An Adaptive Multi-level Hashing Structure for Fast Approximate Similarity Search

Alexander Ocsa, Elaine P. M. de Sousa

Universidade de São Paulo, Brazil
{aocsa, parros}@icmc.usp.br

Abstract. Fast information retrieval is an essential task in data management, mainly due to the increasing availability of data. To address this problem, database researchers have developed indexing techniques to logically organize elements from large datasets in order to answer queries efficiently. In this context, an approximate similarity search algorithm known as Locality Sensitive Hashing (LSH) was recently proposed to query high-dimensional datasets with efficient computational time. The query cost of LSH is sub-linear on the dataset size. However, some drawbacks of LSH have not been solved entirely, such as the need of several hash indexes to achieve accurate query results and the critical dependency on data distribution and parameter values. Therefore, the LSH solution is unsuitable to many real applications involving dynamic datasets. In this paper, we propose an adaptive Multi-level hashing to support dynamic index construction efficiently. By employing a Multi-level scheme it is possible to dynamically adapt the data domain parameters and exploit the resulting multi-resolution index structure to speed up the query process. Experimental studies on large real and synthetic datasets show the similarity search performance of the proposed technique.

Categories and Subject Descriptors: H. Information Systems [**H.m. Miscellaneous**]: Databases

Keywords: Access methods, LSH, Multidimentional Index, Similarity Information Retrieval

1. INTRODUCTION

In the past few decades, the information retrieval and database communities have been challenged by the increasing availability of complex data from different domains, such as multimedia (video sequences, images, CAD drawings), internet (XML), Biology (DNA sequences), sensor networks (data streams), and so on. Representing and searching complex data are non-trivial tasks. Usually, data representation is not precise and search processes are not based on equality criteria. For instance, it is highly improbable to obtain two identical descriptor patterns from two different images, even if they we obtained from the same person. Hence, the use of similarity criteria is particularly common in such complex domains, as similarity is an intuitive criterion for comparisons. In this scenario, many research works have put a great deal of effort into developing efficient indexing structures and similarity search algorithms, mainly following two different approaches: exact similarity search, such as Metric Access Methods (MAMs) and Spatial Access Methods (SAMs), and approximate similarity search, such as techniques based on data projection into sub-spaces.

Many multidimensional indexing techniques have been applied to information retrieval tasks. Database applications employ Access Methods such as MAMs and SAMs due to their ability to build data structures for managing and organizing large datasets efficiently. In particular, several empirical results, such as Slim-Tree [Traina et al. 2002] and DBM-Tree [Vieira et al. 2004], support that MAMs are more suitable than SAMs to handle high-dimensional data.

The performance of MAMs depends mainly on the Number of Distance Computations performed

This work has been partially supported by CNPq, CAPES, FAPESP, STIC-AmSud, and Microsoft Research.

during the index construction and the search processing. Generally, in order to answer similarity queries a MAM needs to hierarchically explore the index data and try and prune regions which do not overlap the query region by virtue of the triangle inequality (see details in Section 2). The remaining data (i.e. relevant candidate set) is exhaustively analyzed to report only objects that satisfy the query condition.

Although several MAMs have been proposed to speed up similarity queries, most of them are either affected by the well-known "curse of dimensionality" or suffer from overlapping among regions. Some studies have shown that the idea of data representation with hyper spherical or rectangular region hierarchies can deteriorate similarity queries even compared with sequential search [Böhm et al. 2001].

Different approaches have been studied to solve the "curse of dimensionality". One of the research lines is to try to avoid the dimensionality problem by relaxing the query precision to speed up the query time. Potentially, this approach is feasible for applications that do not require exact answers and for which speed is more important than search accuracy. Moreover, the metric space definition already leads to an approximation of the true answer, and thus a second approximation at search time may be acceptable [Chávez et al. 2001]. In this direction, Locality Sensitive Hashing (LSH) [Datar et al. 2004] is one of the recent hash-based techniques proposed to organize and query high-dimensional data. Indeed, LSH is one of the few techniques that provide solid theoretical analysis and predictable loss of accuracy in the results. To answer similarity queries, LSH searches only regions, which are represented by buckets, to which the query object is hashed (i.e., the candidate buckets containing the dataset objects with a high probability of similarity to the query object). Therefore, there is no need to fully explore the index data, and only the objects into the candidate buckets require further processing. However, some drawbacks of LSH have not been solved entirely, as follows:

- LSH requires several indexes such that each index organizes the whole dataset using a hash table with independent hash functions. This requirement is remarkably critical to improve search accuracy, but leads to high memory consumption. In a recent work, a variant of LSH called Multi-probe LSH [Lv et al. 2007] was proposed to keep the memory cost acceptable. The Multi-probe LSH is build on the basic LSH, but while the LSH query algorithm examines only one bucket for each hash table, Multi-probe LSH effectively probes multiple buckets that are likely to contain query results in each hash table. Consequently, the number of hash tables is reduced with no significant loss of accuracy.
- A recent research on LSH [Dong et al. 2008] approaches shows that their performance on point queries depends not only on the overall distribution of the dataset, but also on the local geometry in the vicinity of the particular query point. In the original Multi-probe LSH, a fixed number of probes may be insufficient for some queries and larger than necessary for others.
- LSH has a critical dependency on parameter values and data domain, which determine the number of hash functions and number of hash tables. Moreover, its parameters also depend on the number of objects to be indexed; as a consequence, LSH is not an incremental solution. To deal with this issue, a scheme called LSH Forest [Bawa et al. 2005] was developed to support a self-tuning behavior with respect to number of hash functions. Essentially, LSH Forest is a collection of prefix trees where each one can handle a different number of hash functions. However, It does not ensure the same search accuracy and performance as LSH.
- The basic LSH parameters are tuned considering a static dataset. However, in many real applications data distribution changes dynamically and LSH cannot handle this dynamic workload efficiently.

This paper proposes a novel hashing technique, named Adaptive Multi-Level LSH, designed to perform approximate similarity search on dynamic datasets. Our technique is based on the idea of LSH of hashing similar objects to the same bucket, but unlike LSH it does not expect the data domain parameters. Hence, we do not need the complete dataset to define the search space. Instead, we dynamically adapt the data domain parameters during the indexing process thanks to the self-adaptive abilities of a multi-resolution index structure (see details in Section 3). Additionally, the

multi-resolution approach can speed up the query process.

Our contributions are as follows. First, we introduce and define a multi-resolution index structure designed to handle a dynamic set of objects. Second, we exploit the multi-resolution approach to speed up similarity queries. Finally, we present our Adaptive Multi-Level LSH algorithms and report experiments on real and synthetic datasets, comparing our approach to recent MAMs and current LSH extensions. To the best of our knowledge, this is the first work that reports a complete performance study comparing state-of-the-art MAMs and LSH indexing techniques.

