Searching Massive Databases Using Locality Sensitive Hashing

Ting Li
Wal-mart Stores Inc.
702 SW 8th St
Bentonville, AR 72716
dragonflyting@hotmail.com

Haiying Shen*
Department of Electrical and Computer Engineering
Clemson University
Clemson, SC, 29634
shenh@clemson.edu

Abstract

Locality Sensitive Hashing (LSH) is a method of performing probabilitic dimension reduction of high-dimensional data. It can be used for approximate nearest neighbor search on a high-dimensional dataset. We first present a LSH-based similarity searching method. However, it needs large memory space and long processing time in a massive dataset. In addition, it is not effective on locating similar data in a very high-dimensional dataset. Further, it cannot easily adapt to data insertion and deletion. To address the problems, we then propose a new LSH-based similarity searching scheme that intelligently combines SHA-1 consistent hash function and Min-wise independent permutation into LSH (SMLSH). SMLSH effectively classifies information according to the similarity with reduced memory space requirement and in a very efficient manner. It can quickly locate similar data in a massive dataset. Experimental results show that SMLSH is both time and space efficient in comparison with LSH. It yields significant improvements on the effectiveness of similar searching over LSH in a massive dataset.

Keywords: Locality sensitive hashing, High-dimensional dataset, Similarity searching, Min-Wise permutations, Consistent hashing

1 Introduction

Driven by the tremendous growth of information in a massive dataset, there is an increasing need for an efficient similarity searching method that can locate desired information rapidly with low cost. An ideal similarity searching scheme should work well in a high-dimensional database. Specifically, it should locate nearly all the similar records of a query with a short query time and small memory requirement. In addition, it should also deal with data insertion and deletion. Many approaches have been proposed for similarity searching in high-dimensional databases. The approaches treat the records in a database as points in a high-dimensional space and each record is represented by a high-dimensional vector. Figure 1 gives an example of finding the nearest neighbor of a query point. There is a set of points in a two-dimensional space. Point q is a query point. From the figure, we can see point p is the closest point to q.

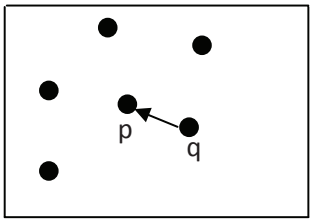

Figure 1: An example of finding nearest neighbor of a query point.

Distance measurement (i.e., Euclidean distance) can be used to decide the closeness of two records. Figure 2 shows an example of three records and their representation in Euclidean space. We see that record 1 and record 3 have three common keywords, while record 1 and record 2 have two common keywords.

Record ID	Record	Representation in Euclidean Space
1	Tom Smith 30 Male	<1, 1, 1, 1, 0, 0, 0>
2	John White 30 Male	<0, 0, 1, 1, 1, 1, 0>
3	Tom Smith 22 Male	<1, 1, 0, 1, 0, 0, 1>

Figure 2: An example of data records and their representation in Euclidean space.

By measuring the Euclidean distance between the records, we can find that record 1 is closer to record 3 than to record 2, i.e., record 3 is more similar to record 1 than record 2.

When querying in a massive dataset, many searching methods generate a high-dimensional vector for each object and then conduct the k-nearest neighbors searching [11]. However, such a method is not efficient when the dataset size is very large and the dimension is very high. Other methods relying on a tree structure (such as kd-trees, BDD-trees and vp-trees) require substantial memory space and time [1]. Sometimes, they are even less efficient than the linear search approach that compares a query record with each record in the dataset one at a time. Moreover, all these methods compare a query with records during the searching process to locate similar records, degrading the searching performance.

Locality sensitive hashing (LSH) is a well-known method that works faster than the linear search for finding nearest neighbors for high-dimensional datasets. LSH hashes high-dimensional vectors to one-dimensional integers, and uses the difference between the integers to measure the similarity between records. Indyk *et al.* [9] designed a LSH scheme based on p-Stable distributions, which can find the exact near neighbor in $O(\log n)$ time latency, and the data structure is up to 40 times faster than kd-tree [9].

In this entry, we first present an LSH-based similarity searching for a dataset. However, the LSH scheme is not effective on locating similar data in a massive dataset with a very high dimension space. In addition, it has low efficiency in terms of memory space and searching speed. An experimental study shows that the LSH scheme requires many hash tables in order to locate most nearest neighbors, and sometimes the LSH may require over a hundred hash tables to achieve reasonable accurate approximations [8]. Further, the LSH-based method requires all data records to have vectors with the same dimension, as it regards records as points in a multi-dimensional space. This makes it unable to easily adapt to data insertion and deletion. The data insertion and deletion may lead to keyword addition and deletion in the system, necessitating the re-generation of the high-dimensional vectors of all records, a very costly process.

To deal with the problems, we then present a SHA-1 consistent hash function and Min-wise independent permutation based LSH searching scheme (SMLSH) to achieve highly efficient similarity search in a massive dataset. By intelligently integrating SHA-1 and min-wise independent permutations into LSH, SMLSH assigns identifiers to each record and clusters similar records based on the identifiers. Rather than comparing a query with records in a dataset, it facilitates direct and fast mapping between a query and a group of records. The main difference with SMLSH and LSH is that SMLSH does not require that all records have the same dimension. Thus, SMLSH overcomes the aforementioned problems of LSH. False positive results are the records located as similar records but actually are not. LSH needs distance calculation to prune the false positive results, while SMLSH does not necessarily to have this refinement step since it incurs much less false positive results. We investigate the operation of LSH and SMLSH, and compare their performance by experiments. Experimental results show that SMLSH enhances LSH's searching efficiency dramatically.

The rest of this entry is structured as follows. Section 2 presents a brief review of related work. Section 3 introduces a LSH based similarity searching scheme, and Section 4 introduces min-wise independent permutations. Section 5 describes and analyzes the SMLSH searching scheme. Section 6 shows the performance of SMLSH in comparison with LSH. Section 7 concludes this entry with remarks on possible future work.

2 Approaches for similarity searching

The similarity searching problem is closely related to the nearest neighbor search problem, which has been studied by many researchers. Various indexing data structures have been proposed for nearest neighbor searching.

