To appear in Proceedings of IEEE International Conference on Data Engineering (ICDE), April 2008.

Nearest Neighbor Retrieval Using Distance-Based Hashing

Vassilis Athitsos [1], Michalis Potamias [2], Panagiotis Papapetrou [2], and George Kollios [2]

[1] *Computer Science and Engineering Department, University of Texas at Arlington*
Arlington, Texas, USA

[2] *Computer Science Department, Boston University*
Boston, Massachusetts, USA

***Abstract*— A method is proposed for indexing spaces with arbitrary distance measures, so as to achieve efficient approximate nearest neighbor retrieval. Hashing methods, such as Locality Sensitive Hashing (LSH), have been successfully applied for similarity indexing in vector spaces and string spaces under the Hamming distance. The key novelty of the hashing technique proposed here is that it can be applied to spaces with arbitrary distance measures, including non-metric distance measures. First, we describe a domain-independent method for constructing a family of binary hash functions. Then, we use these functions to construct multiple multibit hash tables. We show that the LSH formalism is not applicable for analyzing the behavior of these tables as index structures. We present a novel formulation, that uses statistical observations from sample data to analyze retrieval accuracy and efficiency for the proposed indexing method. Experiments on several real-world data sets demonstrate that our method produces good trade-offs between accuracy and efficiency, and significantly outperforms VP-trees, which are a well-known method for distance-based indexing.**

I. Introduction

Answering a nearest neighbor query consists of identifying, for a given query object, the most similar database objects. Nearest neighbor retrieval is a common and indispensable operation in a wide variety of real systems. A few example applications are nearest neighbor classification (e.g., [1], [2], [3]), analysis of biological sequences (e.g., [4], [5]), and content-based access to multimedia databases (e.g., [6], [7]). Given ever-increasing database sizes, there is a need for efficient and scalable indexing methods, that can facilitate accurate and efficient nearest neighbor retrieval.

Locality Sensitive Hashing (LSH) [8], [9] is a framework for hash-based indexing, with appealing theoretical properties and empirical performance. LSH is an approximate technique; it does not guarantee finding the true nearest neighbor for 100% of the queries. At the same time, LSH provides a statistical guarantee of producing a correct result with high probability. Theoretically, for a database of n vectors of d dimensions, the time complexity of finding the nearest neighbor of an object using LSH is sublinear in n and only polynomial in d. The theoretical advantages of LSH have been also empirically demonstrated in several applications involving high-dimensional data [10], [8], [2], [11], [3].

A key requirement for applying LSH to a particular space and distance measure is to identify a family of *locality sensitive* functions, satisfying the properties specified in [8]. As a consequence, LSH is only applicable for specific spaces and distance measures where such families of functions have been identified, such as real vector spaces with L_p distance measures, bit vectors with the Hamming distance, or strings with a substitution-based distance measure (that does not allow insertions or deletions) [12], [13]. This is in contrast to distance-based indexing methods, that build indexing structures based only on distances between objects, and thus can be readily applied to any space and distance measure.

In this paper we introduce Distance-Based Hashing (DBH), a novel indexing method for efficient approximate nearest neighbor retrieval. Compared to LSH, DBH has several similarities but also some important differences. Overall , the main novelties of DBH are the following:

- DBH is a hash-based indexing method that is *distance-based*. Consequently, DBH can be applied in arbitrary (and not necessarily metric) spaces and distance measures, whereas LSH cannot.
- Indexing performance (in terms of retrieval accuracy and retrieval efficiency) is estimated and optimized using statistics obtained from sample data, whereas in LSH performance guarantees are obtained by using some known geometric properties of a specific space and distance measure. Dependence on known geometric properties is exactly what makes LSH not applicable in arbitrary spaces.

An additional contribution of this paper is a description of two techniques for further improving DBH performance in practice: we describe a hierarchical version of DBH, where different index structures are tuned to different parts of the space of queries, and we also describe a method for significantly reducing the cost of computing the hash values for each query object.

Experiments with several real-world data sets demonstrate that DBH provides very good trade-offs between retrieval accuracy and efficiency, and that DBH outperforms VP-trees, a well-known distance-based method for indexing arbitrary spaces. Furthermore, no known method exists for applying LSH on those data sets, and this fact further demonstrates the need for a distance-based hashing scheme that DBH addresses.

II. Related Work

Various methods have been employed for speeding up nearest neighbor retrieval. Comprehensive reviews on the subject include [14], [15], [16], [17]. A large amount of work focuses on efficient nearest neighbor retrieval in multidimensional vector spaces using an L_p metric [18], [19], [20], [21], [22], [23]. However, many commonly used distance measures are not L_p metrics, and thus cannot be indexed with such methods. Popular examples of non-L_p distance measures include the edit distance for strings [24], dynamic time warping for timeseries [25], the chamfer distance [26] and shape context matching [1] for edge images, and the Kullback-Leibler (KL) distance for probability distributions [27].

A number of nearest neighbor methods can be applied for indexing arbitrary metric spaces; the reader is referred to [28], [29], [16] for surveys of such methods. VP-trees [30], metric trees [31] and MVP-trees [32] hierarchically partition the database into a tree structure by splitting, at each node, the set of objects based on their distances to pivot objects. M-trees [33] and slim-trees [34] are variants of metric trees explicitly designed for dynamic databases. An approximate variant of M-trees is proposed in [35], and achieves additional speed-ups by sacrificing the guarantee of always retrieving the true nearest neighbors. A general problem with the above-mentioned tree-based indexing methods is that they suffer from the curse of dimensionality: performance tends to approach brute-force search as the intrinsic dimensionality of the space exceeds a few tens of dimensions.

In domains with a computationally expensive distance measure, significant speed-ups can be obtained by embedding objects into another space with a more efficient distance measure. Several methods have been proposed for embedding arbitrary spaces into a Euclidean or pseudo-Euclidean space [36], [37], [38], [39], [40]. However, used by themselves, embedding methods simply substitute a fast approximate distance for the original distance, and still use brute force to compare the query to all database objects, albeit using the fast approximate distance instead of the original one.

Non-metric distance measures are frequently used in pattern recognition. Examples of non-metric distance measures are the chamfer distance [26], shape context matching [1], dynamic time warping [25], or the Kullback-Leibler (KL) distance [27]. Methods that are designed for general metric spaces can still be applied when the distance measure is non-metric. However, methods that are exact for metric spaces become inexact in non-metric spaces, and no theoretical guarantees of performance can be made.

A method explicitly designed for indexing non-metric spaces is DynDex [41], which is designed for a specific non-metric distance measure, and is not applicable to arbitrary spaces. SASH [42] is a method that can be used in both metric and non-metric spaces, and can be applied in extremely high-dimensional settings. An alternative method is proposed by Skopal in [43]. In that method, distances are directly modified in a nonlinear way, to become more or less metric, i.e., conform more or less with the triangle inequality. That method can be combined with any distance-based indexing scheme and is orthogonal to such schemes, including the method proposed in this paper.

