Locality Sensitive Hashing Based Searching Scheme for a Massive Database

Haiying Shen, Ting Li, Ze Li, Felix Ching
Department of Computer Science and Computer Engineering
University of Arkansas, Fayetteville, AR 72701
{hshen, txl005, zxl008, ching}@uark.edu

Abstract

The rapid growth of information nowadays makes efficient information searching increasingly important for a massive database with tremendous volume of information. Traditional methods either rely on linear searching or depend on a tree structure. These methods search information in the entire database and compare a query with the records in the database during the searching process, which lead to inefficiency. This paper presents a locality sensitive hashing based searching scheme (LSS) to achieve highly efficient information searching in a massive database. LSS classifies information based on their similarities to facilitate fast information location. Based on the study and analysis of LSS, an improved scheme is further proposed to enhance the searching efficiency. Simulation results demonstrate the efficiency and effectiveness of the LSS schemes in searching information. They yield significant improvements over the efficiency of traditional methods. In addition, they guarantee successful location of the queried records.

1. Introduction

Our society is now defined as the "Information Society", in which the fast growth of information has brought about deep changes in our way of working and living. A massive database is a large database that contains high volume information such as mobile phone call records, surveillance videos, or music and images. For example, the National Security Agency has been collecting phone call records of tens of millions of Americans, using data provided by AT&T, Verizon and BellSouth. The spy agency is using the data to analyze calling patterns in an effort to detect terrorist activity [1].

Driven by the tremendous growth of information in a massive database, there is an increasingly need for an efficient information searching method that can locate desired information rapidly with low cost. Traditional methods depending on linear searching [2] compare a query record with each record in the database one at a time. However, such a method is not efficient in a massive database that has tremendous volume of information. Other methods [11-16] rely on a tree structure. Though it improves the efficiency of linear searching, the tree structure needs high overhead to maintain. Moreover, all these methods compare a query with source records during the searching process to locate the similar records, degrading the searching performance. Efficiently searching massive database for related data is critical for many applications. For instance, it helps FBI to mine massive database in search for terrorists in a timely fashion.

This paper presents a Locality Sensitive Hashing (LSH) based searching scheme (LSS) to achieve highly efficient information searching in a massive database. Based on LSH, it assigns identifiers to each record and clusters similar records based on the identifiers. Rather than comparing a query with records in a database, it facilitates direct and fast mapping between a query and a group of records. LSS further uses distance measurement to refine the located records. We investigate the operation of LSS, and further propose an improved LSH-based scheme to enhance its searching efficiency by Hamming distance measured in the refinement phase and reducing the length of identifiers.

The rest of this paper is structured as follows. Section 2 presents a concise review of representative methods for information searching. Section 3 and Section 4 describe and analyze LSS and the improved scheme in a massive database. Section 5 shows the performance of LSS in comparison with other methods. Section 6 concludes this paper with remarks on possible future work.

2. Related Work

In the past few years, there were numerous studies for the problem of finding the nearest neighbor of a query point in a high dimensional (at least three) space focusing mainly on the Euclidean space: given a database of n points in a d-dimensional space, find the nearest neighbor of a query point. This fundamental problem arises in several applications including data mining, information retrieval, and image search where distinctive features of the objects are represented as points in a d-dimensional space [3, 4, 5, 6, 7, 8, 9, 10].

Linear searching [2] compares a query record with each record in the database one at a time, which leads to inefficiency. Another approach uses the distance information to deduce k-dimensional points for records so that Vector Space Model multidimensional indexing

method [14] can be subsequently used such as FastMap algorithm [15] and Multidimensional Scaling [16].

Bentley et al. proposed k-dimension tree (kd-tree) data structure [11] that is essentially a hierarchical decomposition of space with long dimensions. Kd-tree is effectiveness in a low dimensional space, but its searching performance degrades as the number of dimensions becomes larger than two. Panigrahy [12] proposed an improved kd-tree search algorithm by simply perturbing the query point before traversing the tree, and repeating this for a few iterations [12]. BDD-trees (Balanced Box-Decomposition trees) [13] are extensions of kd-trees with additional auxiliary data structures for approximate nearest neighbor searching. It recursively subdivides space into a collection of cells and measures the distance between a cell and a query point to determine whether the points in the cell should be options in the kd-tree searching. These approaches map each record to a kd point and try to preserve the distances among the points. However, it is difficult to decide on a value of k and then map each domain object into a k-dimensional point while still accurately representing the similarity between objects.

