}{e}$. The theorem follows. □

In the following we consider the LSH family for the Hamming metric of dimension d' as specified in Fact 2. For this case, we show that $\rho \leq \frac{1}{1+\epsilon}$ assuming that $r < \frac{d'}{\ln n}$; the latter assumption can be easily satisfied by increasing the dimensionality by padding a sufficiently long string of 0s at the end of each point's representation.

Fact 3 *Let* $r < \frac{d'}{\ln n}$. *If* $p_1 = 1 - \frac{r}{d'}$ *and* $p_2 = 1 - \frac{r(1+\epsilon)}{d'}$, *then* $\rho = \frac{\ln 1/p_1}{\ln 1/p_2} \le \frac{1}{1+\epsilon}$.

Proof: Observe that

$$\rho = \frac{\ln 1/p_1}{\ln 1/p_2} = \frac{\ln \frac{1}{1-r/d'}}{\ln \frac{1}{1-(1+\epsilon)r/d'}} = \frac{\ln(1-r/d')}{\ln(1-(1+\epsilon)r/d')}.$$

Multiplying both the numerator and the denominator by $\frac{d'}{r}$, we obtain:

$$\begin{aligned}\rho &= \frac{\frac{d'}{r}\ln(1-r/d')}{\frac{d'}{r}\ln(1-(1+\epsilon)r/d')} \\ &= \frac{\ln(1-r/d')^{d'/r}}{\ln(1-(1+\epsilon)r/d')^{d'/r}} = \frac{U}{L}\end{aligned}$$

In order to upper bound ρ, we need to bound U from below and L from above; note that both U and L are negative. To this end we use the following inequalities [35]:

$$(1-(1+\epsilon)r/d')^{d'/r} < e^{-(1+\epsilon)}$$

and

$$\left(1-\frac{r}{d'}\right)^{d'/r} > e^{-1}\left(1-\frac{1}{d'/r}\right).$$

Therefore,

$$\begin{aligned}\frac{U}{L} &< \frac{\ln(e^{-1}(1-\frac{1}{d'/r}))}{\ln e^{-(1+\epsilon)}} \\ &= \frac{-1+\ln(1-\frac{1}{d'/r})}{-(1+\epsilon)} \\ &= 1/(1+\epsilon) - \frac{\ln(1-\frac{1}{d'/r})}{1+\epsilon} \\ &< 1/(1+\epsilon) - \ln(1-1/\ln n)\end{aligned}$$

where the last step uses the assumptions that $\epsilon > 0$ and $r < \frac{d'}{\ln n}$. We conclude that

$$\begin{aligned}n^{\rho} &< n^{1/(1+\epsilon)} n^{-\ln(1-1/\ln n)} \\ &= n^{1/(1+\epsilon)}(1-1/\ln n)^{-\ln n} = O(n^{1/(1+\epsilon)})\end{aligned}$$

□

We now return to the ϵ-NNS problem. First, we observe that we could reduce it to the (r,ϵ)-Neighbor problem by building several data structures for the latter problem with different values of r. More specifically, we could explore r equal to r_0, $r_0(1+\epsilon)$, $r_0(1+\epsilon)^2, \ldots, r_{max}$, where r_0 and r_{max} are the smallest and the largest possible distance between the query and the data point, respectively. We remark that the number of different radii could be further reduced [24] at the cost of increasing running time and space requirement. On the other hand, we observed that in practice choosing only *one* value of r is sufficient to produce answers of good quality. This can be explained as in [10] where it was observed that the distribution of distances between a query point and the data set in most cases does not depend on the specific query point, but on the intrinsic properties of the data set. Under the assumption of distribution invariance, the same parameter r is likely to work for a vast majority of queries. Therefore in the experimental section we adopt a fixed choice of r and therefore also of k and l.

4 Experiments

In this section we report the results of our experiments with locality-sensitive hashing method. We performed experiments on two data sets. The first one contains up to 20,000 histograms of color images from COREL Draw library, where each histogram was represented as a point in d-dimensional space, for d up to 64. The second one contains around 270,000 points of dimension 60 representing texture information of blocks of large large aerial photographs. We describe the data sets in more detail later in the section.