LSH [9], [8] is the method most closely related to DBH, the method proposed in this paper. As pointed out in the introduction, the key difference is that LSH can only be applied to specific spaces, where a family of locality sensitive hashing functions is available. The formulation of DBH is distance-based, and thus DBH can be applied for indexing arbitrary distance measures. The remainder of the paper describes DBH in detail, highlighting similarities and differences between LSH and DBH.

III. Locality Sensitive Hashing

Let $\mathbb{X}$ be a space of objects, to which database and query objects belong. Let D be a distance measure defined on $\mathbb{X}$. In this paper we also use notation $(\mathbb{X}, D)$ to jointly specify the space and distance measure. Let $\mathcal{H}$ be a family of hash functions $h : \mathbb{X} \rightarrow \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. As described in [8], $\mathcal{H}$ is called *locality sensitive* if there exist real numbers r_1, r_2, p_1, p_2 such that $r_1 < r_2$, $p_1 > p_2$, and for any $X_1, X_2 \in \mathbb{X}$:

$$D(X_1, X_2) < r_1 \Rightarrow \Pr_{h \in \mathcal{H}}(h(X_1) = h(X_2)) \geq p_1 \ . \quad (1)$$

$$D(X_1, X_2) > r_2 \Rightarrow \Pr_{h \in \mathcal{H}}(h(X_1) = h(X_2)) \leq p_2 \ . \quad (2)$$

Given a locality sensitive family $\mathcal{H}$, Locality Sensitive Hashing (LSH) indexing works as follows: first, we pick integers k and l. Then, we construct l hash functions $g_1, g_2, \ldots, g_l$, as concatenations of k functions chosen randomly from $\mathcal{H}$:

$$g_i(X) = (h_{i1}(X), h_{i2}(X), \ldots, h_{ik}(X)) \ . \quad (3)$$

Each database object X is stored in each of the l hash tables defined by the functions g_i. Given a query object $Q \in \mathbb{X}$, the retrieval process first identifies all database objects that fall in the same bucket as Q in at least one of the l hash tables, and then exact distances are measured between the query and those database objects.

As shown in [8], if k and l are chosen appropriately, then a near neighbor of Q is retrieved with high probability (note that LSH is *not* an exact indexing method, as it may produce the wrong result for some queries). The method can be applied both for near-neighbor retrieval (for range queries) and nearest-neighbor retrieval (for similarity queries). In Euclidean space $\mathbb{R}^d$, the time complexity of retrieval using LSH is linear in the dimensionality d and sublinear in the number n of database objects [9].

Applying the LSH framework to a specific space and distance measure requires identifying a locality sensitive family $\mathcal{H}$. Such families have been identified for certain spaces, such as vector spaces with L_p metrics [9], [8], or strings with a substitution-based distance measure [12], [13]. An improvement that can drastically reduce the memory requirements of LSH in Euclidean spaces is described in [44].

IV. Distance-Based Hashing

In this section we introduce Distance-Based Hashing (DBH), a method for applying hash-based indexing in arbitrary spaces and distance measures. In order to make our method applicable to arbitrary spaces, a key requirement is to use the distance measure as a black box. Therefore, the definition of the hash functions should only depend on distances between objects. To keep the method general, no additional assumptions are made about the distance measure. In particular, the distance measure is *not* assumed to have Euclidean or metric properties.

The first step in our formulation is to propose a family $\mathcal{H}$ of hash functions. These functions are indeed defined using only distances between objects, and thus they can be defined in arbitrary spaces. The second and final step is to introduce a framework for analyzing indexing performance and picking parameters. We shall see that the proposed family $\mathcal{H}$ of hash functions is *not* always locality sensitive (depending on the space and distance measure), and therefore our method cannot be analyzed using the LSH framework. Consequently, we introduce a different framework, whereby indexing behavior is analyzed using statistical data collected from sample objects of $\mathbb{X}$.

A. A Distance-Based Family of Hash Functions

In existing literature, several methods have been proposed for defining functions that map an arbitrary space $(\mathbb{X}, D)$ into the real line $\mathbb{R}$. An example is the pseudo line projections proposed in [38]: given two arbitrary objects $X_1, X_2 \in \mathbb{X}$, we define a "line projection" function $F^{X_1,X_2} : \mathbb{X} \rightarrow \mathbb{R}$ as follows:

$$F^{X_1,X_2}(X) = \frac{D(X,X_1)^2 + D(X_1,X_2)^2 - D(X,X_2)^2}{2D(X_1,X_2)} \ . \tag{4}$$

If $(\mathbb{X}, D)$ is a Euclidean space, then $F^{X_1,X_2}(X)$ computes the projection of point X on the unique line defined by points X_1 and X_2. If $\mathbb{X}$ is a general non-Euclidean space, then $F^{X_1,X_2}(X)$ does not have a geometric interpretation. However, as long as a distance measure D is available, F^{X_1,X_2} can still be defined and provides a simple way to project $\mathbb{X}$ into $\mathbb{R}$.

We should note that the family of functions defined using Equation 4 is a very rich family. Any pair of objects defines a different function. Given a database $\mathbb{U}$ of n objects, we can define about $n^2/2$ unique functions by applying Equation 4 to pairs of objects from $\mathbb{U}$.

Functions defined using Equation 4 are real-valued, whereas hash functions need to be discrete-valued. We can easily obtain discrete-valued hash functions from F^{X_1,X_2} using thresholds $t_1, t_2 \in \mathbb{R}$:

$$F^{X_1,X_2}_{t_1,t_2}(X) = \begin{cases} 0 & \text{if } F^{X_1,X_2}(X) \in [t_1, t_2] \ . \\ 1 & \text{otherwise} \ . \end{cases} \tag{5}$$

In practice, t_1 and t_2 should be chosen so that $F^{X_1,X_2}_{t_1,t_2}(X)$ maps approximately half the objects in $\mathbb{X}$ to 0 and half to 1, so that we can build balanced hash tables. We can formalize this notion by defining, for each pair $X_1, X_2 \in \mathbb{X}$, the set $\mathbb{V}(X_1, X_2)$ of intervals $[t_1, t_2]$ such that $F^{X_1,X_2}_{t_1,t_2}(X)$ splits the space in half:

$$\mathbb{V}(X_1, X_2) = \{[t_1, t_2] | \Pr_{X \in \mathbb{X}}(F^{X_1,X_2}_{t_1,t_2}(X) = 0) = 0.5\} \ . \tag{6}$$

Note that, in most cases, for every t there exists a t' such that F^{X_1,X_2} maps half the objects of $\mathbb{X}$ either to $[t, t']$ or to [t', t]. For a set of n objects, there are $n/2$ ways to split those objects into two equal-sized subsets (if n is even) based on the choice of $[t_1, t_2] \in \mathbb{V}(X_1, X_2)$. One of several alternatives is to choose an interval $[t_1, \infty]$ such that $F^{X_1,X_2}(X)$ is less than t_1 for half the objects $X \in \mathbb{X}$. Another alternative is to choose an interval $[t_1, t_2]$ such that, using F^{X_1,X_2}, one sixth of the objects in X are mapped to a value less than t_1 and two sixths of the objects are mapped to a value greater than t_2. The set $\mathbb{V}(X_1, X_2)$ includes intervals for all these possible ways to split X into two equal subsets.