2.1 Tree structures

Some of the similarity searching methods rely on tree structures, such as R-tree, SS-tree and SR-tree. These data structures partition the data objects based on their similarity. Therefore, during a query, only a part of the data records have to be compared with the query record, which is more efficient than the linear search that compares a query with every data record in the database. Though these data structures can support nearest neighbor searching, they are not efficient in a large and high-dimensional database (i.e., the dimensionality is more than 20). The M-tree [4] was proposed to organize and search large datasets from a generic metric space, i.e., where object proximity is only defined by a distance function satisfying the positivity, symmetry, and triangle inequality postulates. The M-tree partitions objects on the basis of their relative distances measured by a specific distance function, and stores these objects into nodes that correspond to constrained regions of the metric space [4]. All data objects are stored in the leaf nodes of M-tree. The non-leaf nodes contain "routing objects" which describe the objects contained in the branches. For each routing object, there is a so-called covering radius of all its enclosing objects, and the distances to each child node are pre-computed. When a range querying is completed, sub-trees are pruned if the distance between the query object and the routing object is larger than the routing object's covering radius plus the query radius. Because a lot of the distances are pre-computed, the query speed is dramatically increased. The main problem is the overlap between different routing objects in the same level.

2.2 Vector approximation file

Another kind of similarity searching method is the vector approximation file (VA-file) [17], which can reduce the amount of data that must be read during similarity searches. It divides the data space into grids and creates an approximation for each data object that fall into a grid. When searching for the near neighbors, the VA-file sequentially scans the file containing these approximations, which is smaller than the size of the original data file. This allows most of the VA-file's disk accesses to be sequential, which are much less costly than random disk accesses [6]. One drawback of this approach is that the VA-file requires a refinement step, where the original data file is accessed using random disk accesses [6].

2.3 Approximation tree

Approximation tree (A-tree) [11] has better performance than VA-file and SR-tree for high dimensional data searching. The A-tree is an index structure for similarity search of high-dimensional data. A-tree stores virtual bounding rectangles (VBRs), which contain and approximate minimum bounding rectangles (MBR) and data objects, respectively. MBR is a bounding box to bind data object. iDistance [19] partitions the data into different regions and defines a reference point for each partition. The data in each region is transformed into a single dimensional space based on their similarity with the reference point in the region. Finally, these points are indexed using a B+-tree structure and similarity search is performed in the one-dimensional space. As reported in [19], iDistance outperforms the M-tree and linear search.

2.4 Hashing

Hashing is a common approach to facilitate similarity search in high dimension databases, and spectral hashing [18] is one state-of-the-art work for data-aware hashing. Spectral hashing applies the machine learning techniques to minimize the semantic loss of hashed data resulting from embedding. However, the drawback of Spectral hashing lies in its limited applicability. As spectral hashing relies on Euclidean

distance to measure the similarity between two data records, and it requires that data points are from a Euclidean space and are uniformly distributed.

Most recently, much research also has been conducted on locality-sensitive hashing. Dasgupta *et al.* [5] proposed a new and simple method to speed up the widely-used Euclidean realization of LSH. At the heart of the method is a fast way to estimate the Euclidean distance between two d-dimensional vectors; this is achieved by the use of randomized Hadamard transforms in a non-linear setting. Traditionally, several LSH functions are concatenated to form a "static" compound hash function for building a hash table. Gan *et al.* [7] proposed to use a base of m single LSH functions to construct "dynamic" compound hash functions, and defined a new LSH scheme called Collision Counting LSH (C2LSH). In C2LSH, if the number of LSH functions under which a data object o collides with a query object q is greater than a pre-specified collision threshold, then o can be regarded as a good candidate of c-approximate nearest neighbors of q. Slaney and Casey [16] described an LSH technique that allows one to quickly find similar entries in large databases. This approach belongs to a novel and interesting class of algorithms that are known as randomized algorithms, which do not guarantee an exact answer but instead provide a high probability guarantee of returning correct answer or one close to it.

Recent work has also explored ways to embed high-dimensional features or complex distance functions into a low-dimensional Hamming space where items can be efficiently searched. However, existing methods do not apply for high-dimensional kernelized data when the underlying feature embedding for the kernel is unknown. Kulis and Grauman [10] showed how to generalize locality-sensitive hashing to accommodate arbitrary kernel functions, making it possible to preserve the algorithm's sub-linear time similarity search guarantees for a wide class of useful similarity functions. Semantic hashing [12] seeks compact binary codes of data-points so that the Hamming distance between codewords correlates with semantic similarity. Weiss *et al.* [18] showed that the problem of finding a best code for a given dataset is closely related to the problem of graph partitioning and can be shown to be NP hard. By relaxing the original problem, they obtained a spectral method whose solutions are simply a subset of thresholded eigenvectors of the graph Laplacian. Satuluri and Parthasarathy [13] proposed BayesLSH, a principled Bayesian algorithm for performing candidate pruning and similarity estimation using LSH. They also presented a simpler variant, BayesLSH-Lite, which calculates similarities exactly. BayesLSH can quickly prune away a large majority of the false positive candidate pairs. The quality of BayesLSH's output can be easily tuned and does not require any manual setting of the number of hashes to use for similarity estimation.

3 Locality Sensitive Hashing

In this section, we introduce LSH, LSH-based similarity searching method, and min-wise independent permutations.

LSH is an algorithm used for solving the approximate and exact near neighbor search in high dimensional spaces [9]. The main idea of the LSH is to use a special family of hash functions, called LSH functions, to hash points into buckets, such that the probability of collision is much higher for the objects which are close to each other in their high-dimensional space than for those which are far apart. A collision occurs when two points are in the same bucket. Then, query points can identify their near neighbors by using the hashed query points to retrieve the elements stored in the same buckets.

For a domain S of a set of points and distance measure D, the LSH family is defined as:

DEFINITION 1. A family $\mathcal{H} = \{h : S \rightarrow U\}$ is called (r_1, r_2, p_1, p_2) sensitive for D if for any point v, q belongs to S

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

where r_1, r_2, p_1, p_2 satisfy $p_1 < p_2$ and $r_1 < r_2$.

LSH is a dimension reduction technique that projects objects in a high-dimensional space to a lower-dimensional space while still preserving the relative distances among objects. Different LSH families can be used for different distance functions.

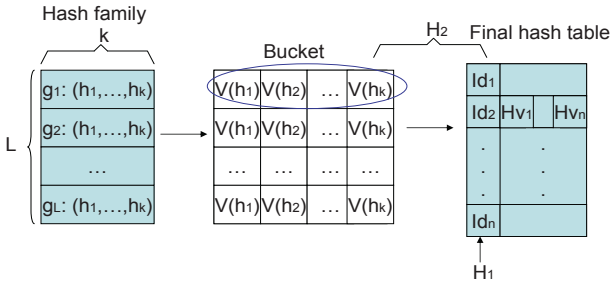

Figure 3: The process of LSH.