The paper is organized as follows. Section 2 summarizes the background for this work. Section 3 describes the proposed technique and Section 4 reports experimental results on real and synthetic datasets. Finally, we conclude in Section 5.

2. BACKGROUND

This section presents background concepts and related work. Table I lists the main symbols used hereinafter. We first define a universe of objects $\mathbb{U}$ and a function $d : \mathbb{U} \times \mathbb{U} \rightarrow \mathbf{R}$ that measures

Table I. Summary of symbols.

Symbol	Definition	Symbol	Definition
$\mathbb{U}$	domain of objects	$h()$	locality sensitive hash function
S	set of objects (dataset) in domain $\mathbb{U}$	$H = \{h_1(), ..., h_m()\}$	a family of m hash functions
N	number of objects in a dataset	m	number of hash functions
D	intrinsic dimensionality of a dataset	C	bucket capacity
q	a query object (or query center)	T	hash table
r	radius of range query	L	number of hash tables
k	number of neighbors in a k-NN query	w	quantization bucket width
$d()$	distance function	I	bucket in a hash table
c	approximation ratio	t	level in a multi-level data structure

the distance between two objects of $\mathbb{U}$. On metric spaces, $S \subseteq \mathbb{U}$ is a finite set of data that can be preprocessed, where the function $d()$ measures the dissimilarity between objects and satisfies the following proprieties, $\forall x, y, z \in \mathbb{U}$:

1. $d(x, y) \geq 0$ positiveness
2. $d(x, y) = d(y, x)$ symmetry
3. $d(x, y) = 0 \leftrightarrow x = y$ reflexivity
4. $d(x, y) \leq d(x, z) + d(y, z)$ triangle inequality

Triangle inequality is an important property applied to speed up similarity search algorithms. It gives bounding values on a distance we may not have computed, i.e., if we have the distances $d(x, z)$ and $d(y, z)$, the bounding values for the unknown distance $d(x, y)$ are $|d(x, z) - d(y, z)| \leq d(x, y) \leq d(x, z) + d(y, z)$.

Given a query object $q \in \mathbb{U}$, in order to recover similar objects to q, the following basic types of queries are defined:

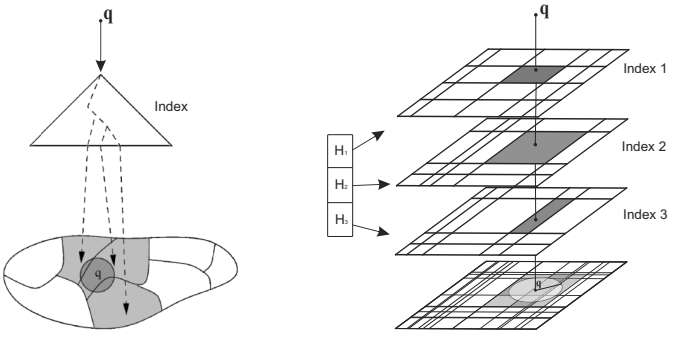
- **Range query** $Rq(q, r)$: finds all elements within the query radius r, i.e., $Rq(q, r) = \{u \in \mathbb{U} | d(u, q) \leq r\}$.
- **k-Nearest Neighbor query** $kNN(q, k)$**:** finds the k nearest neighbors of q, i.e., Finds a set $A \subseteq \mathbb{U}$ so that $|A| = k$ and $\forall u \in A, v \in U - A, d(q, u) \leq d(q, v)$.
- **Nearest Neighbor query** $NN(q)$**:** finds the nearest neighbor of q, i.e., $NN(q) = kNN(q, 1)$.

In order to improve the performance of similarity queries, different indexing approaches have been proposed. Metric Access Methods (MAMs) work on metric spaces, organizing data according to a similarity criterion. Many MAMs, such as tree-based techniques VP-Tree [Yianilos 1993], SAT [Navarro 2002], M-Tree [Ciaccia et al. 1997] and their extensions Slim-Tree [Traina et al. 2002], DBM-Tree [Vieira et al. 2004] and DF-Tree [Traina et al. 2002] are found in the literature. Good surveys on MAMs can be found in [Chávez et al. 2001; Hjaltason and Samet 2003].

Spatial Access Methods (SAM), such as Kd-Tree [Bentley 1979], R-tree [Guttman 1984] or methods based on them such as R*-Tree [Beckmann et al. 1990], R+-Tree [Sellis et al. 1987], X-Tree [Berchtold et al. 1996], describe the input data as dimensional vectors $(x_1, x_2, ..., x_n)$. They can index points and geometrical objects. Good survey on SAMs can be found in [Gaede and Günther 1998].

Recently, a promising indexing technique named Locality Sensitive Hashing (LSH) [Gionis et al. 1999; Datar et al. 2004] was proposed to solve the approximate similarity search in high dimensional data efficiently. LSH is based on the idea that closeness between two objects is usually preserved by a projection operation. In other words, if two objects are close together in their original space, then these two objects will remain close after a scalar projection operation.

Figure 1 illustrates similarity search models for both MAM and LSH approaches. Figure 1 (a) shows the unified model for searching on metric spaces presented in [Chávez et al. 2001]. A *metric index* organizes the dataset into regions defined by the distance function, such that each region includes objects which are sufficiently close to each other. At the query time, the triangle inequality is applied to discard regions which do not overlap the query region (sphere in the figure). Hence, only the objects in the qualifying regions (gray regions in the figure) are further processed in order to test the query condition. A similar search model for LSH is shown in Figure 1 (b), where each index is defined by a set of hash functions (H_1, H_2, H_3) which generate three different partitioning of the search space. Hence, each partitioning is associated to a hash table and its corresponding set of hash functions. Each set of hash functions is employed to organize the dataset into regions of the search space such that objects in the same region are considered close together under certain probability. At the time query, the query object is hashed (using each of the three sets of hash functions - one for each partitioning) to regions with high probability of finding similar objects (all regions in the figure). Finally, the qualifying regions are analyzed in order to report only the objects which satisfy the query condition.



(a) search in candidate regions (b) search in subspaces

Fig. 1. Unified model for searching. (a) Metric Access Methods . (b) LSH.

2.1 Locality Sensitive Hashing

Some previous work [Gionis et al. 1999; Datar et al. 2004; Andoni and Indyk 2008] have recently explored the idea of hashing objects and grouping them into buckets with the goal of performing

approximate similarity search within buckets associated to the query element. In fact, LSH was designed to efficiently solve a c-approximate range query, i.e., find the objects whose distance to the query object q is at most $c \times r$, where r is the query radius.