Using the above definitions, we are now ready to define a family $\mathcal{H}_{\text{DBH}}$ of hash functions for an arbitrary space $(\mathbb{X}, D)$:

$$\mathcal{H}_{\text{DBH}} = \{F^{X_1,X_2}_{t_1,t_2} | X_1, X_2 \in \mathbb{X}, [t_1, t_2] \in \mathbb{V}(X_1, X_2)\} \ . \tag{7}$$

Using random binary hash functions h sampled from $\mathcal{H}_{\text{DBH}}$ we can define k-bit hash functions g_i by applying Equation 3. This way, indexing and retrieval can be performed as in LSH, by:

- Choosing parameters k and l.
- Constructing l k-bit hash tables, and storing pointers to each database object at the appropriate l buckets.
- Comparing the query object with the database objects found in the l hash table buckets that the query is mapped to.

B. Differences between LSH and DBH

In the previous paragraphs we have defined a distance-based indexing scheme that uses hash functions. We call that method Distance-Based Hashing (DBH). What DBH has in common with LSH is the indexing structure: we define l hash tables using l hash functions g_i, and each g_i is a concatenation of k simple, discrete-valued (binary, in our case) functions $h \in \mathcal{H}_{\text{DBH}}$.

If the function family $\mathcal{H}_{\text{DBH}}$ were locality sensitive, then DBH would be a special case of LSH, and we would be able to use the LSH framework to optimally pick parameters k and l and provide guarantees of accuracy and efficiency. The main difference between DBH and LSH stems from the fact that we do not assume $\mathcal{H}_{\text{DBH}}$ to be locality sensitive. Whether $\mathcal{H}_{\text{DBH}}$ is actually locality sensitive or not depends on the underlying space and distance measure. Since we want to use DBH for indexing arbitrary spaces, we need to provide a method for analyzing performance without requiring $\mathcal{H}_{\text{DBH}}$ to be locality sensitive.

From an alternative perspective the difference between LSH and DBH is that applying LSH on a particular space requires knowledge of the geometry of that space. This knowledge is used to construct a family $\mathcal{H}$ of hash functions for that space and to prove that $\mathcal{H}$ is locality sensitive. If the goal is to design an indexing scheme for arbitrary spaces, then clearly no geometric information can be exploited, since arbitrary spaces have arbitrary geometries.

A simple example to illustrate that the family $\mathcal{H}_{\mathrm{DBH}}$ defined in Section IV-A is not always locality sensitive is the following: let us construct a finite space $(\mathbb{X}, D)$, by defining a distance matrix M, where entry $M_{i,j}$ is the distance $D(X_i, X_j)$ between the i-th and j-th object of $\mathbb{X}$. We set the diagonal entries $M_{i,i}$ to zero, we set all off-diagonal entries to random numbers from the interval $[1, 2]$, and we enforce that M be symmetric. Under that construction, space $(\mathbb{X}, D)$ is metric, as it satisfies symmetry and the triangle inequality.

In such a scenario, for any two objects $X_i, X_j \in \mathbb{X}$, the probability $\mathrm{Pr}_{h \in \mathcal{H}_{\mathrm{DBH}}}(h(X_i) = h(X_j))$ does not depend at all on the distance between X_i and X_j, and in practice $\mathrm{Pr}_{h \in \mathcal{H}_{\mathrm{DBH}}}(h(X_i) = h(X_j))$ is expected to be very close to 0.5, especially as the size of $\mathbb{X}$ becomes larger. Consequently, regardless of our choice of r_1 and r_2, there is no reason for appropriate p_1, p_2 to exist so as to satisfy the locality sensitive conditions expressed in Equations 1 and 2.

More generally, the random manner in which we constructed matrix M violates the fundamental assumption of any distance-based indexing method: the assumption that knowing $D(X_i, X_j)$ and $D(X_j, X_k)$ provides useful information/constraints about $D(X_i, X_k)$. The reason that distance-based methods work in practice is that, in many metric and nonmetric spaces of interest, distances are indeed not random, and knowing distances between some pairs of objects we can obtain useful information about distances between other pairs of objects.

Based on the above discussion, designing a useful distance-based indexing method requires identifying and exploiting the information that distances between objects provide, when such information is indeed present. When geometric constraints (such as Euclidean properties and/or the triangle inequality) are not available, we can still exploit statistical information obtained from sample data, i.e., from known objects sampled from the space of interest. We now proceed to describe how to obtain and use such statistical information in the context of DBH.

C. Statistical Analysis of DBH

An important question in analyzing any indexing scheme is identifying the factors that determine indexing performance, i.e., the factors that determine:

- Retrieval accuracy: how often is the true nearest neighbor retrieved using this indexing scheme?
- Retrieval efficiency: how much time does nearest neighbor retrieval take? What fraction of the database is pruned by the indexing scheme?

We now proceed to perform this analysis for DBH.

As before, let $(\mathbb{X}, D)$ be the underlying space and distance measure. Let $\mathbb{U} \subset \mathbb{X}$ be a database of objects from $\mathbb{X}$. Let $\mathcal{H}_{\mathrm{DBH}}$ be the family of binary hash functions defined in Equation 7. A key quantity for analyzing the behavior of DBH is the probability $C(X_1, X_2)$ of collision between any two objects of $\mathbb{X}$ over all binary hash functions in $\mathcal{H}_{\mathrm{DBH}}$:

$$C(X_1, X_2) = \mathrm{Pr}_{h \in \mathcal{H}_{\mathrm{DBH}}}(h(X_1) = h(X_2)) \ . \qquad (8)$$

Given family $\mathcal{H}_{\mathrm{DBH}}$ and the two objects X_1 and X_2, quantity $C(X_1, X_2)$ can be measured directly by applying all functions $h \in \mathcal{H}_{\mathrm{DBH}}$ to X_1 and X_2, if $\mathcal{H}_{\mathrm{DBH}}$ is finite. Alternatively, $C(X_1, X_2)$ can be estimated by applying only a sample of functions $h \in \mathcal{H}_{\mathrm{DBH}}$ to X_1 and X_2.

