Efficient Incremental Near Duplicate Detection Based on Locality Sensitive Hashing

Marco Fisichella, Fan Deng, and Wolfgang Nejdl

Forschungszentrum L3S, Hannover 30167, Germany
{fisichella,deng,nejdl}@L3S.de

Abstract. In this paper, we study the problem of detecting near duplicates for high dimensional data points in an incremental manner. For example, for an image sharing website, it would be a desirable feature if near-duplicates can be detected whenever a user uploads a new image into the website so that the user can take some action such as stopping the upload or reporting an illegal copy. Specifically, whenever a new point arrives, our goal is to find all points within an existing point set that are close to the new point based on a given distance function and a distance threshold before the new point is inserted into the data set. Based on a well-known indexing technique, Locality Sensitive Hashing, we propose a new approach which clearly speeds up the running time of LSH indexing while using only a small amount of extra space. The idea is to store a small fraction of near duplicate pairs within the existing point set which are found when they are inserted into the data set, and use them to prune LSH candidate sets for the newly arrived point. Extensive experiments based on three real-world data sets show that our method consistently outperforms the original LSH approach: to reach the same query response time, our method needs significantly less memory than the original LSH approach. Meanwhile, the LSH theoretical guarantee on the quality of the search result is preserved by our approach. Furthermore, it is easy to implement our approach based on LSH.

1 Introduction

Similarity search is an important research topic which finds applications in different areas. For example, finding all similar images of a query image in a large image collection based on certain similarity measures and thresholds. Feature vectors can be extracted from the images. Once this is done, the set of images can be considered as a set of high dimensional points. In general, similarity search can refer to a variety of related problems. In this paper, the problem we consider is to answer range search queries in an incremental manner. That is, whenever a new point arrives, find all similar/close points (based on a pre-specified similarity threshold) from the set of high dimensional points arrived earlier, and then insert the new point into the data set.

P. García Bringas et al. (Eds.): DEXA 2010, Part I, LNCS 6261, pp. 152–166, 2010.
© Springer-Verlag Berlin Heidelberg 2010

The motivating application of this work is online near duplicate detection for multimedia content sharing websites like Flickr[1] and Youtube[2]. Whenever a user is uploading an image or video, it would be desirable if near-duplicates that are very similar (content-wise) to the one being uploaded can be retrieved and returned to the user in real-time. In this way, the user can identify redundant copies of the object promptly and decide if he/she should continue the upload. In addition to personal users, enterprise users may also need this type of applications. For example, media companies such as broadcasters and newspapers may continuously upload their images or videos to a multimedia content repository. The copy right issue is one of their main concerns. It would be a useful feature if near-duplicate copies can be retrieved and reported to the users during the upload period so that the user can identify pirated copies promptly. If the new object is illegal, the user should immediately stop the upload process.

Compared to the traditional similarity search problem, fast response is more important for this type of applications since similarity search is only part of the online content upload process which must be completed within at most a few seconds. In addition to the online requirement, another characteristic of the motivating applications is that the similarity search operations is executed together with data point insertions. In other words, the data set is created incrementally where the near neighbors of each point are known before the point is inserted into the data set.

To speed up the searching process, in-memory indexing techniques are ideal solutions if the help of disk-based index are not necessary since a disk access is an order of magnitude slower than a memory operation. For a data set with 1 million points, an index storing all the point IDs once only needs 12MB memory assuming each ID takes 12 bytes; if each point is a 162-dimensional point and each dimension of a point takes 4 bytes, storing all the points needs 648MB, which is tolerable even for an inexpensive PC nowadays. Although processing Web-scale data set with billions of points may need clusters with tens or hundreds of distributed machines, indexing an enterprise-scale data set with tens or hundreds of millions points in main-memory is feasible using a single server with a larger memory size. Unfortunately, to give a fast query response, the index size needed for high-dimensional points is usually larger than the size we computed, and it can be even larger than the data set size. Thus, in this work we focus on reducing memory consumption of in-memory index while providing fast query response.

Although decades of research have been conducted on similarity search, the problem is still considered challenging. One important reason is the "curse of dimensionality". It has been shown that exponential space in n (number of points in the data set) is needed to speed up the similarity search process or the searching time increases exponentially with the dimensionality [2,4]. It is also shown both theoretically and empirically [24] that all partitioning and clustering based indexing approaches degrade to a brute force linear scan approach when the dimensionality is sufficiently high.

[1] http://www.flickr.com

[2] http://www.youtube.com

To our knowledge, a state-of-the-art solution to the similarity search problem in practice, which provides fast query response time, is Locality Sensitive Hashing (LSH)[12,15] although it has been proposed for a decade. Meanwhile, LSH also provides theoretical guarantees on the quality of the solution. However, also suffering from the "curse of dimensionality", LSH needs large amount of space to achieve fast query response.

1.1 Our Contributions

- We proposed a novel approach, SimPair LSH, to speed up the original LSH method; the main idea is to take advantage of a certain number of existing similar point pairs, and use them to prune LSH candidate sets relevant for a given query.
- The correctness and effectiveness of the new approach is analyzed.
- Thorough experiments conducted on 3 real-world data sets show that our method consistently outperforms LSH in terms of query time in all cases that we tried, with a small amount of extra memory cost. To achieve the same query time saving, we show that LSH need significantly more space. Meanwhile, we show that our method preserves the important theoretical guarantee on the recall of query answers.