The idea behind LSH is that if two objects are close together in their original space, then these two objects will remain close after a scalar projection operation. So, let $h(x)$ be a hash function that maps a d-dimensional point x to a one-dimensional value. The function $h(x)$ is said to be *locality sensitive* if the probability of mapping two d-dimensional points x_1, x_2 to the same value grows as their distance $d(x_1, x_2)$ decreases. Formally:

Definition 1. Given a distance value r, an approximation ratio c, the probability values P_1 and P_2 such that $P_1 > P_2$, a hash function $h()$ is *locality sensitive* if it satisfies the following conditions:

1. If $d(x_1, x_2) \leq r$ then $Pr[h(x_1) = h(x_2)] \geq P_1$, i.e., if two points x_1 and x_2 in R^d are close to each other, there is a high probability P_1 that they fall into the same bucket;
2. If $d(x_1, x_2) > c \times r$ then $Pr[h(x_1) = h(x_2)] \leq P_2$, i.e., if two points x_1 and x_2 in R^d are far apart, there is a low probability $P2 < P_1$ that they fall into the same bucket.

The LSH scheme proposed in [Datar et al. 2004] uses p-stable distributions as follows: given a d-dimensional vector $\vec{a}$ with entries independently chosen from a p-stable distribution (Cauchy or Gaussian), a real number b uniformly chosen from the range $[0, w]$, the hash value of a d-dimensional point $\vec{x}$, for the ℓ_2 norm, is defined as:

$$h(x) = \lfloor \frac{\vec{a} \cdot \vec{x} + b}{w} \rfloor \tag{1}$$

Equation 1 has a simple geometric interpretation. Consider Figure 2: lets $\vec{p_1}$ and $\vec{p_2}$ be two vectors in $\mathbf{R}^2$, $\vec{a}$ a unit norm vector (without loss of generality) and b a random real number. The slope of the line crossing the origin coincides with the direction of $\vec{a}$. We project $\vec{p_1}$ onto line $\vec{a}$ by performing the dot product operation $\vec{a} \cdot \vec{p_1}$. This projection is then finally quantized into intervals of fixed width w, defining point A. The same procedure is repeated for $\vec{p_2}$, determining point B.

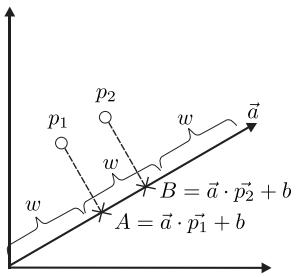


Fig. 2. LSH geometric interpretation. Hashing $\vec{p_1}$ and $\vec{p_2}$.

It is possible to magnify the difference between P_1 and P_2 by using a set of m different hash functions $H = \{h_1, h_2, ..., h_m\}$, each one defined according to Equation 1. It increases the ratio of the probabilities since $(P_1/P_2)^m > (P_1/P_2)$, ensuring that if two points are far away from each other, they are unlikely to fall in the same bucket. However, although preventing distant points from falling in the same bucket is crucial to preserve the spatial proximity, it is not enough. It is equally important to ensure that close points will appear in the same bucket with high probability. Unfortunately, the latter cannot be accomplished by using a single hash structure. Thus, LSH overcomes this problem cleverly

by considering L independent projections (i.e., building L independent hash structures) [Slaney and Casey 2008; Tao et al. 2009].

The process of indexing a complete dataset S using L hash tables with their respective sets of hash functions H is summarized in the Algorithm 1. Note that each element x of the dataset S is a d-dimensional point and, for each hash table T_i, x is projected according to each hash function in H_i. The result of these m projections is a m-dimensional vector x', where each entry corresponds to a hash value. The quantization process $quantize(x')$ is then computed using Equation 1. Finally, the appropriate bucket I is located and a new entry e is created.

Algorithm 1 Construction algorithm for LSH

Input: The dataset $S = \{x_1, ..., x_N\}$ to be indexed
Output: All objects are hashed in the L hash tables

```
1: for i = 1 to L do
2:    H_i ⇐ {h_1, ..., h_m} // Initialize hash table T_i with a set of hash functions H_i
3: end for
4: for i = 1 to L do
5:    for each x in S do
6:       x' ⇐ [h_1(x), ..., h_m(x)] // Project x: m projections using the set of hash functions H_i
7:       g ⇐ quantize(x') // Quantization process - compute the final hash value
8:       I ⇐ T_i[ g mod |T_i| ] // Locate the bucket I
9:       add entry e=<x, g> on bucket I // Store a reference to x and hash value g
10:   end for
11: end for
```

By the LSH construction algorithm, each bucket is created considering intra-similarity. Thus, similarity queries are performed by hashing the query object q into a number of buckets (qualifying buckets), also considering the L hash tables and their corresponding sets of hash functions H. As a consequence, the object q is mapped to buckets where similar objects are likely to be placed, limiting the number of required distance computations. In a nearest neighbor query, only the distances between q and the elements in the qualifying buckets are computed in order to find the nearest element. Similarly, in a range query the elements within a query radius r are output as the answer of the query. The range query algorithm for LSH is described in the Algorithm 2.

Algorithm 2 Range query algorithm for LSH

Input: The query object q and the query radius r
Output: Objects that satisfy the query condition.

```
1: for i = 1 to L do
2:    q' ⇐ [h_1(q), ..., h_m(q)] // Project q: m projections using the set of hash functions H_i
3:    g ⇐ quantize(q') // Quantization process - compute the final hash value
4:    I ⇐ T_i[ g mod |T_i| ] // Locate the bucket I
5:    for each entry e in I do
6:       // Query condition. Note: e =< x, g >
7:       if e.g = g1 ∧ d(q, e.x) ≤ r then
8:          report e.x if it has not been previously reported
9:       end if
10:   end for
11: end for
```

Note that the kNN query algorithm can be obtained by modifying the query condition in Algorithm 2. Thus, the kNN query algorithm will stop once it has found k distinct objects or if there is not

more buckets to explore. However, it is important to recall that LSH ensures quality results only for c-approximate range queries. Thus, the quality results in kNN queries can be improved if one of the next strategies are considered: (1) Using multiples LSH schemes configured with different radii in order to cover the most distances between q and its k nearest neighbors. However, this approximation requires expensive space and query cost. (2) Using the Multi-probe approach where multiples buckets are analyzed by each hash table in order to increment the candidate elements. However, a fixed number of probes used in the original Multi-probe LSH may be insufficient for some queries and larger than necessary for others. As we can see in the next section we use the last approach to continue ensuring quality results for kNN queries.

In summary, the original LSH algorithm tessellates the search space by employing m hash functions randomly chosen from a Gaussian (or a Cauchy) distribution. The number of hash functions (m) determines how sparse or dense the search space will be. By increasing the value of m, objects tend to be uniformly distributed into buckets, reducing query accuracy as it is more likely that similar objects fall into different buckets. That is the reason why many hash tables are required to ensure quality results as they mitigate this bad effect. On the other hand, by decreasing the value of m, the consequent large number of collisions degrades the performance of the queries, as the amount of qualifying buckets and candidate elements to be processed increase.