Suppose that we have chosen parameters k and l, and that we construct l k-bit hash tables by choosing randomly, uniformly, and with replacement, kl functions from $\mathcal{H}_{\mathrm{DBH}}$. The probability $C_k(X_1, X_2)$ of collision between two objects on a k-bit hash table is:

$$C_k(X_1, X_2) = C(X_1, X_2)^k \ . \qquad (9)$$

Finally, the probability $C_{k,l}(X_1, X_2)$ that two objects collide in at least one of the l hash tables is:

$$C_{k,l}(X_1, X_2) = 1 - (1 - C(X_1, X_2)^k)^l \ . \qquad (10)$$

Suppose that we have a database $\mathbb{U} \subset \mathbb{X}$ of finite size $n = |\mathbb{U}|$, and let $Q \in \mathbb{X}$ be a query object. We denote by $N(Q)$ the nearest neighbor of Q in $\mathbb{U}$. The probability that we will successfully retrieve $N(Q)$ using DBH is simply $C_{k,l}(Q, N(Q))$. The accuracy of DBH, i.e., the probability over all queries Q that we will retrieve the nearest neighbor $N(Q)$ is:

$$\mathrm{Accuracy}_{k,l} = \int_{Q \in \mathbb{X}} C_{k,l}(Q, N(Q)) \mathrm{Pr}(Q) dQ \ , \qquad (11)$$

where $\mathrm{Pr}(Q)$ is the probability density of Q being chosen as a query. This probability density is assumed to be uniform in the rest of this paper.

Quantity $\mathrm{Accuracy}_{k,l}$ can be easily estimated by:

1) sampling queries $Q \in \mathbb{X}$,
2) finding the nearest neighbors $N(Q)$ of those queries in the database $\mathbb{U}$,
3) estimating $C(Q, N(Q))$ for each sample Q by sampling from $\mathcal{H}_{\mathrm{DBH}}$,
4) using the estimated $C(Q, N(Q))$, and applying Equations 9 and 10 to compute $C_{k,l}(Q, N(Q))$ for each sample Q, and
5) computing the average value of $C_{k,l}(Q, N(Q))$ over all sample queries Q.

Besides accuracy, the other important performance measure for DBH is efficiency. In particular, we want to know how many database objects we need to consider for each query using DBH. Naturally, in brute force search we need to consider every single database object. The expected number of database objects we need to consider for a query Q is denoted as $\mathrm{LookupCost}(Q)$ and is simply the expected number of objects that fall in the same bucket with Q in at least one of the l hash tables. This quantity can be computed as:

$$\text{LookupCost}_{k,l}(Q) = \sum_{X \in \mathbb{U}} C_{k,l}(Q, X) \ . \tag{12}$$

For efficiency, an estimate for $\text{LookupCost}(Q)$ can be computed based on a sample of database objects, as opposed to computing $C_{k,l}(Q, X)$ for all database objects $X \in \mathbb{U}$.

An additional cost incurred by retrieval using DBH is the cost of computing the outputs $g_i(Q)$ of the l k-bit hash functions g_i. Overall, we need to apply kl binary hash functions $h \in \mathcal{H}_{\text{DBH}}$ on Q. Since each such function h is of the form specified in Equation 5, computing such an $h(Q)$ involves computing the distances $D(Q, X_1)$ and $D(Q, X_2)$ between the query and the two objects X_1 and X_2 used to define h. We denote by $\text{HashCost}_{k,l}$ the number of such distances we need to compute, in order to compute $h(Q)$ for all binary hash functions. Note that $\text{HashCost}_{k,l}$ is independent of the query Q, as $\text{HashCost}_{k,l}$ is simply the number of unique objects used as X_1 and X_2 in the definitions of the kl binary hash functions h. In the worst case, $\text{HashCost}_{k,l} = 2kl$, but in practice $\text{HashCost}_{k,l}$ can be smaller because the same object X can be used as X_1 or X_2 in the definitions of multiple binary hash functions h.

The total cost $\text{Cost}_{k,l}(Q)$ of processing a query is therefore the sum of the two separate costs:

$$\text{Cost}_{k,l}(Q) = \text{LookupCost}_{k,l}(Q) + \text{HashCost}_{k,l} \ . \tag{13}$$

Finally, the average query cost can be computed using sample queries, as was done for computing indexing accuracy. In particular:

$$\text{Cost}_{k,l} = \int_{Q \in \mathbb{X}} \text{Cost}_{k,l}(Q)\text{Pr}(Q)dQ \ . \tag{14}$$

In conclusion, the accuracy and efficiency of DBH, given parameters k and l, can be measured by sampling from the space of queries, sampling from the set of database objects, and sampling from the set $\mathcal{H}_{\text{DBH}}$ of binary hash functions.

D. Finding Optimal Parameters

Given parameter k, clearly indexing accuracy increases and efficiency decreases as we increase l. Consequently, given a desired retrieval accuracy, and given k, we can choose l by computing $\text{Accuracy}_{k,l}$ for $l = 1, 2, \ldots$ until we identify an l that yields the desired accuracy. Instead of successively measuring accuracy for each l, binary search can also be used, as a more efficient method for identifying the smallest l that yields the desired accuracy.

To find the optimal k we repeat the above process (of searching for an l given k) for different values $k = 1, 2, \ldots$. Different pairs of k, l that yield roughly the same indexing accuracy $\text{Accuracy}_{k,l}$ are likely to yield different costs $\text{Cost}_{k,l}$. Thus it is beneficial to choose the combination of k, l that, while achieving the desired accuracy, minimizes $\text{Cost}_{k,l}$. In practice, for a given accuracy, as we consider $k = 1, 2, \ldots$, efficiency typically improves up to a point and then it starts decreasing. Therefore, the optimal k can be identified as the last k for which efficiency improves.

In summary, given a desired retrieval accuracy rate, the optimal parameters k and l can be computed by searching over possible k and l and identifying the combination that, while yielding the desired accuracy, also maximizes efficiency. The accuracy and efficiency attained for each k, l pair is estimated as described in Section IV-C. Computing the optimal k and l is naturally done off-line, as a preprocessing step, and the costs of that computation have no bearing on the cost $\text{Cost}_{k,l}$ of the online retrieval stage.

V. Additional Optimizations

The previous section described a complete implementation of DBH. In this section we consider some practical methods for further improving performance. In particular, we describe a way to apply DBH in a hierarchical manner, using multiple pairs of (k, l) parameters, and we also describe a practical method for drastically reducing $\text{HashCost}_{k,l}$.

A. Applying DBH Hierarchically

The accuracy and efficiency of DBH for a particular query object Q essentially depends on the collision rate $C(Q, N(Q))$ between the query and its nearest neighbor, and the collision rates $C(Q, X)$ between Q and the rest of the database objects $X \in \mathbb{U}$. In an arbitrary space $\mathbb{X}$, without a priori knowledge of the geometry of that space, these collision rates can only be estimated statistically, and they can differ widely for different query objects.

The key motivation for designing a hierarchical version of DBH is the observation that, typically, different choices of k and l may be optimal for different query objects. Empirically, we have found that the optimal choice of k and l depends mainly on the distance $D(Q, N(Q))$. This correlation makes sense intuitively: the closer two objects are to each other the more likely it is that these objects are mapped to the same bit by a random binary hash function. Therefore, as $D(Q, N(Q))$ decreases, we expect the optimal parameters k and l for that query object to lead to increasingly fewer collisions for the same indexing accuracy.