Vantage point tree (vp-tree) [17] is a data structure that chooses vantage points to perform a spherical decomposition of the search space. This method is suited for non-Minkowski metrics and for lower dimensional objects embedded in a higher dimensional space [18]. The main drawback of vp-tree is that the region inside the median sphere and the region outside the median sphere are extremely asymmetric, and since volume grows rapidly as the radius of a sphere increases, the outside of the sphere tends to be very thin [19].

These tradition methods either depend on linear searching or rely on a structure for searching queried objects among all the objects in a database. LSS algorithm clusters objects based on their similarity, and maps a query directly to an object group in searching. Rather than in the entire database, the small searching scope within a group significantly improves the searching efficiency.

3. LSH-based Searching Scheme

LSH is an algorithm for solving the approximate and exact near neighbor search in high dimensional spaces [20, 21, 22]. For a domain S of a points set and distance measure D, the LSH family is defined as:

Definition 1. A family H = {h: S→U} is called (r_1, r_2, p_1, p_2) sensitive for D if for any points v, q ∈ S

- If $v \in B(q, r_1)$ then $Pr_H[h(q) = h(v)] \geq p_1$,
- If $v \notin B(q, r_2)$ then $Pr_H[h(q) = h(v)] \leq p_2$.

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

It provides a dimension reduction technique which projects objects in high-dimensional spaces to lower-dimensional spaces while still preserving the relative distances among objects. Different LSH families can be used for different distance functions. LSS relies on the LSH technique in Euclidean spaces proposed by Datar et al. [17] that uses p-stable distributions.

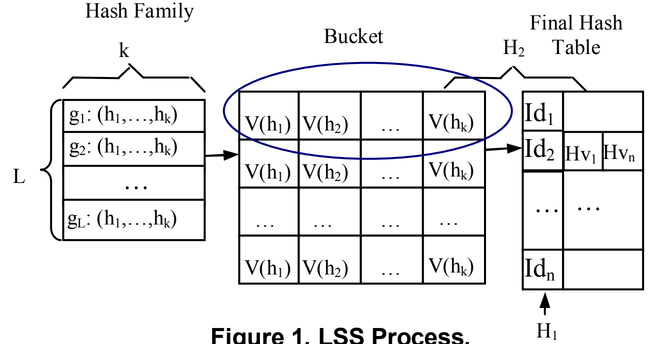

Figure 1. LSS Process.

Definition 2. A distribution D over R is called p-stable, if there exists p ≥ 0 such that for any n real numbers v_1, …, v_n and independent identically distributed (i.i.d.) variables X_1, …, X_N with distribution D, the random variable $\sum_i v_i X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p}X$, where X is a random variable with distribution D [5].

Stable distributions exist for any p that belongs to (0, 2] [22]. In particular two examples are:

- A Cauchy distribution D_C, defined by the density function $c(x) = \frac{1}{\pi}\frac{1}{1+x^2}$, is 1-stable (p = 1).
- A Gaussian (normal) distribution D_G, defined by the density function $g(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}$, is 2-stable (p = 2).

P-stable distributions can be used to reduce the dimension of high-dimensional points while preserving the relative distances among points.

Figure 1 shows the process of LSS. LSS stores the indices of similar records in the same row in the final hash table. Given a family H of hash functions, $h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor$, where *a* is a d-dimensional random vector whose each entry is drawn from a p-stable distribution (generated from *g(x)*); *b* is a random real number chosen uniformly from [0, *w*], and *w* is a specified value. LSS defines a function family G = {g: S→U^k} such that $g(v) = (h_1(v), …, h_k(v))$, where h_i belongs to H. The hashed value of each source record v, $h_i(v)$ ($1 \leq i \leq k$), is stored in bucket $g_j(v)$ ($1 \leq j \leq L$). Then LSS uses the hash function

$$h_1(a_1, a_2, ..., a_k) = \left(\left(\sum_{i=1}^{k} r_i' a_i\right) \bmod prime\right) \bmod tableSize$$

to compute the index of each bucket. This index indicates which row in the final hash table that the bucket should be stored. Rather than storing the original record in the final hash table, LSS uses the function

$$h_2(a_1, a_2, ..., a_k) = \left(\sum_{i=1}^{k} r_i'' a_i\right) \bmod prime$$

to get a single value of the record and stores the value in the hash table [20] in order to save memory.