We decided not to use randomly-chosen synthetic data in our experiments. Though such data is often used to measure the performance of *exact* similarity search algorithms, we found it unsuitable for evaluation of *approximate* algorithms for the high data dimensionality. The main reason is as follows. Assume a data set consisting of points chosen independently at random from the same distribution. Most distributions (notably uniform) used in the literature assume that all coordinates of each point are chosen independently. In such a case, for any pair of points p, q the distance $d(p,q)$ is sharply concentrated around the mean; for example, for the uniform distribution over the unit cube, the expected distance is $O(d)$, while the standard deviation is only $O(\sqrt{d})$. Thus almost all pairs are approximately within the same distance, so the notion of approximate nearest neighbor is not meaningful — almost *every point* is an approximate nearest neighbor.

Implementation. We implement the LSH algorithm as specified in Section 3. The LSH functions can be computed as described in Section 3.1. Denote the resulting vector of coordinates by $(v_1, \ldots, v_k)$. For the second level mapping we use functions of the form

$$h(v_1, \ldots, v_k) = a_1 \cdot v_1 + \cdots + a_d \cdot v_k \bmod M,$$

where M is the size of the hash table and $a_1, \ldots, a_k$ are random numbers from interval $[0 \ldots M-1]$. These functions can be computed using only $2k-1$ operations, and are sufficiently random for our purposes, i.e., give low probability of collision. Each second level bucket is then directly mapped to a disk block. We assumed that each block is 8KB of data. As each coordinate in our data sets can be represented using 1 byte, we can store up to $8192/d$ d-dimensional points per block. Therefore, we assume the bucket size $B = 100$ for $d = 64$ or $d = 60$, $B = 300$ for $d = 27$ and $B = 1000$ for $d = 8$.

For the SR-tree, we used the implementation by Katayama, available from his web page [28]. As above, we allow it to store about 8192 coordinates per disk block.

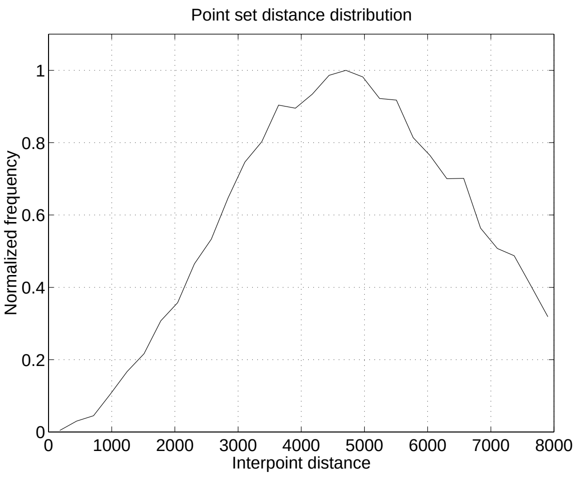
Performance measures. The goal of our experiments was to estimate two performance measures: speed (for both SR-tree and LSH) and accuracy (for LSH). The speed is measured by the number of disk blocks accessed in order to answer a query. We count all disk accesses, thus ignoring the issue of caching. Observe that in case of LSH this number is easy to predict as it is clearly equal to the number of indices used. As the number of indices also determines the storage overhead, it is a natural parameter to optimize.

The error of LSH is measured as follows. Following [2] we define (for the Approximate 1-NNS problem) the *effective error* as

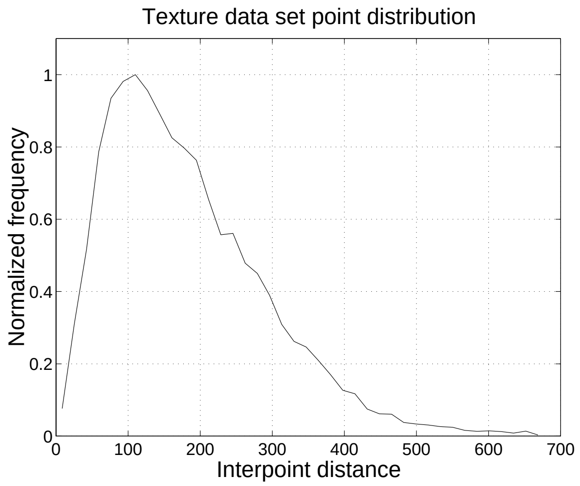
$$E = \frac{1}{|Q|} \sum_{\text{query } q \in Q} \frac{d_{LSH}}{d^*},$$

where d_{LSH} denotes the distance from a query point q to a point found by LSH, d^* is the distance from q to the closest point, and the sum is taken of all queries for which a nonempty index was found. We also measure the (small) fraction of queries for which no nonempty bucket was found; we call this quantity *miss ratio*. For the Approximate K-NNS we measure separately the distance ratios between the closest points found to the nearest neighbor, the 2nd closest one to the 2nd nearest neighbor and so on; then we average the ratios. The miss ratio is defined to be the fraction of cases when *less than* K points were found.