	V_1	V_2	V_3	V_4	q
Ann	1	1	0	0	1
Mike	0	0	1	0	0
John	0	0	0	1	0
Johnson	1	1	0	0	1
Smith	0	0	1	0	0
White	0	0	0	1	0
16	1	0	1	0	0
20	0	1	0	0	1
24	0	0	0	1	0
Female	1	1	0	0	1
Male	0	0	1	1	0
Ann248	1	0	0	0	0
168	0	1	0	0	1
1301	0	0	1	0	0
Dickson	1	0	0	0	0
Street	1	0	0	0	0
Garland	0	1	0	0	1
Hwy	0	0	1	0	0
Fayetteville	0	0	0	1	0
72701	0	0	0	1	0

Figure 4: Multi-dimensional keyword space.

Based on LSH on p-stable distribution [9], we develop a similarity searching method. Figure 3 shows the process of LSH. The hash function family of LSH has L groups of function functions, and each group has k hash functions. Given a data record, LSH applies the hash functions to the record to generate L buckets, and each bucket has k hash values. LSH uses hash function H_1 on the k hash values of each bucket to generate the location index of the record in the final hash table, and uses hash function H_2 on the k hash values of each bucket to generate the value of the record to store in the location. Finally, the record has L values stored in the final hash table. Given a query, LSH uses the same process to produce the L indices and values of the query, and finds similar records based on the indices, and identifies final similar records based on the stored values. Let us take an example to explain how the LSH-based similarity searching works. Assume that the records in a dataset are as follows:

Ann Johnson	16	Female	248 Dickson Street
Ann Johnson	20	Female	168 Garland
Mike Smith	16	Male	1301 Hwy
John White	24	Male	Fayetteville 72701

First, LSH constructs a keyword list which consists of all unique keywords in all records, with each keyword functioning as a dimension. The scheme then transforms these records into binary data based on the keyword list. Specifically, if a record contains the keyword, the dimension representing the keyword has the value 1, otherwise, it has the value 0. Figure 4 shows the process to determine the vector of each record. The number of dimensions of a record is the total length of the keyword list. Finally, the records are transformed to multi-dimensional vector:

v_1: 1 0 0 1 0 0 1 0 0 1 0 1 0 0 1 1 0 0 0 0

v_2: 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0

v_3: 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0

v_4: 0 0 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 1 1

As shown in Figure 3, LSH then produces the hash buckets $g_i(v)$ $(1 \leq i \leq L)$ for every record. Thereafter, LSH computes the hash value for every bucket. Finally, record v's hashed value by H_2 hash function, H_v, is stored in final hash tables pointed by the hashed value by H_1. Figure 5 shows the process of searching similar records of a query. If a query record q is:

Ann Johnson | 20 | Female | 168 Garland

Using the same procedure, q will be transformed to

q: 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0

Then, the index of q will be stored in the final hash tables through the same procedure. Consequently, the records that are in the same rows with q in hash table 1 to hash table L are similar records. In the example, v_2 and v_3 are in the similar record set. Finally, the Euclidean Space Distance between each located record and the query is computed to prune the results. A record will be removed from the located record set if its distance to the query is larger than R, which is a pre-defined threshold of distance.

The following formular is used to compute the Eu-

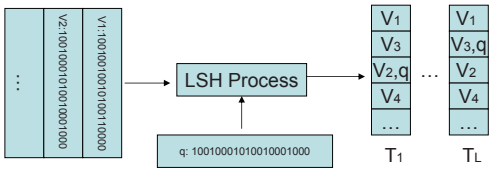

Figure 5: An example of LSH similarity searching.

clidean Space Distance between x_i and y_i.

$$d(x,y) = \| x - y \| = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}.$$

This distance calculation refinement phase is to prune false positive results that are located as similar records but actually are not.

From this example, we can see that LSH does not need to search the query in the entire dataset scope. It shrinks the searching scope to a group of records similar to the query, and conducts refinement. Given n records in a dataset, traditional methods based on tree structures need $O(\log n)$ time for a query, and linear searching methods need $O(n)$ time for query. LSH can locate the similar records in $O(L)$ time, where L is a constant. It means that LSH is more efficient in a massive dataset that has a large number of dimensions and records.

The drawback of LSH is large memory consumption. Because LSH needs to require a large number of hash tables to cover most near neighbors. For example, over 100 hash tables are needed in [8], and 583 hash tables are used in [3]. Because each hash table has as many entries as the number of data records in the database, the size of each hash table is decided by the size of the database. When the space requirement for the hash tables exceeds the main memory size, a disk I/O may be required for searching more hash tables, which causes query delay.

4 Min-Wise Independent Permutations

In this section, we introduce min-wise independent permutations. Broder *et al.* [2] defined that $\mathcal{F} \subseteq S_n$

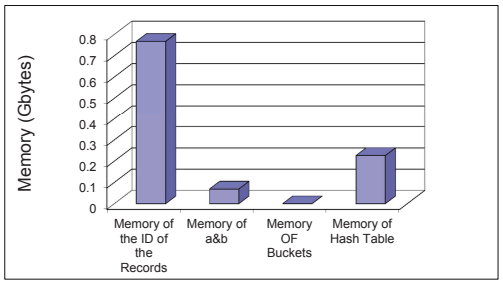

Figure 6: Memory consumption of LSH.

is min-wise independent if for any set $X \subseteq [n]$ and $x \in X$, when π is chosen at random in $\mathcal{F}$,

$$Pr(min\{\pi(X)\} = \pi(x)) = \frac{1}{\mid X \mid},$$

where Pr is the probability. All the elements of any fixed set X have an equal chance to become the minimum element of the image of X under π .