Based on the above observations, a natural strategy is to create multiple DBH indexes, so that each index is optimized for a different set of queries and corresponds to a different choice of parameters k, l. In particular, we rank query objects Q according to $D(Q, N(Q))$, and we divide the space $\mathbb{X}$ of possible queries into disjoint subsets $\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_s$, so that $\mathbb{X}_i$ contains queries ranked in the top $(i-1)/s$ to i/s percentiles according to $D(Q, N(Q))$. Then, given the database $\mathbb{U}$ and the desired accuracy rate, we choose optimal parameters k_i and l_i for each query set $\mathbb{X}_i$, and we create a DBH index structure for that query set. We denote by D_i the smallest value such that for all objects $Q \in \mathbb{X}_i$ it holds that $D(Q, N(Q)) \leq D_i$.

Naturally, at runtime, given a previously unseen query object Q, we cannot know what $\mathbb{X}_i$ Q belongs to, since we do not know $D(Q, N(Q))$. What we can do is perform nearest neighbor retrieval successively using the DBH indexes created for $\mathbb{X}_1, \mathbb{X}_2, \ldots$ If using the DBH index created for $\mathbb{X}_i$ we retrieve a database object X such that $D(Q, X) \leq D_i$, then we know that $D(Q, N(Q)) \leq D(Q, X) \leq D_i$. In that case, the retrieval process does not proceed to the DBH index for $\mathbb{X}_{i+1}$, and the system simply returns the nearest neighbor found so far, using the DBH indexes for $\mathbb{X}_1, \ldots, \mathbb{X}_i$.

In practice, what we typically observe with this hierarchical scheme is this: the first DBH indexes, designed for queries with small $D(Q, N(Q))$, successfully retrieve (at the desired accuracy rate) the nearest neighbors for such queries, while achieving a lookup cost much lower than that of using a single global DBH index. For query objects Q with large $D(Q, N(Q))$, in addition to the lookup cost incurred while using the DBH index for that particular $D(Q, N(Q))$, the hierarchical process also incurs the lookup cost of using the previous DBH indexes as well. However, we expect this additional lookup cost to be small, since the previous DBH indexes typically lead to significantly fewer collisions for objects with large $D(Q, N(Q))$. So, overall, compared to using a global DBH index, the hierarchical scheme should significantly improve efficiency for queries with low $D(Q, N(Q))$, and only mildly decrease efficiency for queries with high $D(Q, N(Q))$.

B. Reducing the Hashing Cost

As described in Section IV-C, the hashing cost $\mathrm{HashCost}_{k,l}$ is the number of unique objects used as X_1 and X_2 in the definitions of the kl binary functions needed to construct the DBH index. If those kl binary functions are picked randomly from the space of all possible such functions, then we expect $\mathrm{HashCost}_{k,l}$ to be close to $2kl$. In practice, we can significantly reduce this cost, by changing the definition of $\mathcal{H}_{\mathrm{DBH}}$.

In Section IV-A we defined $\mathcal{H}_{\mathrm{DBH}}$ to be the set of all possible functions $F_{t_1,t_2}^{X_1,X_2}$ defined using any $X_1, X_2 \in \mathbb{X}$. In practice, however, we can obtain a sufficiently large and rich family $\mathcal{H}_{\mathrm{DBH}}$ using a relatively small subset $\mathbb{X}_{\mathrm{small}} \subset \mathbb{X}$:

$$\mathcal{H}_{\mathrm{DBH}} = \{F_{t_1,t_2}^{X_1,X_2} \mid X_1, X_2 \in \mathbb{X}_{\mathrm{small}}, \\ [t_1, t_2] \in \mathbb{V}(X_1, X_2)\} \ . \quad (15)$$

If we use the above definition, the number of functions in $\mathcal{H}_{\mathrm{DBH}}$ is at least equal to the number of unique pairs X_1, X_2 we can choose from $\mathbb{X}_{\mathrm{small}}$, and is actually larger in practice, since in addition to choosing X_1, X_2 we can also choose an interval $[t_1, t_2]$. At any rate, the size of $\mathcal{H}_{\mathrm{DBH}}$ is quadratic to the size of $\mathbb{X}_{\mathrm{small}}$. At the same time, regardless of the choice of parameters k, l, the hashing cost $\mathrm{HashCost}_{k,l}$ can never exceed the size of $\mathbb{X}_{\mathrm{small}}$, since only elements of $\mathbb{X}_{\mathrm{small}}$ are used to define functions in $\mathcal{H}_{\mathrm{DBH}}$.

In practice, we have found that good results can be obtained with sets $\mathbb{X}_{\mathrm{small}}$ containing as few as 50 or 100 elements. The significance of this is that, in practice, the hashing cost is bounded by a relatively small number. Furthermore, the hashing cost actually becomes increasingly negligible as the database becomes larger and the size of $\mathbb{X}_{\mathrm{small}}$ remains fixed, since the lookup cost starts dominating the total cost of processing a query.

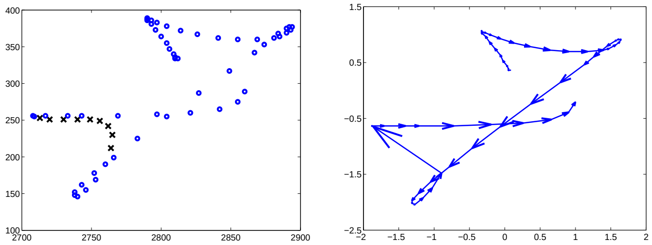

Fig. 1. Left: Example of a "seven" in the UNIPEN data set. Circles denote "pen-down" locations, x's denote "pen-up" locations. Right: The same example, after preprocessing.

VI. Experiments

In the experiments we evaluate DBH by applying it to three different real-world data sets: the isolated digits benchmark (category 1a) of the UNIPEN Train-R01/V07 online handwriting database [45] with dynamic time warping [46] as the distance measure, the MNIST database of handwritten digits [47] with shape context matching [1] as the distance measure, and a database of hand images with the chamfer distance as the distance measure. We also compare DBH with VP-trees [30], a well-known distance-based indexing method for arbitrary spaces. We modified VP trees as described in [36] so as to get different trade-offs between accuracy and efficiency. We should note that, in all three data sets, the underlying distance measures are not metric, and therefore VP trees cannot guarantee perfect accuracy.

A. Datasets

Here we provide details about each of the datasets used in the experiments. We should specify in advance that, in all datasets and experiments, the set of queries used to measure performance (retrieval accuracy and retrieval efficiency) was completely disjoint from the database and from the set of sample queries used to pick optimal k and l parameters during DBH construction. Specifically, the set of queries used to measure performance was completely disjoint from the sample queries that were used, offline, in Equations 11 and 14 to estimate $\mathrm{Accuracy}_{k,l}$ and $\mathrm{Cost}_{k,l}$.