To search near neighbors in Euclidean spaces, r is initially set to a specified value R. To process a query q, LSS also makes buckets $g_1(q)$, …, $g_L(q)$ first; then searches all hashed values of $g_1(q)$, …, $g_L(q)$ in the final hash table. Let v_1, … , v_t be the points which LSS found in the hash table for q, it then computes the Euclidean Space Distance between q and v_1, …, v_t using $d(x,y)=\|x-y\|=\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$. For each v_j, if v_j belongs to B(q, R), which means $d(x, y) \leq R$, then v_j is the similar point in range R. Otherwise, v_j is not the point satisfying to the query q. This refinement phase is to prune false positive results that are located as similar records but actually are not.

Let's take an example to explain how LSS works. Assume that the records in a database 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, LSS 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, the value is 0. The number of dimensions of a record is the total length of the keyword list. After transformation, the records will be:

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
…

Next, LSS produces the hash buckets $g_i(v)$ ($1 \leq i \leq L$) for every record. Thereafter, LSS computes the hash value for every bucket. Finally, the hashed value by h_2 of v is stored in the final hash table pointed by the hashed value by h_1. Figure 2 shows how the indices of records are stored in the final hash table based on LSS.

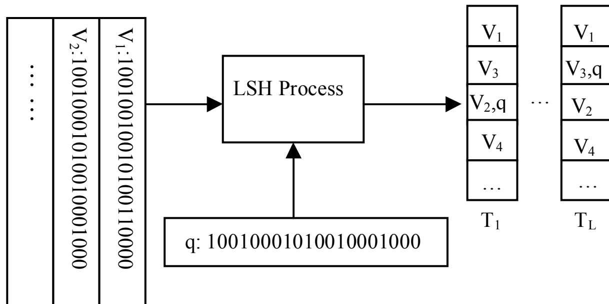

Figure 2. An example LSS.

If a query record 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 table through the same procedure. Consequently, the records that are in the same rows with q in hash table 1 to L are similar records. In the example, v_2 and v_3 are in the similar record set. Finally, the Euclidean Space Distance between a record and the query is computed, and the record will be removed from the returned record set if the distance is larger than R.

From this example, we can see that LSS does not need to search the query in the entire database scope. It shrinks the searching scope to a group of records similar to the query, and conducts refinement. Given n pieces of records in a database, traditional methods based on tree structures need O(log n) time for a query, and linear searching methods need O(n) time for query. LSS can locate the similar records in O(L) time, where L is a constant. It means that LSS is more efficient in a massive database that has rapidly increasing number of records. Unlike tree structure, LSS does not need to maintain a structure. Additionally, unlike the traditional methods, LSS does not need to compare the query with every source record.

4. Improved LSH-based Searching Scheme

A massive database has tremendous number of keywords, and a record may contains only a few keywords. As a result, the identifier of a record has 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 LSS returns many records that are not similar with the query even though all expected records are returned.

LSH has two important parameters: k and L. k is the number of projections per hash value. It represents a tradeoff between the time spent in computing hash values and time spent in pruning false positives. L is the number of hash tables. Given a k, an optimal value of L is found which ensures that the number of false positive is no more than a user specified threshold [20]. Larger k requires larger memory for hash tables and longer time for computing hash values, while smaller k leads to larger number of false positives before refinement phase, and hence longer time for pruning. This is confirmed by our simulation results in Figure 3 and Figure 4, we use *refinement rate* to denote the rate between the number of located records before the refinement phase and those after the refinement phase.

It is desirable to set a small value to k to reduce memory consumption and meanwhile to reduce time for pruning false positives. To improve the accuracy of returned results, and at the same time to reduce memory consumption and pruning time, we propose an improved LSH-based searching scheme (ILSS). Hamming distance between two equal length strings is the number of positions where the corresponding symbols are different [23]. For example, the hamming distance between 100101 and 101001 is 2. LSS

use its opposite to calculate the hamming distance. That is, bit comparison between two equal length strings is the number of positions where the corresponding symbols are the same. Instead of using Euclidean Space Distance calculation in the refinement phase, ILSS employs bit comparison calculation. Typically ILSS compares each bit between a source record vector and a query record vector and records the number of 1s at the same position (i.e. the number of the same keywords). For example, the bit comparison between string 100101 and string 101001 is 2.