Data Sets. Our first data set consists of 20,000 histograms of color thumbnail-sized images of various contents taken from the COREL library. The histograms were extracted after transforming the pixels of the images to the 3-dimensional CIE-Lab color space [44]; the property of this space is that the distance between each pair of points corresponds to the perceptual dissimilarity between the colors that the two points represent. Then we partitioned the color space into a grid of smaller cubes, and given an image, we create the color histogram of the image by counting how many pixels fall into each of these cubes. By dividing each axis into u intervals we obtain a total of u^3 cubes. For most experiments, we assumed $u = 4$ obtaining a 64-dimensional space. Each histogram cube (i.e., color) then corresponds to a dimension of space representing the images. Finally, quantization is performed in order to fit each coordinate in 1 byte. For each point representing an image each coordinate effectively counts the number of the image's pixels of a specific color. All coordinates are clearly non-negative



(a) Color histograms



(b) Texture features

Figure 3: The profiles of the data sets.

integers, as assumed in Section 3. The distribution of interpoint distances in our point sets is shown in Figure 3. Both graphs were obtained by computing all interpoint distances of random subsets of 200 points, normalizing the maximal value to 1.

The second data set contains 275,465 feature vectors of dimension 60 representing texture information of blocks of large aerial photographs. This data set was provided by B.S. Manjunath [31, 32]; its size and dimensionality "provides challenging problems in high dimensional indexing" [31]. These features are obtained from Gabor filtering of the image tiles. The Gabor filter bank consists of 5 scales and 6 orientations of filters, thus the total number of filters is $5 \times 6 = 30$. The mean and standard deviation of each filtered output are used to constructed the feature vector ($d = 30 \times 2 = 60$). These texture features are extracted from 40 large air photos. Before the feature extraction, each airphoto is first partitioned into nonoverlapping tiles of size 64 times 64, from which the feature vectors are computed.

Query Sets. The difficulty in evaluating similarity searching algorithms is the lack of a publicly available database containing typical query points. Therefore, we had to construct the query set from the data set itself. Our construction is as follows: we split the data set randomly into two disjoint parts (call them S_1 and S_2). For the first data set the size of S_1 is 19,000 while the size of S_2 is 1000. The set S_1 forms a database of images, while the first 500 points from S_2 (denoted by Q) are used as query points (we use the other 500 points for various verification purposes). For the second data set we chose S_1 to be of size 270,000, and we use 1000 of the remaining 5,465 points as a query set. The numbers are slightly different for the scalability experiments as they require varying the size of the data set. In this case we chose a random subset of S_1 of required size.

4.1 Experimental Results

In this section we describe the results of our experiments. For both data sets they consist essentially of the following three steps. In the first phase we have to make the following choice: the value of k (the number of sampled bits) to choose for a given data set and the given number of indices l in order to minimize the effective error. It turned out that the optimal value of k is essentially independent of n and d and thus we can use the same value for different values of these parameters. In the second phase, we estimate the influence of the number of indices l on the error. Finally, we measure the performance of LSH by computing (for a variety of data sets) the minimal number of indices needed to achieve a specified value of error. When applicable, we also compare this performance with that of SR-trees.

4.2 Color histograms

For this data set, we performed several experiments aimed at understanding the behavior of LSH algorithm and its performance relative to SR-tree. As mentioned above, we started with an observation that the optimal value of sampled bits k is essentially independent of n and d and approximately equal to 700 for $d = 64$. The lack of dependence on n can be explained by the fact that the smaller data sets were obtained by sampling the large one and therefore all of the sets have similar structure; we believe the lack of dependence on d is also influenced by the structure of the data. Therefore the following experiments were done assuming $k = 700$.

Our next observation was that the value of storage overhead α does not exert much influence over the performance of the algorithm (we tried α's from the interval $[2, 5]$); thus, in the following we set $\alpha = 2$.

In the next step we estimated the influence of l on E. The results (for $n = 19,000$, $d = 64$, $K = 1$) are shown on Figure 4. As expected, one index is not sufficient to achieve reasonably small error — the effective error can easily exceed 50%. The error however decreases very fast as l increases. This is due to the fact that the probabilities of finding empty bucket are independent for different indices and therefore the probability that all buckets are empty decays exponentially in l.