In [2], a family of hash functions $\mathcal{F}$ is said to be a LSH function family corresponding to similarity function $sim(A,B)$ if for all $h \in \mathcal{F}$ operating on two sets A and B, we have:

$$Pr_{h\in\mathcal{F}}[h(A) = h(B)] = sim(A,B),$$

where $sim(A,B) \in [0,1]$ is a similarity function. Min-wise independent permutations provide Jaccard index to measure the similarity

$$sim(A,B) = \frac{\mid A \cap B \mid}{\mid A \cup B \mid}.$$

For example, $U = \{a,b,c,d,e,f\}$, $S = \{a,b,c\}$, $S' = \{b,c,d\}$, and $S \subset U$ and $S' \subset U$. A random permutation π of the universe U is $\pi =< d,f,b,e,a,c >$. Because b contained in S is the first element that appears in π, b represents the smallest image of S under π, denoted by $b = min\{\pi(S)\}$. Because d contained in S' is the first element that appears in π, so d represents the smallest image of S' under π, denoted by $d = min\{\pi(S')\}$. $S \cap S' = \{b,c\}$ and $S \cup S' = \{a,b,c,d\}$. For a random permutation of the universe

$$U : \pi(U) = \{e, p_1, p_2, f, p_3, p_4\},$$

where p_1, p_2, p_3 and p_4 can be a, b, c and d in any order, if p_1 is from $\{b, c\}$, then $min\{\pi(S)\} = min\{\pi(S')\}$, and S and S' are similar. From

$$Pr(min\{\pi(S)\} = min\{\pi(S')\}) = \frac{\mid S \cap S' \mid}{\mid S \cup S' \mid},$$
$$= \frac{\mid \{b, c\} \mid}{\mid \{a, b, c, d\} \mid},$$

we can compute the similarity between S and S'.

5 SMLSH Searching Scheme

A massive dataset has a tremendous number of keywords, and a record may contain only a few keywords. As a result, in LSH, the identifier of a record may have a lot of 0s, and only a few 1s. This identifier sparsity leads to low effectiveness of Euclidean Space Distance measurement to quantify the closeness of two records. This is confirmed by our simulations results that the LSH returns many records that are not similar to the query even though all expected records are returned. We also observe that the memory required for the LSH scheme is mainly used to store the identifiers of records and the hash tables. Figure 6 shows the memory used for different objects in LSH.

5.1 Record Vector Construction

SMLSH reduces the false positive results and meanwhile reduces the memory for records and hash tables. It does not require all records have the same dimension. That is, it does not need to derive a vector for each record from a unified multi-dimensional space consisting of keywords.

The records in databases are usually described in string format. Therefore, the original data record cannot be used to do the computation. SMLSH first uses SHA-1 consistent hash function to generate an identifier for each keyword in a record. SHA stands for Secure Hash Algorithm, which includes five cryptographic hash functions. The hash algorithms can compute a fixed-length digital representation of an input data sequence of any length. SHA-1 hash function, which is one of the five cryptographic hash functions, has been employed in several widely used security applications and protocols, such as TLS (Transport Layer Security), SSL (Secure Sockets Layer) and IPsec (Internet Protocol Security). SHA-1 hash function is supposed to be collision-resistant, so it can be used to hash keywords into integers. Since SHA-1 distinguishes uppercase and lowercase keywords. SMLSH firstly changes all keywords to uppercase. As shown in the following, after changing all the keywords of a record into capital letters, SMLSH uses SHA-1 to hash all the capital-letter keywords to a set of integers:

Original record:

Ann	EDNA	Shelby	NC	0541

Uppercase record:

ANN	EDNA	SHELBY	NC	0541

Hashed record:

1945773335	628111516	2140641940
2015065058	125729831	

LSH requires that all record vectors have the same dimension to construct buckets with universal hash function. In LSH, the length of each record vector equals to the length of the keyword list consisting of all keywords in the dataset. In contrast, SMLSH does not require that all records have the same dimension. In SMLSH, the length of a record vector only equals to the number of keywords in itself. Thus, SMLSH reduces the memory of LSH for vectors. In SMLSH, the min-wise independent permutations are defined as:

$$\pi(x) = (ax + b) \bmod prime, \quad (1)$$

where a and b are random integers, $0 < a \leq prime$ and $0 \leq b \leq prime$.

Figure 7 shows an example of building buckets for a record. First, the keywords of the original record are represented as integer numbers by SHA-1 hash function. Second, for a pair of a and b values in Function (1), we get different $\pi(x)$ values for different keywords. The minimum number of $\pi(x)$, denoted by $\min\{\pi(x)\}$, is chosen. We then use the keyword corresponding to $\min\{\pi(x)\}$ as the value of an element in the buckets. We then continue to generate a new pair of a and b values, another $\min\{\pi(x)\}$s can be computed. This process will not stop until $n \times m$ $min\{\pi(x)\}$ values are calculated. Therefore, n buckets are built for a record, and each bucket has m

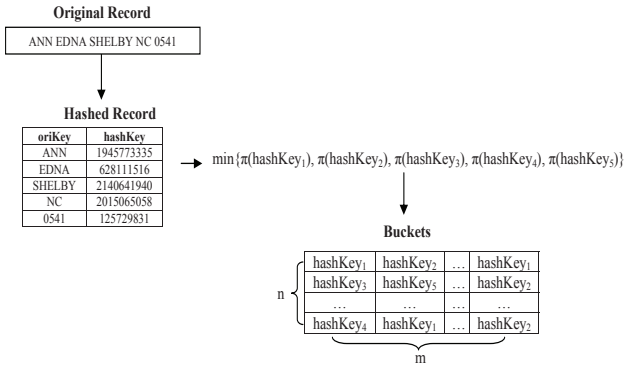

Figure 7: An example of building buckets for a record.

values. Algorithm 1 shows the procedure of bucket construction in SMLSH.

Algorithm 1. Bucket construction in SMLSH.

(1) determine $n \times m$ values of a and b
(2) **for** each k[i] **do** *//k[i] is one of the keywords of //a record*
(3) Use SHA-1 to hash k[i] into hashK[i]
(4) **for** each pair of a[p][q] and b[p][q] **do**
(5) $g[p][q] = (a[p][q] * hashK[i] + b[p][q]) \bmod prime$
(6) **if** i == 0 **then**
(7) min[p][q]=g[p][q]
(8) **else if** $g[p][q] < min[p][q]$ **then**
(9) min[p][q] = g[p][q]
(10) **endif**
(11) **endif**
(12) **endfor**
(13) **endfor**

5.2 Record Clustering

SMLSH makes n groups of m min-wise independent permutations. Applying the $m \times n$ hash values to a record, SMLSH constructs n buckets with each bucket having m hashed values. SMLSH then hashes each bucket to a hash value with similarity preservation and clusters the data records based on their

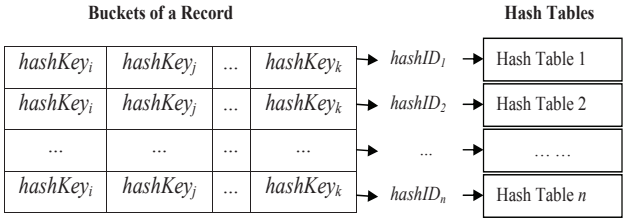

Figure 8: The process of finding locations for a record.

similarity. Specifically, SMLSH uses XOR operation on the elements of each bucket to get a final hash value. Consequently, each record has n final hashed values, denoted by $hashID_i$.

$$hashID_i = (min\{\pi_1(S')\}\ XOR\ min\{\pi_m(S')\}) \bmod tableSize, \quad (2)$$

where S' is a SHA-1 hased integer set, $1 \leq i \leq n$. Algorithm 2 shows the pseudo-code for the procedure of records clustering in SMLSH.