We set a threshold for the number of similar keywords. During the comparison, when the number of similar keywords reaches the threshold, the comparison stops and the source record will be a true similar record. Algorithm 1 shows the pseudocode of ILSS in the refinement phase.

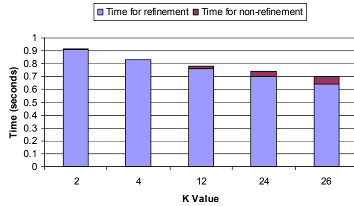

Figure 3. Searching latencies of different values of k.

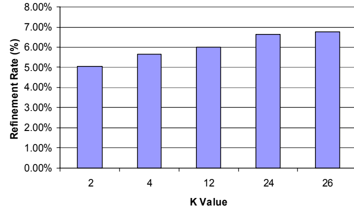

Figure 4. Refinement rates of different values of k.

Algorithm 1: Pseudo-code for bit comparison procedure in refinement phase in ILSS.

```
(1)  result ← 0
            \\count of the number of common keywords
(2)  for each i from 0 to dimension do
(3)      temp ← source [i] – query [i]    \\comparison
(4)      if temp = 0 and source [i] has meaning
(5)          then result ← result + 1
(6)          if result ≥ threshold
(7)              then return 1
(8)  endfor
(9)  return 0
```

Assume v_1, v_2 and v_3 are similar records found before refinement phase. ILSS then compares q to each similar record.

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

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

The first bits of v_1 and v_2 are the same as q's first bit. Therefore v_1 and v_2 are the true similar records of q if the threshold is 1.

We also observe that the memory required for LSH algorithm is mainly used to store the identifiers of records and the hash tables. Figure 5 shows the memory used for different objects in LSS. In LSS, the k is chosen to minimize the time in pruning false positives given a certain available memory space. Shorter length of record identifier will lead to less value of k.

As a result, record identifier compression can not only reduce the memory for storing the identifiers of resource records, but also helps to reduce k, hence the memory for hash tables.

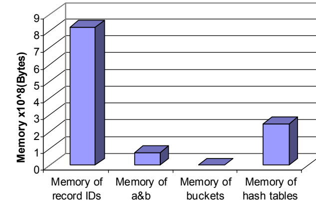

Figure 5. Memory usages for LSS.

We propose a method to compress records which can still preserve the locality of records. In the method, a record is divided into a certain number of groups, and each group is replaced by the position of the first 1. For example, as shown in the following, a record identifier is divided into four groups with four bits in each group:

Before compression: 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 1

After compression: 4 0 3 2

This method decreases the dimension of the record from 16 to 4: 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 1 => 4 0 3 2

16 dimension   4 dimension

5. Performance Evaluation

We conducted experiments on E^2LSH 0.1 of MIT [24]. That is a simulator for the high-dimensional near neighbor search based on LSH in the Euclidean space. Our testing dataset was the same style as the example dataset in section 3. The dimension of the dataset is 20,591. The number of source records was 10,000. 96 query records were selected from source records. We use target records to denote the records in the dataset that are similar to the query record. In the hash function $h_{a,b}(v)=\left\lfloor \frac{a\cdot v+b}{w} \right\rfloor$, w was set to 4 as an optimized value [25]. The distance threshold of R was set to 3 in all experiments.

Figure 6 show the query time of searching methods based on linear method, kd-tree and LSS respectively. As expected, the query time of the linear method is the highest, and LSS leads to faster similar records location than kd-tree method. We also conducted experiments for the following methods: (1) LSH with Euclidean Space Distance computation, denoted as LSS; (2) LSH with bit comparison and one keyword threshold, denoted as LSS-1; (3) LSH

with bit comparison and one word threshold on the compressed (9 times) dataset, denoted as ILSS-1; and (4) LSH with bit comparison on the original dataset with two word threshold, denoted as LSS-2. Table 1 shows the dimension of records in each method. We can see that with compression, the dimension is reduced by a factor of 5.

5.1 Query Time. Figure 7 shows the total query time. We can see that LSS-1 leads to faster query speed than LSS. This is because Euclidean Space Distance computation needs multiple operations to compute distance: add, minus and square. Bit comparison only needs to do minus, and once LSS finds one common word, it will consider this record as the query's true similar record. We can also find that ILSS-1 further significantly reduces the query time of others. It means that identifier compression is effective to accelerate the query.