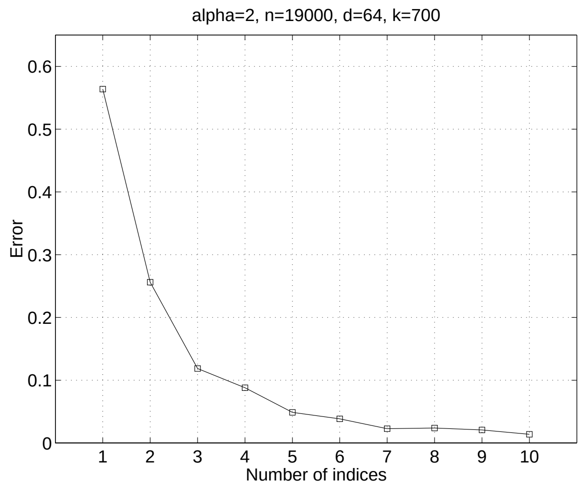


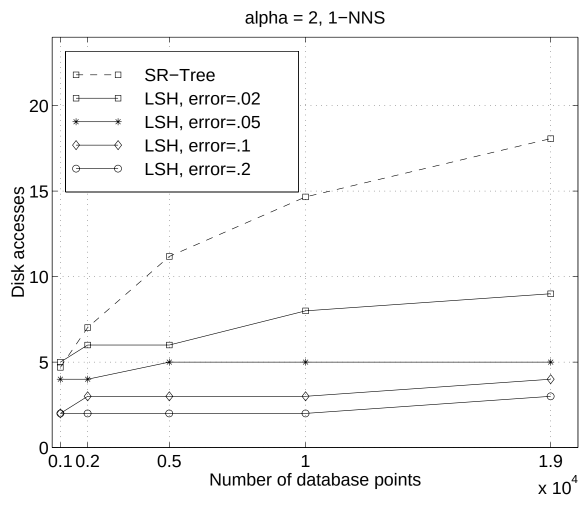
Figure 4: Error vs. the number of indices.

In order to compare the performance of LSH with SR-tree, we computed (for a variety of data sets) the minimal number of indices needed to achieve a specified value of error E equal to 2%, 5% , 10% or 20%. Then we investigated the performance of the two algorithms while varying the dimension and data size.

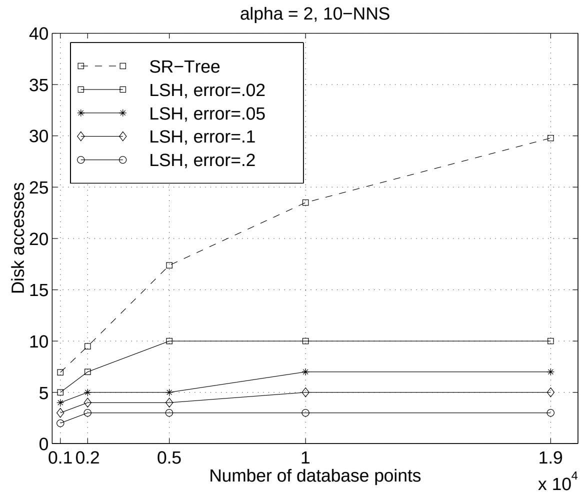
Dependence on Data Size. We performed the simulations for $d = 64$ and the data sets of sizes 1000, 2000, 5000, 10000 and 19000. To achieve better understanding of scalability of our algorithms, we did run the experiments twice: for Approximate 1-NNS and for Approximate 10-NNS. The results are presented on Figure 5.

Notice the strongly sublinear dependence exhibited by LSH: although for small $E = 2\%$ it matches SR-tree for $n = 1000$ with 5 blocks accessed (for $K = 1$), it requires 3 accesses more for a data set 19 times larger. At the same time the I/O activity of SR-tree increases by more than 200%. For larger errors the LSH curves are nearly flat, i.e., exhibit little dependence on the size of the data. Similar or even better behavior occurs for Approximate 10-NNS.