Algorithm 2. Record clustering in SMLSH.

(1) **for** each hashID[j] **do**
(2) hashID[j]=0
(3) **for** each min[j][t] **do**
(4) hashID[j]^=min[j][t]
(5) **endfor**
(6) hashID[j]=hashID[j]mod tableSize
(7) Insert the index of the record into the hash table
(8) **endfor**

Figure 8 presents the process of finding locations of a record. There are n buckets for each record. Each row of hashKeys in a bucket is used to calculate a final hash value for the bucket. Therefore, n bucket hash values are produced (i.e., $hashID_1$, , $hashID_n$). n hash tables are needed for saving all the buckets of the records in a database. Each hashID of a record representing the location of the record is stored in the corresponding hash table.

5.3 SMLSH Searching Process

When searching a record's similar records, SMLSH uses SHA-1 to hash the query record to integer representation. Then, SMLSH builds buckets for the query record. Base on the clustering algorithm mentioned above, SMLSH gets the n hashIDs for the query record. It then searches every location based on hashID in every hash table, exports all the records with the same hashID as the candidates of similar records of the query record. In order not to miss other similar records (i.e., reduce false negatives), at the same time, SMLSH continues to build new n buckets from each located record for further similar record search. Specifically, SMLSH generates n buckets from a located record using the method introduced previously. To generate the i^{th} bucket, it randomly chooses elements in the i^{th} bucket of the query record and in the i^{th} bucket of the located record. It then calculates the hashID of each newly constructed bucket and searches similar records using the above introduced method. As the new buckets are generated from the buckets of the query record and its similar record, the located records based on the new buckets may have a certain similarity with the query record.

Figure 9 shows the SMLSH similarity searching process. Let us say after computing the buckets of the query record, we get the first hashID equals 1. Therefore, SMLSH checks the records which hashID equals to 1 in the first hash table (i.e., HashTable1). As the figure shows, the hashID of record v equals to 1 in HashTable1. We generate n buckets from v. Then, we use the i^{th} bucket of records q and v to generate the i^{th} new bucket in the new group of n buckets. The elements in the i^{th} new bucket are randomly picked from the i^{th} bucket of record q and from the i^{th} bucket of record v. XOR operation is used to compute the hashIDs of the new buckets. According to the hashIDs of the new buckets, SMLSH searches the HashTable1 again to collect all the records having the same hashIDs and considers them as the candidates of the similar records of query record q. After finishing searching the first hash table for hashID1, SMLSH continues searching the hash tables for other hashIDs until finishing searching the n^{th} hash table

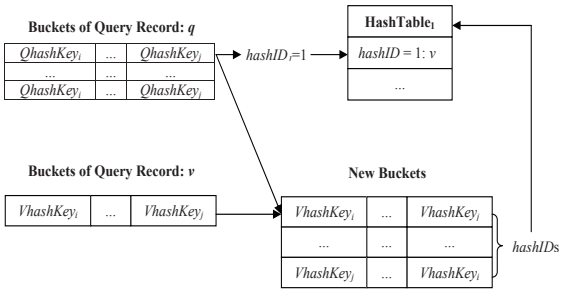

Figure 9: The process of similarity search.

for $hashID_n$.

Algorithm 3. Searching process in SMLSH.

(1) Calculate n hashIDs of the query record
(2) **for** each hashID[j] of the query record **do**
(3) Get the records v with hashID[j] in the j-th hash table
(4) **for** each record v[k] **do**
(5) Insert record v[k] into the similar record list
(6) Collect all the elements in j-th bucket of query record and record v[k]
(7) Randomly pick elements from the collection to build n new buckets
(8) Compute new hashIDs for the new buckets
(9) Retrieve the records with new hashIDs in the j-th hash table
(10) Insert the retrieved records into the similar record list
(11) **endfor**
(12) **endfor**
(13) Compute the similarity of the records in the similar record list
(14) Output the similar records with similarity greater than a threshold r

Algorithm 3 shows the pseudo-code of the searching process in SMLSH. For a record searching, SMLSH gets the hashIDs for the query record based on the algorithm. It then searches the hash table, exports all the records with the same hashID as the similar records of the query record. A range also can

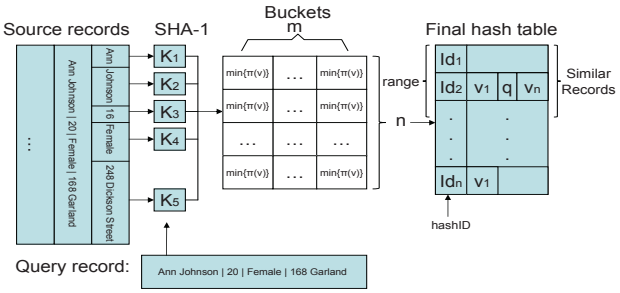

Figure 10: Similarity searching process of SMLSH.

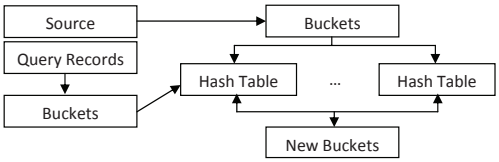

Figure 11: A high-level view of similarity searching process in SMLSH.

be set to enlarge the searching scope. With the range, the records with hashID[j] that satisfy condition:

$$| hashID[j] - RhashID | \leq range$$

are also checked, where $RhashID$ is the hashID of a query record. This method enlarges the searching scope of similar records, and increase the searching accuracy. Using the dataset example in Section 3 and assuming $range = 1$, Figure 10 shows the searching process of SMLSH.