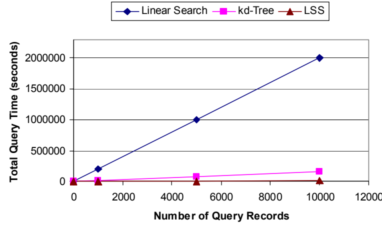

Figure 6. Total query time of linear searching, kd-tree and LSS.

Table 1. The dimension for the records.

Experiment	LSS	LSS-1	ILSS-1	LSS-2
Dimension	20,591	20,591	4,575	20,591

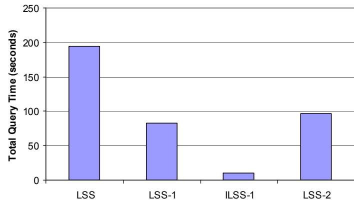

Figure 7. Total query time.

5.2 Memory Cost. Figure 8 shows the memory size for storing source records and hash tables in each experiment. It demonstrates that the memory consumption for both source records and hash tables with compression is much smaller than without compression. It is due to the reason that shorter record identifiers need less memory for storage. In addition, the memory for hash tables is determined by the value of k. Compression reduces k resulting in less memory for hash tables. Experiment results show that k equals to 3 in ILSS-1, and equals to 26 in others based on k determination method in LSH.

5.3 Effectiveness. In addition to the efficiency is in terms of memory consumption and query time. One important metric for searching method is that how many target records are missed in the returned record set. This metric represents the effectiveness of a searching method to locate target results. We define

$$\text{Accuracy} = \frac{\text{Total number of target records located}}{\text{Total number of target records}}$$

and tested the performance of searching methods in terms of accuracy. Figure 9 shows the accuracy for each method.

We can see that LSS, LSS-1 and LSS-2 have high accuracy. They can find nearly all of the target records. But ILSS-1's accuracy is lower than others. After compression, the record identifier is not as sensitive as before and some information may be lost, so the accuracy is not as high as that without compression.

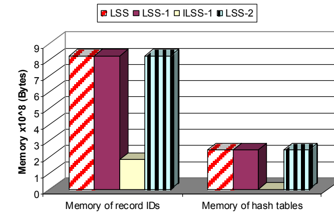

Figure 8. Memory for source record and hash tables.

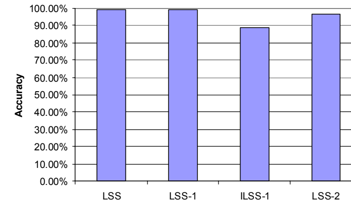

Figure 9. Accuracy.

An effective method should return fewer false positive records. Figure 10 shows the number of returned records. We can observe that at most of the time, LSS-1 can reduce the number of returned records of LSS, but LSS-2 can always reduce the number. Due to the record identifier sparsity, Euclidean Space Distance computation may return some records which do not have common word with the query. Hence, it generates a large number of false positives. Recall that in LSS-m, when a source record has one common word with the query, the source record is considered as a similar record of the query. Therefore, the bit comparison guarantees a certain similarity between the returned records and the query. Higher keyword threshold leads to higher similarity, hence less false positives. Therefore LSS-2 returns fewer records than LSS-1. It means that with appropriate threshold of compression, LSS with compression can greatly improve the effectiveness of LSS. We can also find that ILSS-1 returns the least records compare with LSS and LSS-2. It is shorter and condensed record ID helps it to eliminate the false positives in its returned record set.

6. Conclusions

Traditional information searching methods rely on linear searching or a tree structure. They search a query in the

entire scope of a database, and compare a query with the records in the database in the searching process, leading to low efficiency. This paper presents a locality sensitive hashing based searching scheme (LSS) that can efficiently and successfully search information in a massive database. LSS clusters similar records and maps a query to a group directly without sequential comparison. Based on the study and analysis of the LSS, an improved scheme is further proposed to enhance searching efficiency. Simulation results demonstrate the efficiency and effectiveness of LSH-based schemes in searching information. They dramatically improve the efficiency over the traditional methods. In addition, it guarantees successful location of queried records.

The future work will be focused on shortening the query time, reducing the memory usage and improving the accuracy of LSS.

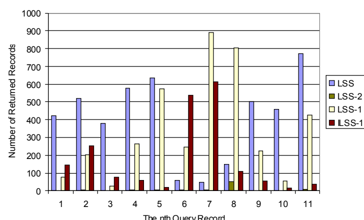

Figure 10. The number of returned results.