3. ADAPTIVE MULTI-LEVEL LSH

In this section, we propose the *Adaptive Multi-level LSH* to solve approximate similarity search on a dynamic set of objects. As introduced in Section 1, our strategy is to design an adaptive data structure by considering a multi-resolution index structure. The Adaptive Multi-level LSH, like LSH, is based on the idea of projecting similar objects into the same bucket. But unlike LSH, it does not expect the data domain parameters. Hence, we do not need the full dataset to define the search space. Instead, we dynamically adapt the data domain parameters during the indexing process by using a multi-resolution approach, i.e., hash tables with dynamic capacity of hash functions are created during the index construction.

As the performance of a hash table degrades if there are too many objects in one bucket, the number of buckets may need to grow dynamically. Two important methods to support graceful growth are extensible and linear hashing. Both start by hashing search-key values to long bit-strings and use a varying number of those bits to determine the bucket [Ullman et al. 2001]. In our work, we apply the linear hashing approach because it allows expansion of the hash table one bucket at a time.

Additionally, the multi-resolution index structure supports the use of the relations among different resolution levels to extend the candidate set of objects during the query process. As a consequence, fewer LSH hash tables are needed and the query process can be improved.

3.1 The Multi-level structure

Recall that the original LSH [Datar et al. 2004] consists of L hash tables with a fixed number of hash functions per hash table. Each hash table indexes the whole dataset and each bucket can contain an arbitrary number of objects. Our technique changes a one-level hash table into a Multi-level data structure. In each level t the data is organized using distinct numbers of hash functions. The first level of the structure uses $m_1 = 4$ hash functions. The next levels use $m_t = m_{t-1} + 2, t > 1$ hash functions. The initial value $m = 4$ is the minimum number of hash functions needed to distribute data uniformly into buckets for a standard dataset. Each bucket I can store C objects and a reference list, which keeps references to buckets located in the next level and with a vicinity relation with bucket I. Consecutive levels are thus connected.

Figure 3 illustrates the basic concept of the multi-level hashing by showing a 3-level structure, each level with resolution $4, 6, 8$ respectively. The bucket capacity C for this example is 2.

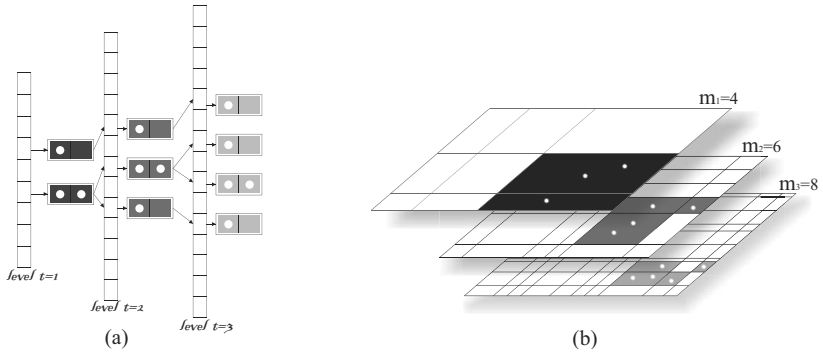


Fig. 3. Multi-level LSH structure. (a) internal view. (b) logical view.

3.2 Construction of Multi-level LSH

While the basic LSH scheme indexes the whole dataset using L hash tables with independent hash functions considering a fixed resolution m (as described in Algorithm 1), the adaptive Multi-level scheme can index data by incrementally adapting the number of hash functions. Thus, data is organized into a multi-resolution structure.

The Multi-level LSH is constructed by executing a recursive insertion procedure for each object of the dataset, for each hash table $T_i, 1 \leq i \leq L$. Thus, for a hash table T_i, if an object is supposed to be inserted into a bucket of level t that is already full, it is hashed to the next level $t+1$, and so on. The relation between the bucket in the level t and its vicinity in the following level $t+1$ is then established. The insertion algorithm for the Adaptive Multi-level LSH is described in Algorithm 3. Notice that the input parameters are the object to be inserted and number of hash functions m, which is initially set to $m = 4$ in order to initialize the hash table T_i.

Algorithm 3 Insert(level t, Object x, Integer m) Note: for each hash table T_i, call $insert(T_i, x, 4)$

Input: The object to insert x and the current level value m (# of hash functions)
Output: Insert the object x into T_i

1: **if** H_t is NULL **then**
2: $H_t \Leftarrow \{h_1, ..., h_m\}$ //Initialize the current level t with a set of hash functions
3: **end if**
4: $x' \Leftarrow [h_1(x), ..., h_m(x)]$ // Project x: m projections using the set of hash functions H_t
5: $g \Leftarrow quantize(x')$ // Quantization process - compute the final hash value
6: $I \Leftarrow T_t[\ g \bmod |T_t|\ ]$ // Locate the bucket I
7: **if** $|I| <$ C **then**
8: add entry <x, g> on bucket I // Store a reference to x and value g if I is not full
9: **else**
10: Insert(nextLevel(t), x, $m+2$) // Otherwise, insert x recursively in the next level
11: Save the connection between the current level and the next level on the reference list of I
12: **end if**

An important issue to be considered is the nonexistence of sufficient buckets in the index to partition the dataset effectively. To handle this problem, we follow the linear hashing approach [Litwin 1988] to grow the number of buckets by 1 whenever the average bucket occupation reaches a threshold. We empirically set the default threshold to 75%. Finally, as the population of a single bucket cannot cause the table to expand, overflow buckets are needed in some situations.

3.3 Similarity search in Adaptive Multi-level LSH

To solve similarity queries in the Multi-level LSH scheme, the query object q is hashed to locate the appropriate bucket I in each level t by using m_t hash functions. Once the bucket I is located, the relevant candidate set are formed by I, the neighbors in the next level stored in the reference list of I. Then, the elements in the candidate set are exhaustively analyzed in order to recover only the objects that satisfy the query condition. This process is performed for each of the L hash tables.

The range query algorithm for the Adaptive Multi-level LSH is described in Algorithm 4. For some queries (say kNN queries), the number of buckets to be explored are insufficient to ensure the quality results. Thus, to increment the candidate elements the basic LSH can use more hash tables. However, there is a much clever idea, the Multi-probe LSH [Lv et al. 2007] employs multiple probes to extend the candidate set at query time. The problem is determine the number of probes required by a specific query. As we can see in the next section the number of required probes depends on the query region density.