5.4 Example of Similarity Searching in SMLSH

Figure 10 and Figure 11 present the similarity searching process in SMLSH. Source records are hashed to n hash tables based on their buckets. When searching the similar records of a query record, the query record needs to generate n buckets first. According to the hash values from the buckets, n hash tables are searched. During searching in the hash tables, new buckets are built and further searching in the hash tables is conducted.

THE HASH VALUES AND HASH IDS OF RECORDS

Record	SHA-1 Hash Values	hashID
v_1	895479561 | 1630694977 | 612003623 | 1870446154 | 669783043 | 537429199	132419788
		1416462664
v_2	786139273 | 1186247512 | 114561690 | 1658656342 | 288242920	0
		0
v_3	370869835 | 1937983344 | 114561690 | 1870446154 | 323411961 | 2010266570 | 1226489228	1687479347
		0
v_4	1002692496 | 1630694977 | 586341023 | 1870446154 | 847272969 | 458697817 | 300130902	179605007
		275941014
q	1945773335 | 1937983344 | 114561690 | 1658656342 | 323411961 | 537429199 | 1226489228	1687479347
		1906035119

Figure 12: The hash values and hash IDs of records.

Let us use an example to explain the similarity search in SMLSH. Given a database contains four records:

v_1: Tom White	16	Male	248 Main	
v_2: Lucy Oliver	20	Female	AR	
v_3: Mike Smith	20	Male	123 AR St.	
v_4: John White	24	Male	Little Rock	7201

A query record q is:

q: Ann Smith | 20 | Female | 123 AR St.
First, SMLSH transfers all the keywords to integers using SHA-1 hash function. Second, SMLSH builds two buckets for the records with each bucket having four hash values. Finally, Function (2) is used to hash the source records to two hash tables. When searching the similar records of query q, for each located record, SMLSH makes new buckets and computes hashIDs to continue searching similar records of query q. Table 12 shows the hash values and hash IDs of source records and query record.

Figure 13 presents the hashing results of Hash Table 1 and Hash Table 2. According to the hashIDs of query q, record v_3 is found in Hash Table 1 and records v_2 and v_3 are found in Hash Table 2. New buckets are generated by using the elements in the buckets of q, v_2 and v_3. Searching Hash Table 1 again based on the hashIDs of the new buckets, record v_2 is found. In Hash Table 2, record v_1 is found by using new buckets. Therefore, SMLSH combines the searching results from Hash Table 1 and Hash Table 2, and returns records v_1, v_2 and v_3 as candidates of the similar records.

To enhance the accuracy of returned similar records, refining can be conducted based on similar-

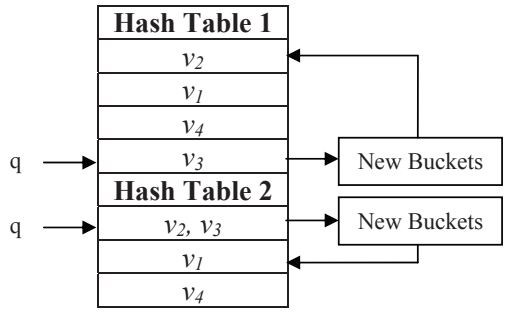

Figure 13: An example of SMLSH similarity searching.

ity. That is, the similarity between each returned record and the query record is calculated, the records whose similarities are less than a pre-defined threshold are removed from the list. Given two records A and B, the similarity of B to A is calculated as follows:

$$similarity = \frac{\mid A \cup B \mid}{\mid A \mid} \tag{3}$$

For example,

A: Ann	Johnson	16	Female
B: Ann	Johnson	20	Female

To A, the similarity of B is $\frac{3}{4} = 0.75$.

Then, in the example, the refinement phase filters the dissimilar records. The similarity between query q and record v_2 is 2/7, and the similarity between query q and record v_3 is 5/7. Records v_2 and v_3 are finally identified as similar records with query q.

6 Performance Evaluation

We implemented the SMLSH searching system on E2LSH 0.1 of MIT [20]. E2LSH 0.1 is a simulator for the high-dimensional near neighbor search based on LSH in the Euclidean space. Our testing sample dataset is obtained from Acxiom Corporation. After each record is transformed to 0/1 binary vector, the dimension of the record is 20,591. The number of source records was 10,000. We selected 97 query records randomly from the source records. We use *target records* to denote the records in the dataset that are similar to the query record. In the hash function $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{w} \rfloor$ of LSH, w was set to 4 as an optimized value [20]. The distance threshold of R was set to 3 in all experiments. In SMLSH, m was set to 4 and n was set to 20.

We compared the performance of SMLSH with LSH in terms of accuracy, query time, percentage of dissimilar records, scope of retrieved similar records and effectiveness in locating similar records

- *Accuracy.* This is the rate between the number of located target records and the number of target records. High accuracy means that a similarity searching method can locate more similar records of query records.

- *Query time.* This is the time period from the time when queries are initiated to the time when similar records are returned. It shows the efficiency of a similarity searching method in terms of searching latency.

- *Percentage of dissimilar records.* This is the percentage of false positives in the located records. This metric shows the effectiveness of a similarity searching method in identifying similar records.

- *The scope of retrieved similar records.* This shows whether a similarity searching method can locate the similar record with different similarity as the query record.

- *Effectiveness.* This represents the rate between the number of located target records and the number of located records before the refinement phase. High effectiveness means that a similarity searching method can locate target records more accurately.

Figure 14 shows the query time of searching methods based on the linear method, kd-tree and LSH respectively. In the linear method, the query is compared with each record in the dataset in data search. As expected, the query time of the linear search method is the highest, and LSH leads to faster similar records location than kd-tree method.

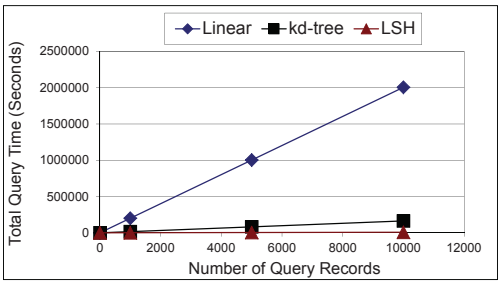

Figure 14: Query time of linear search, kd-tree and LSH.

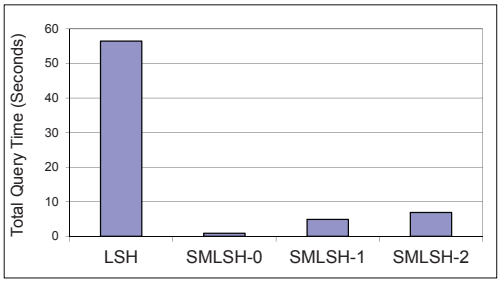

Figure 15: Total query time with refinement.