We also computed the miss ratios, i.e., the fraction of queries for which no answer was found. The results are presented on Figure 6. We used the parameters from the previous experiment. On can observe that for say $E = 5\%$ and Approximate 1-NNS, the miss ratios are quite high (10%) for small n, but decrease

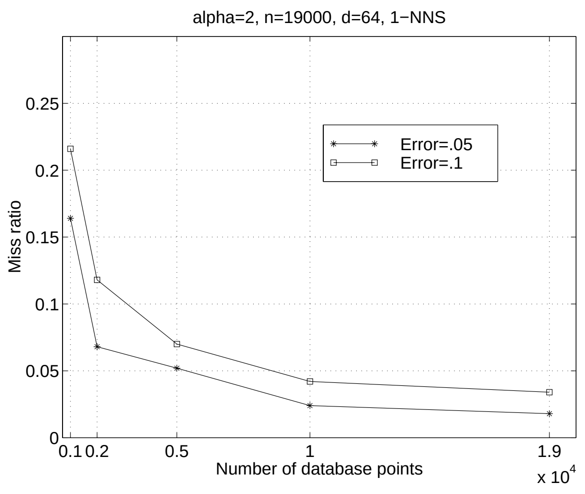


(a) Approximate 1-NNS

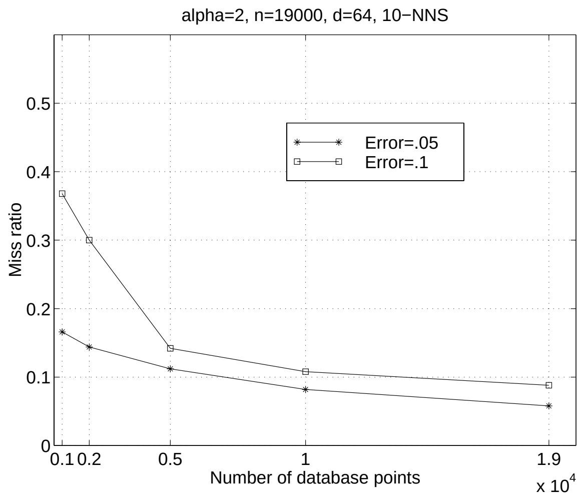


(b) Approximate 10-NNS

Figure 5: Number of indices vs. data size.



(a) Approximate 1-NNS



(b) Approximate 10-NNS

Figure 6: Miss ratio vs. data size.

to around 1% for $n = 19,000$.

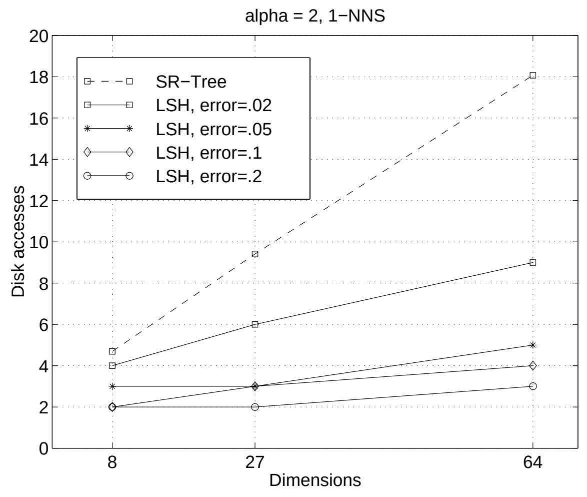
Dependence on Dimension. We performed the simulations for $d = 2^3, 3^3$ and 4^3; the choice of d's was limited to cubes of natural numbers because of the way the data has been created. Again, we performed the comparison for Approximate 1-NNS and Approximate 10-NNS; the results are shown on Figure 7. Note that LSH scales very well with the increase of dimensionality: for $E = 5\%$ the change from $d = 8$ to $d = 64$ increases the number of indices only by 2. The miss ratio was always below 6% for all dimensions.

This completes the comparison of LSH with SR-tree. For a better understanding of the behavior of LSH, we performed an additional experiment on LSH only. Figure 8 presents the performance of LSH when the number of nearest neighbors to retrieve vary from 1 to 100.