Algorithm 4 Range query algorithm for the Adaptive Multi-level LSH

Input: The query object q and the query radius r
Output: Objects that satisfy the query conditions.

```
1: for i = 1 to L do
2:   for each level t in the hash table T_i do
3:     q' ⇐ [h_1(q), ..., h_m(q)] // Project x: m projections using the set of hash functions H_(i,t)
4:     g ⇐ quantize(q') // Quantization process - compute the final hash value
5:     I ⇐ T_(i,t)[ g mod |T_(i,t)| ] // Locate the bucket I
6:     for each entry e in I, the vicinity of I (probes), the reference list of I do
7:       // Query condition. Note: e =< x, g >
8:       if e.g = g1 ∧ d(q, e.x) ≤ r then
9:         report e.x if it has not been previously reported
10:      end if
11:    end for
12:  end for
13: end for
```

3.4 Selection of parameters

As discussed previously, there is a tradeoff between time and space on LSH techniques, which is related with parameters configuration. But in our proposal, these parameters are not so crucial to answer similarity queries on dynamic datasets.

The increment parameter, which defines the number m of hash functions in the next level, was empirically determined by observing that when it is set with a high value, the elements though the multi-level structure tend to be uniformly distributed into the buckets. As a consequence, most connection tend to have a one-to-one relationship. Under this experimental observation, we found that the increment value 2 as the best value for which the Multi-level structure reaches a acceptable balance between space and time efficiency.

Another parameter is the bucket capacity C. If it is set with a high value, the number of levels will be small; otherwise, the number of levels will be large. To simplify our approach, we only consider the bucket capacity as a tuning parameter.

Thus, both the bucket capacity and the increment parameter will determine the number of connections among buckets at different levels. We observed that these connections also determine density regions. Hence, as the number of connections among buckets at different levels is a good indicator to

determine density regions, the number of probes to respond a specific query is defined by using the following rule: The first level uses $T_1 = all\_connections$ probes, where $all\_connections$ is the sum of all connections of implicated buckets when the query object q is projected throughout all levels. The next levels use $T_t = T_{t-1}/2; t > 1$ probes. In other words, the number of probes is set with a high value if the query region is dense.

4. EXPERIMENTS

We performed a comprehensive performance evaluation of our algorithm in terms of the *query performance (number of distance computations and response time)*, *accuracy*, *scalability* and *memory usage*. The performance of Multi-level LSH was compared to those of the two most well-known LSH methods, namely LSH [Datar et al. 2004] and Multi-probe LSH [Lv et al. 2007], and tree structures Slim-Tree [Traina et al. 2002] DF-Tree [Traina et al. 2002] and DBM-Tree [Vieira et al. 2004]. Furthermore, for all the LSH-based methods we compared their query results with linear-scan search to measure the search accuracy. The Euclidean distance (L_2) was used as the distance function in our experiments.

The Multi-level and all the LSH indexing methods were implemented in C++ into the Arboretum library [1], all with the same code optimization, to obtain a fair comparison. All of the experiments were performed on a 3.4Ghz Pentium 4 with 2Gb RAM.

4.1 Dataset description

We used synthetic and real datasets in our experiments.

(1) **synt16** This synthetic dataset contains 10,000 16-dimensional vectors normally distributed into 10 clusters in a 16-d unit hypercube.
(2) **synt32** Similar to Synthetic16D, but this contains 100,000 32-dimensional vectors.
(3) **synt64** Similar to Synthetic16D, but this is a 64-d unit hypercube.
(4) **synt256** Similar to Synthetic16D, but this is a 256-d unit hypercube.
(5) **color** This real dataset contains 68,000 32-dimensional vectors. Each vector describes the color histogram of an image in the Corel collection [2].
(6) **mnist** This real dataset contains 60,000 50-dimensional vectors. The MNIST [3] dataset of handwritten digits is a subset of a larger set available from NIST (National Institute of Standards and Technology) database. The dimensionality is reduced by taking the 50 dimensions with the largest variances.
(7) **audio** This real dataset contains 54,387 192-dimensional vectors. The audio dataset comes from the LDC SWITCHBOARD-1 [4] collection. It is a collection of about 2,400 two-sided telephone conversations among 543 speakers from all areas of the United States.

The process to generate the synthetic datasets is described in [Ciaccia et al. 1997]. These synthetic datasets were widely used to test metric access methods due to their simplicity to create complex sceneries. The color, mnist, and audio are real datasets and they were mainly used to test LSH in [Datar et al. 2004] and [Lv et al. 2007]. In our experiments, a test set was created for each dataset using 500 objects randomly chosen from the dataset. Half of them (250) were removed from the dataset before creating the indexes. This configuration allows us to evaluate queries with centers into the index or not.

[1] http://www.gbdi.icmc.usp.br/arboretum/
[2] http://kdd.ics.uci.edu/databases/CorelFeatures/
[3] http://yann.lecun.com/exdb/mnist/
[4] http://www.ldc.upenn.edu/Catalog/docs/switchboard/

LSH parameters		
Dataset	Method	Parameters
synt16 (16-D)	LSH	$L = 10$, $m = 8$
	Multi-probe LSH	$L = 3$, $m = 8$, $T = 20$
	Multi-level LSH	$L = 3$, $C = 64$
synt32 (32-D)	LSH	$L = 135$, $m = 24$
	Multi-probe LSH	$L = 14$, $m = 10$, $T = 30$
	Multi-level LSH	$L = 17$, $C = 64$
synt64 (64-D)	LSH	$L = 54$, $m = 10$
	Multi-probe LSH	$L = 8$, $m = 10$, $T = 30$
	Multi-level LSH	$L = 8$, $C = 64$
synt256 (256-D)	LSH	$L = 231$, $m = 16$
	Multi-probe LSH	$L = 40$, $m = 16$, $T = 40$
	Multi-level LSH	$L = 40$, $C = 256$
color (32-D)	LSH	$L = 153$, $m = 14$
	Multi-probe LSH	$L = 35$, $m = 14$, $T = 20$
	Multi-level LSH	$L = 35$, $C = 128$
mnist (50-D)	LSH	$L = 231$, $m = 16$
	Multi-probe LSH	$L = 37$, $m = 16$, $T = 30$
	Multi-level LSH	$L = 37$, $C = 128$
audio (192-D)	LSH	$L = 62$, $m = 20$
	Multi-probe LSH	$L = 10$, $m = 20$, $T = 40$
	Multi-level LSH	$L = 10$, $C = 256$

Table II. LSH parameters for synt16, synt64, synt256, mnist, color, and audio datasets.

4.2 Experiment 1: Similarity search performance

LSH based methods report efficient results when adequate values for m (number of hash functions), L (number of indexes), T (number of probes for Multi-probe LSH) are chosen. The LSH parameters (m and L) used in this experiment were tuned according to the implementation of Exact Euclidean LSH (E2LSH) [5]. The tuning parameter m is chosen as a function of the dataset to minimize the running time of a query while the space requirement is within the memory bounds. L is given by $L = m(m-1)/2$. And T is defined using the following reasoning: The original LSH uses L projections to respond a query. The Multi-probe LSH requires fewer indexes to respond a query (say L', $L' < L$). Thus, the number of projections used by the Multi-probe LSH is $L' \times T$ which should be approximately equal to L. This reasoning is not always exact (e.g., non-uniform datasets are especial cases). So for some datasets the T parameter was tuned by hand in order to report comparable search accuracy. The query range for LSH and Multi-probe LSH was set to $r = 10.0\%$ of the largest distance between pairs of objects in the dataset and the approximation ratio was configured to $c = 1.05$ which means 95% of success probability at query time.