Acknowledgments

This research was supported by U.S. Acxiom Corporation.

References

[1] NSA has massive database of Americans' phone calls: http://www.usatoday.com/news/washington/2006-05-10-nsa_x.htm, May, 11, 2006.

[2] J. J. Hu, C. J. Tang, J. Peng, C. Li, C. A. Yuan, A. L. Chen, "A Clustering Algorithm Based Absorbing Nearest Neighbors", *6th International Conference of WAIM 2005*, Hangzhou, China, October 11-13, 2005.

[3] T. Cover, P. Hart, "Nearest Neighbor Pattern Classification", *IEEE Trans* of Information Theory IT-13, pp. 21-27,1967.

[4] S. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Fumas, R. A. Harshman, "Indexing by Latent Semantic Analysis", *Journal of the Society for Information Science*, 41(6), 391-407, 1990.

[5] L. Devroye, T. J. Wagnet, "Nearest Neighbor Methods in Discrimination", *Handbook of Statistics*, volume 2, P. R. Krishnaiah and L. N. Kanal, editors, North-Holland, 1982.

[6] R. Fagin, "Fuzzy Queries in Multimedia Database Systems", *Proc. ACM Symposium on Principles of Database Systems*, pp. 1-10, 1998.

[7] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, P. Yanker, "Query by Image and Video Content: The QBIC system", *Computer 28* 23-32, 1995.

[8] A. Pentland, R. W. Picard, S. Sclaroff, "Photobook: Tools for Content-based Manipulation of Image Databases", *Proc. the SPIE Conference On Storage and Retrieval of Video and Image Databases*, vol. 2185, pp. 34 – 47, February 1994.

[9] C. J. V. Rijsbergen, "Information Retrieval", *Butterworths*, London, United Kingdom, 1990.

[10] G. Salton, "Automatic Text Processing", *Reading*, MA: Addison-Wesley, 1989.

[11] J. L. Bentley, J. H. Friedman, R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time", *ACM Transactions on Mathematical Software*, 3(3):209-226, 1977.

[12] R. Panigrahy, "Nearest Neighbor Search using Kd-trees", December 4, 2006.

[13] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Wu, "An Optimal Algorithm for Approximate Nearest Neighbor Searching", *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pp. 573-582, 1994.

[14] D. A. White, R. Jain, "Algorithms and Strategies for Similarity Retrieval", *Technical Report* VCL-96-101, University of California, San Diego, July 1996.

[15] W. Niblack, R. Barber, W. Equitz, et al. "The QBIC Project: Querying Images by Content using Color, Texture and Shape", *Proc. SPIE: Storage and Retrieval for Image and Video Database*, volume 1908, pp. 173-187, Feb. 1993.

[16] J. B. Kruskal, M. Wish, "Multidimensional Scaling", *SAGE publication*, Beverly Hills, 1978.

[17] A.Fu, P. M. S. Chan, Y. L.Cheung, Y. S. Moon, "Dynamic VP-Tree Indexing for N-Nearest Neighbor Search Given Pair-Wise Distances", *VLDB Journal*, 9(2): 154-173, June 2000.

[18] P. N. Yianlios, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces", *Proc. the 4th annual ACM-SIAM Symposium on Discrete algorithms*, pp. 311-321, Austin, Texas, United States, 1993.

[19] S. Brin, "Near Neighbor Search in Large Metric Space", *Proc. the 21st international Conference on VLDB*, pp. 574-584, 1995.

[20] M. Datar, N. Immorlica, P. Indyk, V.Mirrokni, "Locality-sensitive Hashing Scheme based on p-stable Distributions", *DIMACS Workshop on Streaming Data Analysis and Mining*, 2003.

[21] P. Indyk, R.Motwani, "Approximate Nearest Neighbor: towards Removing the Curse of Dimensionality", *Proc. the Symposium on Theory of Computing*, 1998.

[22] V.M. Zolotarev, "One-Dimensional Stable Distributions", *Trans of Mathematical Monographs*, American Mathematical Society, Vol. 65, 1986.

[23] "Hamming Distance", website: http://en.wikipedia.org/wiki/Hamming_distance.

[24] "LSH Algorithm and Implementation (E2LSH)", website: http://web.mit.edu/andoni/www/LSH/index.html.

[25] A. Andoni, P. Indyk, "E^2LSH 0.1 User Manual" June 21, 2005.