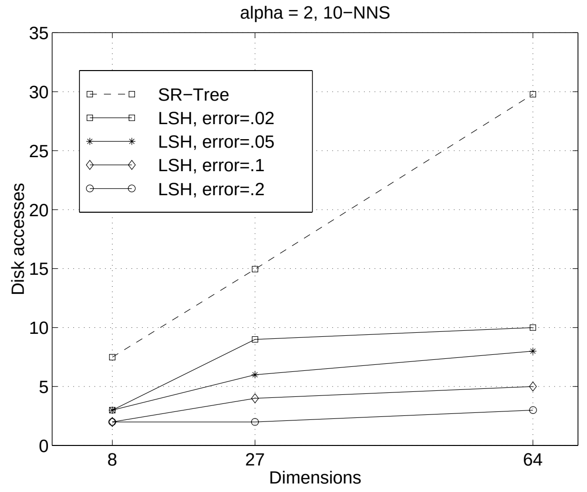
4.3 Texture features

The experiments with texture feature data were designed to measure the performance of the LSH algorithm on large data sets; note that the size of the texture file (270,000 points) is an order of magnitude larger than the size of the histogram data set (20,000 points). The first step (i.e., the choice of the number of sampled bits k) was very similar to the previous experiment, therefore we skip the detailed description here. We just state that we assumed that the number of sampled bits $k = 65$, with other parameters being: the storage overhead $\alpha = 1$, block size $B = 100$, and the number of nearest neighbors equal to 10. As stated above, the value of n was equal to 270,000.

We varied the number of indices from 3 to 100, which resulted in error from 50% to 15% (see Figure 9 (a)). The shape of the curve is similar as in the previous experiment. The miss ratio was roughly



(a) Approximate 1-NNS



(b) Approximate 10-NNS

Figure 7: Number of indices vs. dimension.

4% for 3 indices, 1% for 5 indices, and 0% otherwise.

To compare with SR-tree, we implemented that latter on random subsets of the whole data set of sizes from 10, 000 to 200, 000. For $n = 200,000$ the average number of blocks accessed per query by SR-tree was 1310, which is one to two orders of magnitude larger than the number of blocks accessed by our algorithm (see Figure 9 (b) where we show the running times of LSH for effective error 15%). Observe though that an SR-tree computes exact answers while LSH provides only an approximation. Thus in order to perform an accurate evaluation of LSH, we decided to compare it with a modified SR-tree algorithm which produces *approximate* answers. The modification is simple: instead of running SR-tree on the whole data set, we run it on a randomly chosen subset of it. In this way we achieve a speed-up of the algorithm (as the random sample of the data set is smaller than the original set)

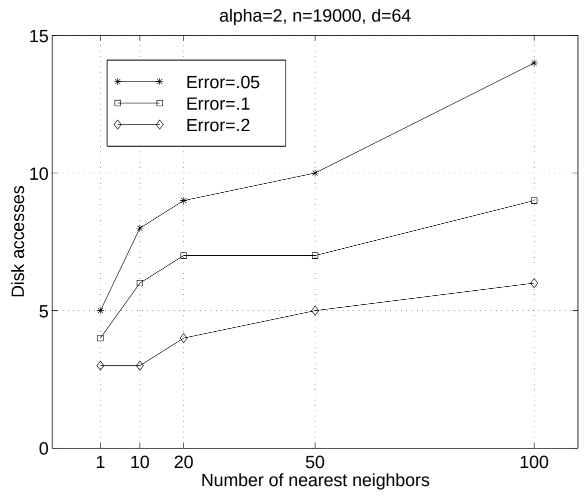


Figure 8: Number of indices vs. number of nearest neighbors.

while incurring some error.

The query cost versus error tradeoff obtained in this way (for the entire data set) is depicted on Figure 9; we also include a similar graph for LSH.

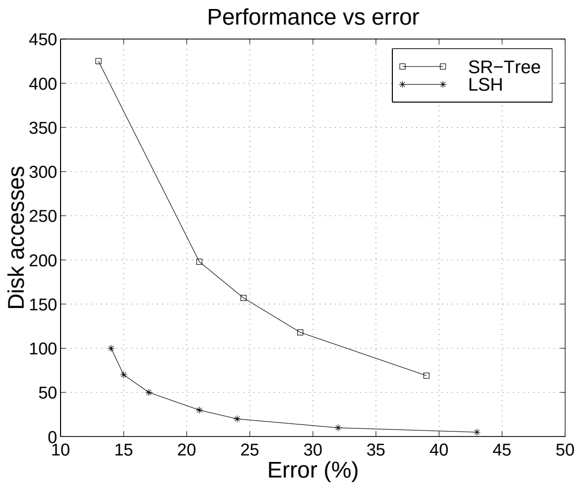
Observe that using random sampling results in considerable speed-up for the SR-tree algorithm, while keeping the error relatively low. However, even in this case the LSH algorithm offers considerably outperforms SR-trees, being up to an order of magnitude faster.