Note that our technique only needs information the bucket capacity C and the number of indexes L. Table II shows the LSH parameters used in these experiments for the original LSH, Multi-probe LSH, and Multi-level LSH. For simplicity, in all experiments the page size for Slim-Tree [Traina et al. 2002], DF-Tree [Traina et al. 2002] and DBM-Tree [Vieira et al. 2004] was configured to keep 64 objects per node.

Number of distance Computations and Response Time. The aim of this experiment is to measure the average number of distance computations and the total time spent to retrieve the nearest neighbor objects from a dataset using range queries rQ and k-nearest neighbor queries $kNNq$. The data structures being compared were tested with different radius values for the rQ queries, ranging from 1.0% to 10.0% of the largest distance between pairs of objects in the dataset and from 1 to 10 for the kNN queries. Figures 4, 5 shows the comparison in terms of average number of distance computations

[5] `http://www.mit.edu/~andoni/LSH/`

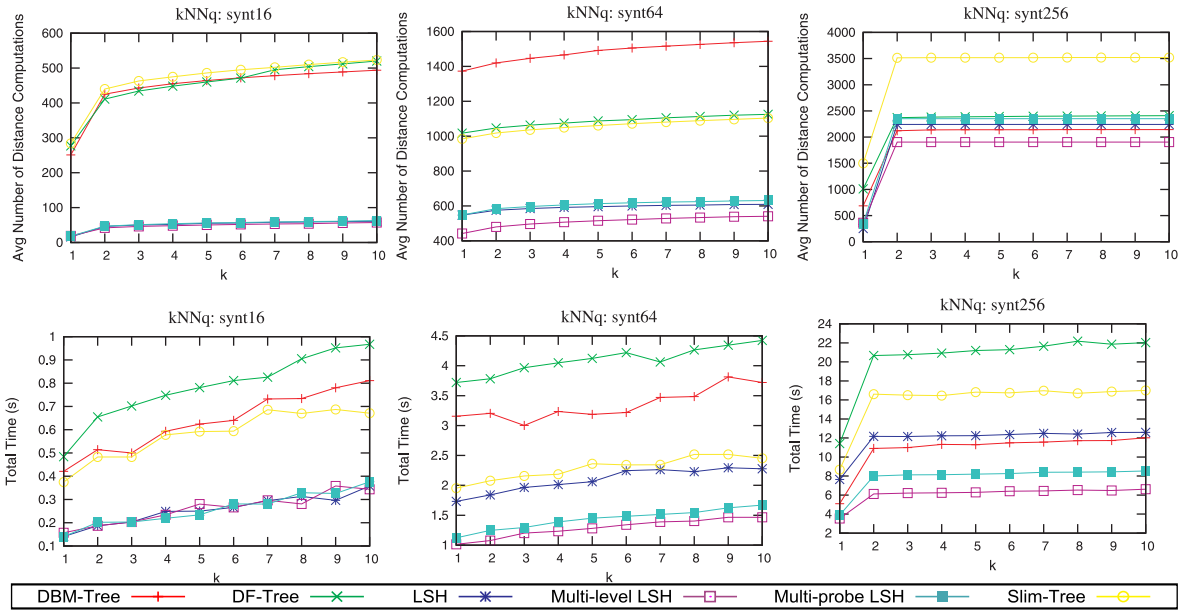


Fig. 4. Comparison of k nearest neighbor queries (kNNq) at various k using the average number of distance computations (first row) and response time (second row) for *synt16* (first column), *synt64* (second column) and *synt256*(third column) datasets.

and total time for all the methods. The basic LSH is faster than metric trees independent of the dimension or size of the dataset. These results indicate that Multi-probe and Multi-level LSH are more efficient than any metric tree considered in this paper. This is expected since metric tree structures suffer from "curse of dimensionality" problem. This implies overlapping among regions when the dimensionality is very high, and as a consequence, they need to explore many paths in the tree structure during the query process.

Tables III and IV shows the average number of distance computation and total response time for all hash based approaches using range queries. Multi-level LSH outperforms the other structures decreasing the query time (number of distance computations) by up to 51% (72%) in comparison to the original LSH and 34% (45%) in comparison to Multi-probe LSH as it can be observed in tables III and IV.

In this experiment, we observed that the structure with the three-level hash tables (for a appropriate parameter C) give a superior performance over one-level hash table (the basic LSH or multi-probe LSH) in term of time and space. Under this configuration our technique is good at distributing objects into buckets uniformly at different levels and quickly retrieving them using hashing functions. This is because in contrast to LSH, Multi-level LSH exploits the multi-resolution index structure to *compute and locate the probes needed* for a specific query. Thus, multiple probes are computed by each index in the query process and as a consequence they do not need more indexes to ensure the same quality results.

4.3 Experiment 2: Accuracy

The goal of this experiment is to evaluate the average accuracy of our approach against other well-known indexes. Given an index and a dataset, we evaluate the average accuracy by performing a range query and k-nearest neighbor query. For each query result we check if it includes the same elements reported by a linear scan with euclidean distance as dissimilarity function. After repeating