The UNIPEN data set. We use the isolated digits benchmark (category 1a) of the UNIPEN Train-R01/V07 online handwriting database [45], which consists of 15,953 digit examples (see Figure 1). The digits have been randomly and disjointly divided into training and test sets with a 2:1 ratio (or 10,630:5,323 examples). We use the training set as our database, and the test set as our set of queries. The target application for this dataset is automatic real-time recognition of the digit corresponding to each query. The distance measure D used is dynamic time warping [46]. On an AMD Athlon 2.0GHz processor, we can compute on average 890 DTW distances per second. Therefore, nearest neighbor classification using brute-force search takes about 12 seconds per query.

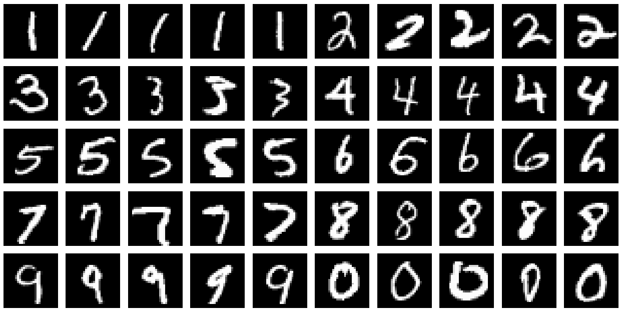

Fig. 2. Example images from the MNIST dataset of handwritten digits.

The nearest neighbor error obtained using brute-force search is 2.05%.

The MNIST data set. The well-known MNIST dataset of handwritten digits [47] contains 60,000 training images, which we use as the database, and 10,000 test images, which we use as our set of queries. Each image is a 28x28 image displaying an isolated digit between 0 and 9. Example images are shown in Figure 2. The distance measure that we use in this dataset is shape context matching [1], which involves using the Hungarian algorithm to find optimal one-to-one correspondences between features in the two images. The time complexity of the Hungarian algorithm is cubic to the number of image features. As reported in [48], nearest neighbor classification using shape context matching yields an error rate of 0.54%. As can be seen on the MNIST web site (`http://yann.lecun.com/exdb/mnist/`), shape context matching outperforms in accuracy a large number of other methods that have been applied to the MNIST dataset.

Using our own heavily optimized C++ implementation of shape context matching, and running on an AMD Opteron 2.2GHz processor, we can compute on average 15 shape context distances per second. As a result, using brute force search to find the nearest neighbors of a query takes on average approximately 66 minutes when using the full database of 60,000 images.

The hand image data set. This dataset consists of a database of 80,640 synthetic images of hands, generated using the Poser 5 software [49], and a test set of 710 real images of hands, used as queries. Both the database images and the query images display the hand in one of 20 different 3D handshape configurations. Those configurations are shown in Figure 3. For each of the 20 different handshapes, the database contains 4,032 database images that correspond to different 3D orientations of the hand, for a total number of 80,640 images. Figure 4 displays example images of a single handshape in different 3D orientations.

The query images are obtained from video sequences of a native ASL signer, and hand locations were extracted from those sequences automatically using the method described in [50]. The distance measure that we use to compare images is the chamfer distance [26]. On an AMD Athlon processor running at 2.0GHz, we can compute on average 715 chamfer distances per second. Consequently, finding the nearest neighbors of each query using brute force search takes about 112 seconds.

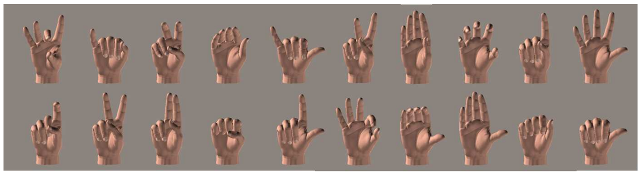

Fig. 3. The 20 handshapes used in the ASL handshape dataset.

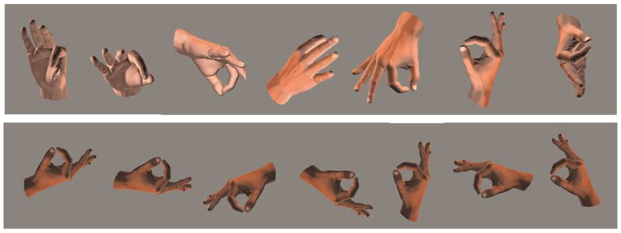

Fig. 4. Examples of different appearance of a fixed 3D hand shape, corresponding to different 3D orientations of the shape.

B. Implementation Details

For each data set we constructed a family $\mathcal{H}_{\text{DBH}}$ of binary hash functions as described in Section V-B. We first constructed a set $\mathbb{X}_{\text{small}}$ by picking randomly 100 database objects. Then, for each pair of objects $X_1, X_2 \in \mathbb{X}_{\text{small}}$ we created a binary hash function by applying Equation 5 and choosing randomly an interval $[t_1, t_2] \in \mathbb{V}(X_1, X_2)$. As a result, $\mathcal{H}_{\text{DBH}}$ contained one binary function for each pair of objects in $\mathbb{X}_{\text{small}}$, for a total of 4950 functions.

To estimate retrieval accuracy using Equation 11, we used 10,000 database objects as sample queries. To estimate the lookup cost using Equation 12 we used the same 10,000 database objects as both sample queries (Q in Equation 12) and sample database objects (X in Equation 12). The retrieval performance attained by each pair k, l of parameters was estimated by applying Equations 11 and 14, and thus the optimal k, l was identified for each desired retrieval accuracy rate.

We should emphasize that Equations 11, 12 and 14 were only used in the offline stage to choose optimal k, l parameters. The accuracy and efficiency values shown in Figure 5 were measured experimentally using previously unseen queries, that were completely disjoint from the samples used to estimate the optimal k, l parameters.

For the hierarchical version of DBH, described in Section V-A, we used $s = 5$ for all data sets, i.e., the hierarchical DBH index structure consisted of five separate DBH indexes, constructed using different choices for k and l.

C. Results

Figure 5 shows the results obtained on the three data sets for hierarchical DBH, single-level DBH (where a single, global DBH index is built), and VP-trees. For each data set we