We also compared the performance of SMLSH with LSH in terms of query time, memory cost and effectiveness in locating the similar records. *Range* is the searching scope based on hash values. For a hash value h, the locations in [h-range, h+range] in the hash tables are checked, and records in these locations are candidates of similar records. We conducted experiments for the following methods:

(1) LSH;

(2) SMLSH with $range = 0$, denoted as SMLSH-0;

(3) SMLSH with $range = 8$, denoted as SMLSH-1;

(4) SMLSH with $range = 16$, denoted as SMLSH-2; Unless otherwise specified, all these methods don't have refinement phase, and the construction of new buckets during searching process is not used.

6.1 Query Time

We conducted an experiment for SMLSH with refinement of similarity calculations. We set the similarity threshold for SMLSH as 0.5. That is, SMLSH will return the records whose similarity to the query record are no less than 0.5. Figure 15 shows the total query time of different methods with the refinement phase. We can see that SMLSH has much faster query speed than LSH. This is due to two reasons. First, LSH needs to conduct more hash value calculations than SMLSH. In LSH, there are 2,346 groups of buckets, and 69 hash functions in each group. In SMLSH, there are 20 groups of buckets, and 4 hash functions in each group. Therefore, LSH needs to do much more hash value calculations than SMLSH. Second, LSH conducts Euclidean Space Distance computation, which includes multiple operations: addition, subtraction and square calculation to remove its false positive results.

The query time of SMLSH-2 is longer than SMLSH-1, and the query time of SMLSH-1 is longer than that of SMLSH-0. This is in expectation since larger range means more hash values needed to be checked, and more similarity calculations need to be conducted in the refinement phase.

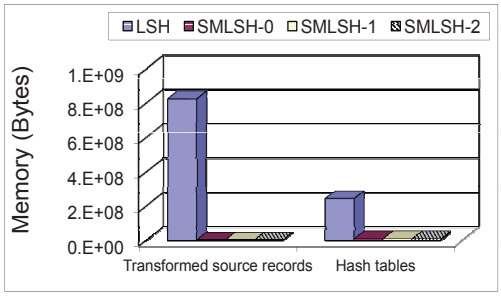

Figure 16: Memory cost for source records and hash tables.

6.2 Memory Cost

Recall that LSH transforms source records to vectors based on a global keyword list, and SMLSH uses SHA-1 to get record vectors. Both of them need memory space for record vectors and hash tables. Figure 16 shows the memory size for storing transformed source records and hash tables of LSH and differnt SMLSHs. It demonstrates that the memory consumption for both transformed source records

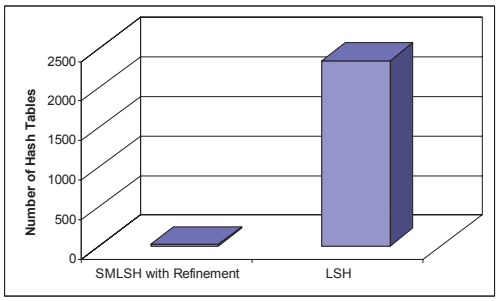

Figure 17: The number of hash tables.

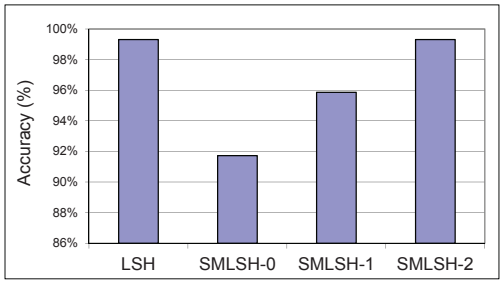

Figure 18: Accuracy.

and hash tables in SMLSH is much smaller than in LSH. This is due to the reason that SMLSH has much shorter record vectors and hence less storage memory. The vector dimension of LSH is 20,591, while the average dimension of SMLSH is 11. Therefore, SMLSH needs less memory for storing the transformed source records than LSH. There are 2,346 groups of buckets in LSH for each record, so there are 2,346 hashed values needed to be saved in the hash table for each record. For 10,000 source records, the hash table should save 23,460,000 hashed values totally. SMLSH only has 20 groups of buckets for each record, and the total number of hashed values contained in hash tables is 200,000. Consequently, LSH's hash table size is about 117 times more than SMLSH's hash table size. These results verify that SMLSH can significantly reduce the memory consumption of LSH.

Figure 17 presents the number of hash tables used in LSH and SMLSH with refinement. From the figure, we can notice that LSH needs one hundred times more hash tables than SMLSH for locating the similar records. As mentioned previously, the LSH algorithm generates 2,346 buckets for each record, while SMLSH only has 20 groups of buckets for each record. There is a hash table responds for each bucket. Therefore, 2,346 hash tables are needed for saving the records clustering results based on the buckets of the records in LSH, while 20 hash tables are needed in SMLSH.

6.3 Effectiveness

In addition to the efficiency in terms of memory consumption and query time, another important metric for searching methods is how many target records are missed in the returned record set. This metric represents the effectiveness of a searching method to locate target results.

Figure 18 shows the accuracy for each method. We observe that LSH and SMLSH-2 have higher accuracy than others, and they can find nearly all of the target records. However, the accuracy of SMLSH-0 and SMLSH-1 is lower than LSH and SMLSH-2. Since SMLSH-0 and SMLSH-1 have smaller range scope of the query record to check the similar records, they may miss some similar records that have less similarities to the query record. Therefore, with an appropriate value of range, SMLSH can achieve comparable effectiveness performance to LSH, but at a dramatically higher efficiency.

More hash tables provide more clustering results, which leads to high probability for locating more similar records of queries. Therefore, by combining the results of Figure 18 and Figure 17, we can observe that the number of hash tables can affect the accuracy of similarity searching. However, with the increase of the number of hash tables, more memory is required for storing the hash tables. When the space requirement for the hash tables exceeds the main memory size, looking up a hash bucket may require a disk I/O, which can cause delay in the query process. Therefore, an efficient similarity searching method can locate as many as similar records with low space requirement. From Figure 18 and Fig-

Table 1: Whether the original record can be found.