5 Previous Work

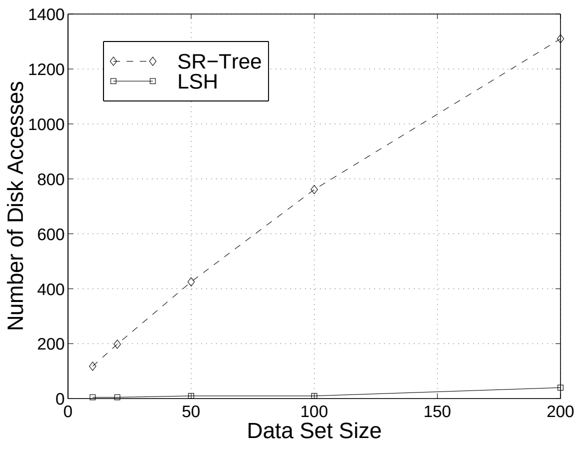
There is considerable literature on various versions of the nearest neighbor problem. Due to lack of space we omit detailed description of related work; the reader is advised to read [39] for a survey of a variety of data structures for nearest neighbors in geometric spaces, including variants of k-d trees, R-trees, and structures based on space-filling curves. The more recent results are surveyed in [41]; see also an excellent survey by [4]. Recent theoretical work in nearest neighbor search is briefly surveyed in [24].

6 Conclusions

We presented a novel scheme for approximate similarity search based on locality-sensitive hashing. We compared the performance of this technique and SR-tree, a good representative of tree-based spatial data structures. We showed that by allowing small error and additional storage overhead, we can considerably improve the query time. Experimental results also indicate that our scheme scales well to even a large number of dimensions and data size. An additional advantage of our data structure is that its running time is



(a)



(b)

Figure 9: (a) number of indices vs. error and (b) number of indices vs. size.

essentially determined in advance. All these properties make LSH a suitable candidate for high-performance and real-time systems.

In recent work [5, 23], we explore applications of LSH-type techniques to data mining and search for copyrighted video data. Our experience suggests that there is a lot of potential for further improvement of the performance of the LSH algorithm. For example, our data structures are created using a randomized procedure. It would be interesting if there was a more systematic method for performing this task; such a method could take additional advantage of the structure of the data set. We also believe that investigation of *hybrid* data structures obtained by merging the tree-based and hashing-based approaches is a fruitful direction for further research.

References

[1] S. Arya, D.M. Mount, and O. Narayan, Accounting for boundary effects in nearest-neighbor searching. *Discrete and Computational Geometry*, 16 (1996), pp. 155–176.

[2] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching, In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1994, pp. 573–582.

[3] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18 (1975), pp. 509–517.

[4] S. Berchtold and D.A. Keim. High-dimensional Index Structures. In Proceedings of SIGMOD, 1998, p. 501. See `http://www.informatik.uni-halle.de/ keim/ SIGMOD98Tutorial.ps.gz`

[5] E. Cohen. M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman. C. Yang. Finding Interesting Associations without Support Pruning. Technical Report, Computer Science Department, Stanford University.

[6] T.M. Chan. Approximate Nearest Neighbor Queries Revisited. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, 1997, pp. 352-358.

[7] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10 (1993), pp. 57–67.

[8] S. Chaudhuri, R. Motwani and V. Narasayya. "Random Sampling for Histogram Construction: How much is enough?". In *Proceedings of SIGMOD'98*, pp. 436–447.

[9] T.M. Cover and P.E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13 (1967), pp. 21–27.

[10] P. Ciaccia, M. Patella and P. Zezula, A cost model for similarity queries in metric spaces In *Proceedings of PODS'98*, pp. 59–68.

[11] S. Deerwester, S. T. Dumais, T.K. Landauer, G.W. Furnas, and R.A. Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Sciences,* 41 (1990), pp. 391–407.

[12] L. Devroye and T.J. Wagner. Nearest neighbor methods in discrimination. *Handbook of Statistics*, vol. 2, P.R. Krishnaiah and L.N. Kanal, eds., North-Holland, 1982.

[13] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, NY, 1973.

[14] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[15] C. Faloutsos, R. Barber, M. Flickner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3 (1994), pp. 231–262.

[16] C. Faloutsos and D.W. Oard. A Survey of Information Retrieval and Filtering Methods. Technical Report CS-TR-3514, Department of Computer Science, University of Maryland, College Park, 1995.