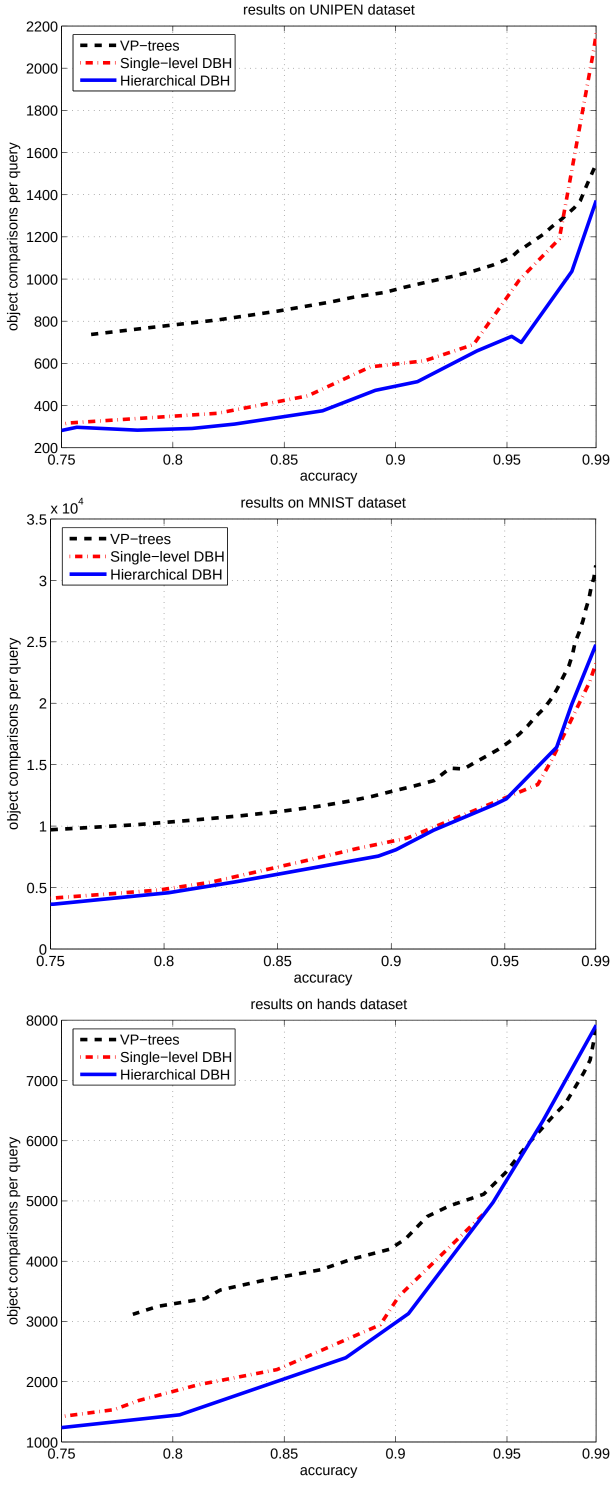

Fig. 5. Results on our three data sets, for VP-trees, single-level DBH, and hierarchical DBH. The x-axis is retrieval accuracy, i.e., the fraction of query objects for which the true nearest neighbor is retrieved. The y-axis is the average number of distances that need to be measured per query object.

plot retrieval time versus retrieval accuracy. Retrieval time is completely dominated by the number of distances we need to measure between the query object and database objects. The number of distances includes both the hashing cost and the lookup cost for each query. To convert the number of distances to actual retrieval time, one simply has to divide the number of distances by 890 distances/sec for UNIPEN, 15 distances/sec for MNIST, and 715 distances/sec for the hands data set. Retrieval accuracy is simply the fraction of query objects for which the true nearest neighbor was returned by the retrieval process.

As we see in Figure 5, hierarchical DBH gives overall the best trade-offs between efficiency and accuracy. The only exceptions are a very small part of the plot for the MNIST data set, where the single layer DBH gives slightly better results, and a small part of the plot for the hands data set, where VP-trees give slightly better results. On the other hand, on all three data sets, for the majority of accuracy settings, hierarchical DBH significantly outperforms VP-trees, and oftentimes DBH is more than twice as fast, or even close to three times as fast, compared to VP-trees. We also see that, almost always, hierarchical DBH performs somewhat better than single-level DBH.

Interestingly, the hands data set, where for high accuracy settings VP-trees perform slightly better, is the only data set that, strictly speaking, violates the assumption on which DBH optimization is based: the assumption that the sample queries that we use for estimating indexing performance are representative of the queries that are presented to the system at runtime. As explained in the description of the hands data set, the sample queries are database objects, which are synthetically generated and relatively clean and noise-free, whereas the query objects presented to the system at runtime are real images of hands, that contain significant amounts of noise.

In conclusion, DBH, and especially its hierarchical version, produces good trade-offs between retrieval accuracy and efficiency, and significantly outperforms VP-trees in our three real-world data sets. We should note and emphasize that all three data sets use non-metric distance measures, and no known method exists for applying LSH on those data sets.

VII. Conclusions

We have presented DBH, a novel method for approximate nearest neighbor retrieval in arbitrary spaces. DBH is a hashing method, that creates multiple hash tables into which database objects and query objects are mapped. A key feature of DBH is that the formulation is applicable to arbitrary spaces and distance measures. DBH is inspired by LSH, and a primary goal in developing DBH has been to create a method that allows some of the key concepts and benefits of LSH to be applied in arbitrary spaces.

The key difference between DBH and LSH is that LSH can only be applied to spaces where locality sensitive families of hashing functions have been demonstrated to exist; in contrast, DBH uses a family of binary hashing functions that is distance-based, and thus can be constructed in any space. As DBH indexing performance cannot be analyzed using geometric properties, performance analysis and optimization is based on statistics collected from sample data. In experiments with three real-world, non-metric data sets, DBH has yielded good trade-offs between retrieval accuracy and retrieval efficiency, and DBH has significantly outperformed VP-trees in all three data sets. Furthermore, no known method exists for applying LSH on those data sets, and this fact demonstrates the need for a distance-based hashing scheme that DBH addresses.

ACKNOWLEDGEMENTS

This work was supported by NSF grant IIS 0308213.

REFERENCES

[1] S. Belongie, J. Malik, and J. Puzicha, “Shape matching and object recognition using shape contexts,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 4, pp. 509–522, 2002.

[2] K. Grauman and T. J. Darrell, “Fast contour matching using approximate earth mover’s distance,” in *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 1, 2004, pp. 220–227.

[3] G. Shakhnarovich, P. Viola, and T. Darrell, “Fast pose estimation with parameter-sensitive hashing,” in *IEEE International Conference on Computer Vision (ICCV)*, 2003, pp. 750–757.

[4] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, “Basic local alignment search tool,” *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–10, 1990.

[5] B. Boeckmann, A. Bairoch, R. Apweiler, M. C. Blatter, A. Estreicher, E. Gasteiger, M. J. Martin, K. Michoud, C. O’Donovan, I. Phan, S. Pilbout, and M. Schneider, “The swiss-prot protein knowledgebase and its supplement TrEMBL in 2003.” *Nucleic Acids Research*, vol. 31, no. 1, pp. 365–370, 2003.

[6] 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*, vol. 28, no. 9, 1995.

[7] Y. Zhu and D. Shasha, “Warping indexes with envelope transforms for query by humming.” in *ACM International Conference on Management of Data (SIGMOD)*, 2003, pp. 181–192.

[8] A. Gionis, P. Indyk, and R. Motwani, “Similarity search in high dimensions via hashing,” in *International Conference on Very Large Databases (VLDB)*, 1999, pp. 518–529.

[9] A. Andoni and P. Indyk, “Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions,” in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2006, pp. 459–468.

[10] A. Frome, D. Huber, R. Kolluri, T. Bulow, and J. Malik, “Recognizing objects in range data using regional point descriptors,” in *European Conference on Computer Vision*, vol. 3, 2004, pp. 224–237.