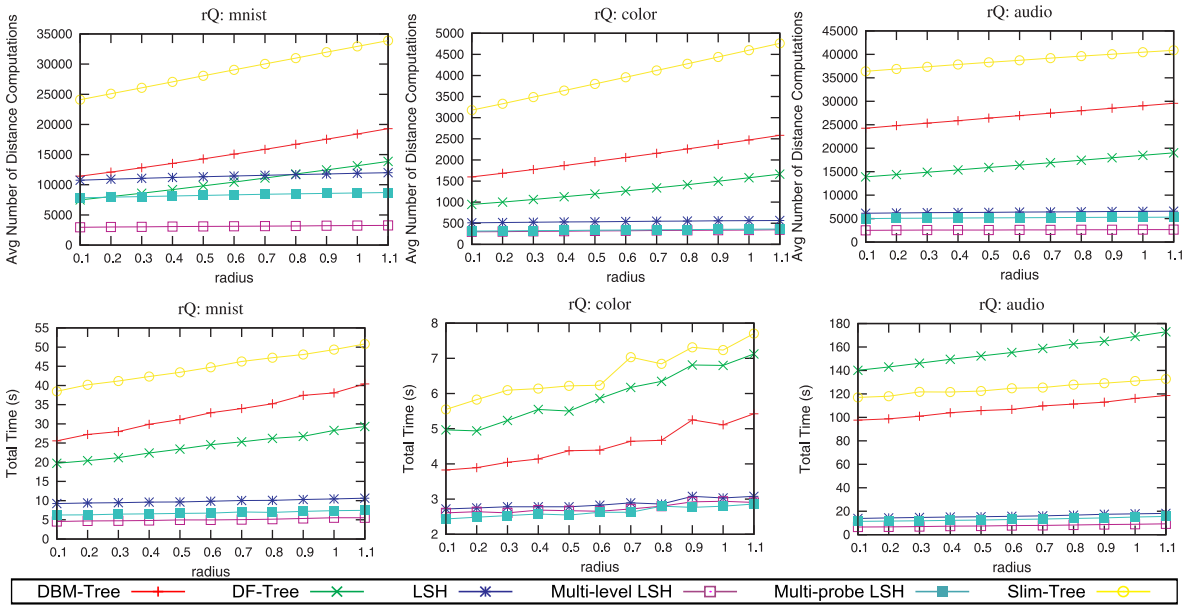


Fig. 5. Comparison of range queries (rQ) at various radii using the average number of distance computations (first row) and response time (second row) for *mnist* (first column), *color* (second column) and *audio*(third column) datasets.

	color		**minst**		**audio**		**synt256**	
	NDC	%gained	NDC	%gained	NDC	%gained	NDC	%gained
LSH	541	0,00%	11.427	0,00%	6.373,36	0,00%	2.207	4,58%
Multi-probe LSH	340	37,12%	8.310	27,28%	5.175,18	18,80%	2.313	0,00%
Multi-level LSH	322	40,38%	3.116	72,73%	2.588,36	59,39%	1.854	19,84%

Table III. Comparison of Number of Distance Computation (NDC) for each LSH method. Results for the color (32-D), minst (50-D), audio (190-D) and synt256 (265-D) datasets

	color		**minst**		**audio**		**synt256**	
	TT	%gained	TT	%gained	TT	%gained	TT	%gained
LSH	2,87	0,00%	9,88	0,00%	15,892	0,00%	12,588	0,00%
Multi-probe LSH	2,64	8,01%	6,82	30,94%	13,133	17,36%	8,715	30,77%
Multi-level LSH	2,74	4,53%	5,05	48,94%	7,776	51,07%	6,345	49,59%

Table IV. Comparison of Total Time (TT) for each LSH method. Results for the color (32-D), minst (50-D), audio (190-D) and synt256 (265-D) datasets

this process for all objects in the test set (500 iterations), we average the precision values of the index. We perform a similar procedure to that of Experiment 1 to evaluate the similarity search performance and show the results in Table V for all hash-based approaches (LSH, Multi-probe LSH, and Multi-level LSH). Since metric trees report exact result only results for hash-based approaches are shown. This is because the distance function used in this experiment defines a suitable metric space to perform similarity queries. In contrast, LSH is based on projection into subspaces in a approximately way and only can report good results for ranges less than $c \times r$. Even though, Multi-probe and Multi-level LSH overcame this problem by using multiples probes in each index, the quality and efficiency has a certain umbral limit for some greatest k in kNN queries or greatest r in range queries as we will see in the scalability experiments.

	color			minst			audio		
	NDC	%gained	accuracy	NDC	%gained	accuracy	NDC	%gained	accuracy
LSH	541	0,00%	0,99	11.427	0,00%	1,00	6.373	0,00%	0,99
Multi-probe LSH	340	37,12%	0,99	8.310	27,28%	1,00	5.175	18,80%	0,99
Multi-level LSH	322	40,38%	0,99	3.116	72,73%	0,995	2.588	59,39%	0,94

Table V. Average accuracy for the color (32-D), minst (50-D) and audio (190-D) datasets.

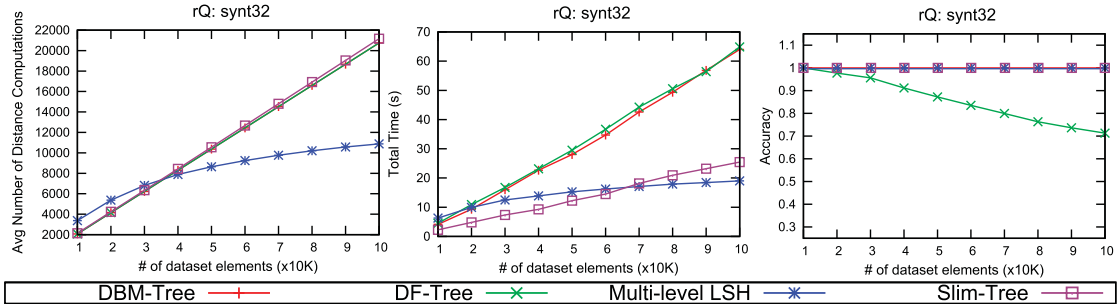


Fig. 6. Comparison of range query (rQ) at 5.0%. The average number of distance computations (first column), response time (second column) and accuracy (third column) is shown for *synt32* dataset.

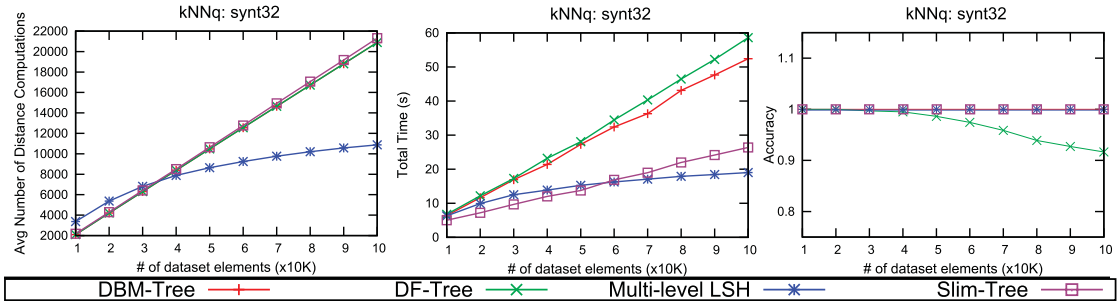


Fig. 7. Comparison of 100-nearest neighbor query (kNNq). The average number of distance computations (first column), response time (second column) and accuracy (third column) is shown for *synt32* dataset.

4.4 Experiment 3: Scalability

In this experiment, we want to study the behavior of our technique and dynamic indexes, Slim-Tree [Traina et al. 2002], DF-Tree [Traina et al. 2002] and DBM-Tree [Vieira et al. 2004], as the size of the dataset increases. We vary the size of the dataset and measure performance to determine scalability. We split the synt32 dataset by 10. After inserting each subset we run sets of queries executing 500 similarity queries. The behavior was equivalent for different values of k and radius, thus we present only the result for $k = 100$ and radius $= 5.0\%$ of the largest distance between pairs of objects in the dataset. Figures 6, 7 show the average number of distance computations, response time and accuracy for synt32 dataset using range queries and k-nearest neighbors queries. Multi-level LSH shows sub-linear behavior when the number of elements indexed grows, what makes the scheme sufficient to index very large datasets. Moreover, Multi-level LSH exhibits good accuracy while the bucket capacity is near to the query condition. For range queries, our approach shows reasonable accuracy ($\geq 90\%$) while the radius is less than 2.5% of the largest distance between pairs of objects in the dataset. For greater ranges, increasing of the number of probes should be sufficient and if it is not the case, the number of indexes can be increased in order to ensure satisfactory results.