[17] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the QBIC system. *IEEE Computer*, 28 (1995), pp. 23–32.

[18] J.K. Friedman, J.L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3 (1977), pp. 209–226.

[19] T. Figiel, J. Lindenstrauss, V. D. Milman. The dimension of almost spherical sections of convex bodies. *Acta Math.* 139 (1977), no. 1-2, 53–94.

[20] A. Gersho and R.M. Gray. *Vector Quantization and Data Compression.* Kluwer, 1991.

[21] T. Hastie and R. Tibshirani. Discriminant adaptive nearest neighbor classification. In *Proceedings of the First International Conference on Knowledge Discovery & Data Mining*, 1995, pp. 142–149.

[22] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 27 (1933). pp. 417–441.

[23] P. Indyk, G. Iyengar, N. Shivakumar. Finding pirated video sequences on the Internet. Technical Report, Computer Science Department, Stanford University.

[24] P. Indyk and R. Motwani. Approximate Nearest Neighbor – Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th Symposium on Theory of Computing*, 1998, pp. 604–613.

[25] K.V. Ravi Kanth, D. Agrawal, A. Singh. "Dimensionality Reduction for Similarity Searching in Dynamic Databases". In *Proceedings of SIGMOD'98*, 166 – 176.

[26] K. Karhunen. Über lineare Methoden in der Wahrscheinlichkeitsrechnung. *Ann. Acad. Sci. Fennicae*, Ser. A137, 1947.

[27] V. Koivune and S. Kassam. Nearest neighbor filters for multivariate data. *IEEE Workshop on Nonlinear Signal and Image Processing*, 1995.

[28] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proc. SIGMOD'97*, pp. 369–380. The code is available from `http://www.rd.nacsis.ac.jp/~katayama/homepage/research/srtree/English.html`

[29] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. In *Proceedings of 35th Annual IEEE Symposium on Foundations of Computer Science*, 1994, pp. 577–591.

[30] M. Loéve. Fonctions aleastoires de second ordere. *Processus Stochastiques et mouvement Brownian*, Hermann, Paris, 1948.

[31] B. S. Manjunath. Airphoto dataset. `http://vivaldi.ece.ucsb.edu/Manjunath/research.htm`

[32] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of large image data. *IEEE Transactions on Pattern Analysis and Machine Intelligence, (Special Issue on Digital Libraries)*, 18 (8), pp. 837-842.

[33] G.S. Manku, S. Rajagopalan, and B.G. Lindsay. Approximate Medians and other Quantiles in One Pass and with Limited Memory. In *Proceedings of SIGMOD'98*, pp. 426–435.

[34] Y. Matias, J.S. Vitter, and M. Wang. Wavelet-based Histograms for Selectivity Estimations. In *Proceedings of SIGMOD'98*, pp. 448–459.

[35] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[36] M. Otterman. Approximate matching with high dimensionality R-trees. M.Sc. Scholarly paper, Dept. of Computer Science, Univ. of Maryland, College Park, MD, 1992.

[37] A. Pentland, R.W. Picard, and S. Sclaroff. Photobook: tools for content-based manipulation of image databases. In *Proceedings of the SPIE Conference on Storage and Retrieval of Image and Video Databases II*, 1994.

[38] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, New York, NY, 1983.

[39] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1989.

[40] J. R. Smith. Integrated Spatial and Feature Image Systems: Retrieval, Analysis and Compression. Ph.D. thesis, Columbia University, 1997. Available at `ftp://ftp.ctr.columbia.edu/CTR-Research/advent/public/public/jrsmith/thesis`

[41] T. Sellis, N. Roussopoulos, and C. Faloutsos. Multidimensional Access Methods: Trees Have Grown Everywhere. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, 1997, 13–15.

[42] A.W.M. Smeulders and R. Jain, eds. *Image Databases and Multi-media Search.* Proceedings of the First International Workshop, IDB-MMS'96, Amsterdam University Press, Amsterdam, 1996.

[43] J.R. Smith and S.F. Chang. Visually Searching the Web for Content. *IEEE Multimedia* 4 (1997): pp. 12–20. See also `http://disney.ctr.columbia.edu/WebSEEk`

[44] G. Wyszecki and W.S. Styles. *Color science: concepts and methods, quantitative data and formulae.* John Wiley and Sons, New York, NY, 1982.

[45] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for Similarity Search Methods in High Dimensional Spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, 1998, pp. 194-205.