[11] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, “Multi-probe lsh: Efficient indexing for high-dimensional similarity search,” in *International Conference on Very Large Databases (VLDB)*, 2007, pp. 950–961.

[12] A. Andoni and P. Indyk, “Efficient algorithms for substring near neighbor problem,” in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2006, pp. 1203–1212.

[13] J. Buhler, “Efficient large-scale sequence comparison by locality-sensitive hashing,” *Bioinformatics*, vol. 17, no. 5, 2001.

[14] C. Böhm, S. Berchtold, and D. A. Keim, “Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases,” *ACM Computing Surveys*, vol. 33, no. 3, pp. 322–373, 2001.

[15] G. Hjaltason and H. Samet, “Properties of embedding methods for similarity searching in metric spaces,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 5, pp. 530–549, 2003.

[16] G. R. Hjaltason and H. Samet, “Index-driven similarity search in metric spaces,” *ACM Transactions on Database Systems*, vol. 28, no. 4, pp. 517–580, 2003.

[17] D. A. White and R. Jain, “Similarity indexing: Algorithms and performance,” in *Storage and Retrieval for Image and Video Databases (SPIE)*, 1996, pp. 62–73.

[18] K. V. R. Kanth, D. Agrawal, and A. Singh, “Dimensionality reduction for similarity searching in dynamic databases,” in *ACM International Conference on Management of Data (SIGMOD)*, 1998, pp. 166–176.

[19] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold, “Clustering for approximate similarity search in high-dimensional spaces,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 792–808, 2002.

[20] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima, “The A-tree: An index structure for high-dimensional spaces using relative approximation,” in *International Conference on Very Large Data Bases*, 2000, pp. 516–526.

[21] E. Tuncel, H. Ferhatosmanoglu, and K. Rose, “VQ-index: An index structure for similarity searching in multimedia databases,” in *Proc. of ACM Multimedia*, 2002, pp. 543–552.

[22] R. Weber and K. Böhm, “Trading quality for time with nearest-neighbor search,” in *International Conference on Extending Database Technology: Advances in Database Technology*, 2000, pp. 21–35.

[23] R. Weber, H.-J. Schek, and S. Blott, “A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces,” in *International Conference on Very Large Data Bases*, 1998, pp. 194–205.

[24] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Physics*, vol. 10, no. 8, pp. 707–710, 1966.

[25] E. Keogh, “Exact indexing of dynamic time warping,” in *International Conference on Very Large Data Bases*, 2002, pp. 406–417.

[26] H. Barrow, J. Tenenbaum, R. Bolles, and H. Wolf, “Parametric correspondence and chamfer matching: Two new techniques for image matching,” in *International Joint Conference on Artificial Intelligence*, 1977, pp. 659–663.

[27] T. M. Cover and J. A. Thomas, *Elements of information theory*. New York, NY, USA: Wiley-Interscience, 1991.

[28] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, “Searching in metric spaces,” *ACM Computing Surveys*, vol. 33, no. 3, pp. 273–321, 2001.

[29] E. Chávez and G. Navarro, “Metric databases.” in *Encyclopedia of Database Technologies and Applications*, L. C. Rivero, J. H. Doorn, and V. E. Ferraggine, Eds. Idea Group, 2005, pp. 366–371.

[30] P. Yianilos, “Data structures and algorithms for nearest neighbor search in general metric spaces,” in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993, pp. 311–321.

[31] J. Uhlman, “Satisfying general proximity/similarity queries with metric trees,” *Information Processing Letters*, vol. 40, no. 4, pp. 175–179, 1991.

[32] T. Bozkaya and Z. Özsoyoglu, “Indexing large metric spaces for similarity search queries,” *ACM Transactions on Database Systems (TODS)*, vol. 24, no. 3, pp. 361–404, 1999.

[33] P. Ciaccia, M. Patella, and P. Zezula, “M-tree: An efficient access method for similarity search in metric spaces,” in *International Conference on Very Large Data Bases*, 1997, pp. 426–435.

[34] C. Traina, Jr., A. Traina, B. Seeger, and C. Faloutsos, “Slim-trees: High performance metric trees minimizing overlap between nodes,” in *International Conference on Extending Database Technology (EDBT)*, 2000, pp. 51–65.

[35] P. Zezula, P. Savino, G. Amato, and F. Rabitti, “Approximate similarity retrieval with M-trees,” *The VLDB Journal*, vol. 4, pp. 275–293, 1998.

[36] V. Athitsos, M. Hadjieleftheriou, G. Kollios, and S. Sclaroff, “Query-sensitive embeddings,” *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 2, 2007.

[37] J. Bourgain, “On Lipschitz embeddings of finite metric spaces in Hilbert space,” *Israel Journal of Mathematics*, vol. 52, pp. 46–52, 1985.

[38] C. Faloutsos and K. I. Lin, “FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets,” in *ACM International Conference on Management of Data (SIGMOD)*, 1995, pp. 163–174.

[39] G. Hristescu and M. Farach-Colton, “Cluster-preserving embedding of proteins,” CS Department, Rutgers University, Tech. Rep. 99-50, 1999.

[40] X. Wang, J. T. L. Wang, K. I. Lin, D. Shasha, B. A. Shapiro, and K. Zhang, “An index structure for data mining and clustering,” *Knowledge and Information Systems*, vol. 2, no. 2, pp. 161–184, 2000.

[41] K.-S. Goh, B. Li, and E. Chang, “DynDex: a dynamic and non-metric space indexer,” in *ACM International Conference on Multimedia*, 2002, pp. 466–475.

[42] M. E. Houle and J. Sakuma, “Fast approximate similarity search in extremely high-dimensional data sets,” in *IEEE International Conference on Data Engineering (ICDE)*, 2005, pp. 619–630.

[43] T. Skopal, “On fast non-metric similarity search by metric access methods,” in *International Conference on Extending Database Technology (EDBT)*, 2006, pp. 718–736.

[44] R. Panigrahy, “Entropy based nearest neighbor search in high dimensions,” in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2006, pp. 1186–1195.

[45] I. Guyon, L. Schomaker, and R. Plamondon, “Unipen project of online data exchange and recognizer benchmarks,” in *12th International Conference on Pattern Recognition*, 1994, pp. 29–33.

[46] J. B. Kruskall and M. Liberman, “The symmetric time warping algorithm: From continuous to discrete,” in *Time Warps*. Addison-Wesley, 1983.

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

[48] V. Athitsos, “Learning embeddings for indexing, retrieval, and classification, with applications to object and shape recognition in image databases,” Ph.D. dissertation, Boston University, 2006.

[49] *Poser 5 Reference Manual*, Curious Labs, Santa Cruz, CA, August 2002.

[50] Q. Yuan, S. Sclaroff, and V. Athitsos, “Automatic 2D hand tracking in video sequences.” in *IEEE Workshop on Applications of Computer Vision*, 2005, pp. 250–256.