4.5 Experiment 4: Usage Space

Table VI shows the memory used in Megabytes by LSH, Multi-probe LSH, Multi-level LSH, Slim-Tree, and DF-Tree. Only the well-balanced tree (Slim-Tree) and the tree designed to prune more sub trees (DF-Tree) are considered for this experiment because they have the minimum and maximum usage space uses among the trees analyzed. We take the number of buckets and nodes with their respectively capacity as measures to compute the space usage. LSH needs more memory than metric trees; however, Multi-level LSH reduces space usage by up to 35% in comparison to the original LSH and it has similar scores in comparison to Multi-probe LSH. This is expected since Multi-level LSH and Multi-probe LSH use fewer number of indexes due to instead of probing only one bucket for each hash table in the query algorithm, these methods use multiple buckets that are likely to contain query results in a hash table, as a result, fewer indexes are needed. An interesting observation about Multi-probe LSH, Multi-level LSH and DF-Tree (the tree designed to prune more sub trees) is that they use a quite similar amount of memory.

method / dataset	synt16	synt64	synt256	mnist	color	audio
LSH	12	58	252	740	1404	1428
Multi-probe LSH	3	14	63	185	351	376
Multi-level LSH	3	15	88	235	352	546
Slim-Tree	2	8	47	29	23	94
DF-Tree	3	23	95	28	38	258

Table VI. Space usage experiments. Comparison of the memory used by LSH, Multi-probe LSH, Multi-level LSH, Slim-Tree, and DF-Tree in Megabytes

5. CONCLUSIONS

In this paper, we presented a new scheme to solve approximate similarity search called Adaptive Multi-level Locality Sensitive Hashing. Our approach considers the linear and multi-level hashing scheme to adjust the number of hash functions and number of buckets needed to index a dynamic set of objects. Due to the self-adaptive abilities of the multi-resolution index structure we can adapt the data domain parameters during the indexing process. Additionally, this scheme allows us to exploit the multi-resolution approach to compute the number of probes needed for a specific query. As a consequence, we can speed up the query process as the need of more indexes decreases.

We conducted performance studies on many real and synthetic datasets. The empirical results show that Multi-level LSH outperforms the metric data structures and LSH-based methods by decreasing the query time (number of distance computations) by up to 51% (72%) in comparison to the original LSH and 34% (45%) in comparison to Multi-probe LSH. Additionally, Multi-level LSH reduces space usage by up to 35% in comparison to the original LSH and it has similar results in comparison to Multi-probe LSH. Our results show the self-tuning behavior of our approach during the indexing process offering sub-linear cost for query processing and preserving an acceptable accuracy score. Additionally, our experimental studies show that, current state-of-the-art metric trees exhibit exact precision and satisfactory performance and usage space. However, even though LSH methods have a trade-off among space, speed and quality, they are still more effective and faster than metric trees specially in sceneries where the dataset size and dimension are very high. That was expected but not clear in the literature considering that ,in contrast to LSH methods, metric tree structures suffer from the "curse of dimensionality" problem.

REFERENCES

Andoni, A. and Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM* 51 (1): 117–122, 2008.

Bawa, M., Condie, T., and Ganesan, P. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*. Chiba, Japan, pp. 651–660, 2005.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Record* 19 (2): 322–331, 1990.

Bentley, J. L. Multidimensional binary search trees in database applications. *Transactions on Software Engineering* 5 (4): 333–340, 1979.

Berchtold, S., Keim, D. A., and Kriegel, H.-P. The X-tree: An index structure for high-dimensional data. In *Proceedings of the International Conference on Very Large Data Bases*. San Francisco, CA, USA, pp. 28–39, 1996.

Böhm, C., Berchtold, S., and Keim, D. A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys* 33 (3): 322–373, 2001.

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

Ciaccia, P., Patella, M., and Zezula, P. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the International Conference on Very Large Data Bases*. San Francisco, CA, USA, pp. 426–435, 1997.

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth annual Symposium on Computational Geometry*. Brooklyn, New York, USA, pp. 253–262, 2004.

Dong, W., Wang, Z., Josephson, W., Charikar, M., and Li, K. Modeling LSH for performance tuning. In *Proceedings of the International Conference on Information and Knowledge Engineering*. Napa Valley, California, USA, pp. 669–678, 2008.

Gaede, V. and Günther, O. Multidimensional Access Methods. *ACM Computing Surveys* 30 (2): 170–231, 1998.

Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases*. San Francisco, CA, USA, pp. 518–529, 1999.

Guttman, A. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. New York, NY, USA, pp. 47–57, 1984.

Hjaltason, G. R. and Samet, H. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems* 28 (4): 517–580, 2003.

Litwin, W. Linear hashing: a new tool for file and table addressing. In *Readings in database systems*. pp. 570–581, 1988.

Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the International Conference on Very Large Data Bases*. Vienna, Austria, pp. 950–961, 2007.

Navarro, G. Searching in metric spaces by spatial approximation. *The VLDB Journal* 11 (1): 28–46, 2002.

Sellis, T., Roussopoulos, N., and Faloutsos, C. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the International Conference on Very Large Data Bases*. New York, NY, USA, pp. 507–518, 1987.

Slaney, M. and Casey, M. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal Processing Magazine* 25 (2): 128–131, 2008.

Tao, Y., Yi, K., Sheng, C., and Kalnis, P. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data Conference*. Providence, Rhode Island, USA, pp. 563–576, 2009.

Traina, Jr., C., Traina, A., Faloutsos, C., and Seeger, B. Fast indexing and visualization of metric data sets using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering* 14 (2): 244–260, 2002.

Traina, Jr., C., Traina, A., Filho, R. S., and Faloutsos, C. How to improve the pruning ability of dynamic metric access methods. In *Proceedings of the International Conference on Information and Knowledge Engineering*. McLean, Virginia, USA, pp. 219–226, 2002.

Ullman, J. D., Garcia-Molina, H., and Widom, J. Index Structures. In , *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, pp. 649–660, 2001.

Vieira, M. R., Jr., C. T., Chino, F. J. T., and Traina, A. J. M. DBM-tree: A dynamic metric access method sensitive to local density data. In *Proceedings of the Brazilian Symposium on Databases*. Brasília, Brasil, pp. 163–177, 2004.

Yianilos, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Philadelphia, PA, USA, pp. 311–321, 1993.