Similarity	SMLSH-0	SMLSH-1	SMLSH-2
1.0	Y	Y	Y
0.9	Y	Y	Y
0.8	Y	Y	Y
0.7	Y	Y	Y
0.6	N	N	Y
0.5	N	N	Y
0.4	N	N	Y
0.3	N	N	Y
0.2	N	N	Y
0.1	N	Y	Y

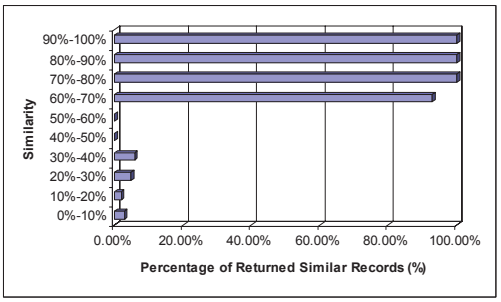

Figure 19: Percentage of similar records returned in different similarity.

ure 17, we can see that SMLSH can locate more than 90% of target records with small numbers of hash tables.

In order to see the similarity degree of located records to the query record of SMLSH, we conducted experiments on SMLSH-0, SMLSH-1 and SMLSH-2. We randomly chose one record, and changed one keyword to make a new record as query record every time. Our purpose is to see if SMLSH can find the original record with the decreasing degree of similarity to the query record. Table 1 shows whether the method can find the original record when it has different similarities to the query record. "Y" means the method can find the original record and "N" means it cannot. The figure illustrates that SMLSH-2 can locate the original record in all similarity levels, and SMLSH-0 and SMLSH-1 can return the records whose similarity are greater than 0.6 to the query record. The reason that SMLSH-2 can locate records with small similarity is because it has a larger scope of records to check. The results imply that in SMLSH, records having higher similarity to the query record have higher probability to be located than records having lower similarity.

Figure 19 depicts the percentage of similar records returned in different similarity in SMLSH. From the figure, we can observe that 100% of the similar records with similarity to the query records greater than 70% can be located in SMLSH. The percentage of returned similar records decreases as the similarity between source records and query records decreases. However, SMLSH still can locate more than 90% of similar records when the similarity of source records and query records are between 60% and 70%. Few similar records can be located with low similarity (less than 60%). Therefore, the source records with high similarity have higher probability to be found than the source records with low similarity.

7 Conclusions

Traditional information searching methods rely on linear searching or a tree structure. Both approaches search similar records to a query in the entire scope of a dataset, and compare a query with the records in the dataset in the searching process, leading to low efficiency. This entry first presents a Locality Sensitive Hashing (LSH) based similarity searching, which is more efficient than linear searching and tree structure based searching in a massive dataset. However, LSH still needs a large memory space for storing source record vectors and hash tables, and leads to long searching latency. In addition, it is not effective in a very high-dimensional dataset and is not adaptive to data insertion and deletion. This entry then presents an improved LSH based searching scheme (SMLSH) that can efficiently and successfully conduct similarity searching in a massive dataset. SMLSH integrates SHA-1 consistent hashing function and min-wise independent permutations into LSH. It avoids sequential comparison by clustering similar records and mapping a query to a group of records directly. Moreover, compared to LSH, it cuts down the space requirement for storing source record vectors

and hash tables, and accelerates the query process dramatically. Further, it is not affected by data insertion and deletion. Simulation results demonstrate the efficiency and effectiveness of SMLSH in similarity searching in comparison with LSH. SMLSH dramatically improves the efficiency over LSH in terms of memory consumption and searching time. In addition, it can successfully locate queried records. Our future work will be focused on further improving the accuracy of SMLSH.

Acknowledgements

This research was supported in part by U.S. NSF grants IIS-1354123, CNS-1254006, CNS-1249603, CNS-1049947, CNS-0917056 and CNS-1025652, Microsoft Research Faculty Fellowship 8300751. An early version of this work was presented in the Proceedings of ICDT'08 [15].

References

[1] C. Bohm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.* 2001, 33(3), 322–373.

[2] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences* 2002, 1(3), 630–659.

[3] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 2001, 5(17), 419–428.

[4] P. Ciaccia, M. Patella, and P. Zezula. M-trees: an efficient access method for similarity search in metric space. In *Proc. of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 25-29, 1997.

[5] A. Dasgupta, R. Kumar, and T. Sarlos. Fast locality-sensitive hashing. In *Proc. of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, San Diego, USA, August 21-24, 2011.

[6] C. Digout and M. A. Nascimento. High-dimensional similarity searches using a metric pseudo-grid. In *Proc. of the 21st International Conference on Data Engineering Workshops*, Tokyo, Japan, April 5-8, 2005.

[7] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the ACM SIGMOD International Conference*, Scottsdale, USA, May 20-24, 2012.

[8] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. *The VLDB Journal* 1999, 2(1), 518–529.

[9] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of the 30th Annual ACM Symposium on Theory of Computing*, Dallas, USA, May 24-26, 1998.

[10] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *Proc. 12th International Conference on Computer Vision*, Kyoto, Japan, September 27-October 4, 2009.

[11] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Integrating semantics-based access mechanisums with P2P file systems. In *Proc. of the the Third International Conference on Peer-to-Peer Computing (P2P)*, Linkping, Sweden, September 1-3,2003.

[12] R. R. Salakhutdinov and G. E. Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*, San Juan, Puerto Rico, March 21-24, 2007.

[13] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB* 2012, 5(5), 430-441.

[14] T. Sellis, N. Roussopoulos, and C. Faloutsos. Multidimensional access methods: Trees have grown everywhere. In *Proc. of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 25-29, 1997.

[15] H. Shen, T. Li, and T. Schweiger. An efficient similarity searching scheme in massive databases. In *Proc. of the Third International Conference on Digital Telecommunications*, Bucharest, Romania, June 29-July 5, 2008.

[16] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Process. Mag.* 2008, 1(2):128–131.

[17] H.-J. S. R. Weber and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of the 24th International Conference on Very Large Data Bases*, New York, USA, August 24-27, 1998.

[18] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Proc. of Neural Information Processing Systems*, Vancouver, Canada, December 8-13, 2008.

[19] C. Yu, B. C. Ooi, K. L. Tan, and H. V. Jagadish. Indexing the distance: an efficient method to knn processing. In *Proc. of the of 26th International Conference on Very Large Data Bases*, Seoul, Korea, September 12-15, 2001.

[20] A. Andoni and P. Indyk. E2LSH 0.1 User Manual, 2005. http://web.mit.edu/andoni/www/LSH/index.html [Accessed in May 2012].