2 Preliminaries and Related Work

2.1 Problem Statement (Incremental Range Search)

In this paper, we focus on the incremental range search problem defined as follows: given a point q and a set P with n d-dimensional points, efficiently find out all points in P that are similar to q based on certain similarity/distance function and a similarity threshold τ before q is inserted into the data set. We call the points similar to q *near neighbors* of q. In this problem, before evaluating the query q, the near neighbors of all points within the data set are retrieved when they are inserted into the data set.

Distance measure. We focus on Euclidean distance since it has been widely used in different applications. It is not hard to extend the technique to other distance functions such as L1 and Hamming distance, as the underlying technique, Locality Sensitive Hashing, can be applied in those cases.

In-memory index structure. We focus on in-memory index structure since fast real-time response is the first priority in the applications we consider. For high dimensional similarity search, the index size can be as large as or even larger than the data set size in order to give an efficient query response time. Therefore, reducing the memory cost while providing fast response is the main concern of this work.

2.2 Straightforward Solution

A straightforward solution to this problem is LinearScan: compute the distance between q and each point p in P; if the similarity is above the given similarity

threshold, output this point. It is not hard to see that this approach can be very slow for large data sets, especially when the dimensionality d is large; in the case of Euclidean distance, LinearScan takes $O(nd)$ time for each query.

2.3 Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) [15,12] was proposed by Indyk and Motwani and finds applications in different areas including multimedia near duplicate detection (e.g. [9], [11], [17]). LSH was first applied in indexing high-dimensional points for Hamming distance [12], and later extended to L_p distance [10] where L_2 is Euclidean distance, which we will use in this paper.

The basic idea of LSH is to use certain hash functions to map each multi-dimensional point into a scalar; the hash functions used have the property that similar points have higher probability to be mapped together than dissimilar points. When LSH is used for indexing a set of points to speed up similarity search, the procedure is as follows: first, create an index (a hash table) by hashing all points in the data set P into different buckets based on their hash values; select L hash functions uniformly at random from a LSH hash function family and create L hash tables; when the query point q arrives, use the same set of hash functions to map q into L buckets, one from each hash table; retrieve all points from the L buckets into a candidate set C and remove duplicate points in C; for each point in C compute its distance to q and output those points similar to q.

An essential part of LSH is the hash function family H. For Euclidean distance, the hash function family can be constructed as follows [10]: map a multi-dimensional point p into a scalar by using the function $h(p) = \lfloor \frac{a \cdot p + b}{r} \rfloor$ where a is a random vector whose coordinates are picked uniformly at random from a normal distribution, and b is a random variable uniformly distributed in the range $[0, r]$. In this hash function, the dot product $a \cdot p$ is projecting each multi-dimensional point p into a random line; the line is cut into multiple intervals with length r; the hash value shows which interval p is mapped to after a random shift of length b. Intuitively, it is clear that closer points have higher chance being mapped into the same interval than distant points under this random projection. Last, generate a new hash function $g(p)$ to be used in constructing a hash table by concatenating k $h_i(p)$ $(i = 1 \dots k)$, each chosen uniformly at random from H, i.e. $g(p) = (h_1(p), \dots, h_k(p))$.

The nice property of the LSH is that the probability that two points p_1 and p_2 are hashed into the same bucket is proportional to their distance c, and this probability can be explicitly computed using the following formulas:

$$p(c) = Pr[h(p_1) = h(p_2)] = \int_0^r \left(\frac{1}{c}\right) f\left(\frac{t}{c}\right) \left(1 - \frac{t}{r}\right) dt, \quad (1)$$

where $f(t)$ is the probability density function of the absolute value of the normal distribution. Having $p(c)$, we can further compute the collision probability under H:

$$P(c) = Pr[H(p_1) = H(p_2)] = 1 - (1 - p(c)^k)^L. \quad (2)$$

2.4 Other LSH-Based Approaches

Since proposed, LSH has been extended in different directions. Lv et al. [20] proposed multi-probe LSH, and showed experimentally that their method significantly reduced space cost while achieving the same search quality and similar time efficiency compared with original LSH. The key idea of multi-probe LSH is that the algorithm not only searches for the near neighbors in the buckets to which the query point q is hashed, it also searches the buckets where the near neighbors have slightly less chance to appear. The benefit of multi-probe LSH is that each hash table can be better utilized since more than one bucket of a hash table is checked, which decreases the number of hash tables. However, multi-probe LSH does not provide the important search quality guarantee as LSH does. The original LSH scheme guarantees that the true results will be returned by the search algorithm with high probability, while multi-probe could not. This makes multi-probe LSH not applicable in those applications where the quality of the retrieval results are required to be guaranteed. The idea of multi-probe LSH was inspired by earlier work investigating entropy-based LSH [21]. The key idea is to guess which buckets the near neighbors of q are likely to appear in by randomly generating some "probing" near neighbors and checking their hash values. Similar to multi-probe LSH, entropy-based LSH also reduces the number of hash tables required. In practice, though, it is difficult to generate proper "probing" near neighbors in a data-independent way [20].

Another extension of LSH is LSH forest [5] where multiple hash tables with different parameter settings are constructed such that different queries can be handled with different settings. In the theory community, a near-optimal LSH [2] has been proposed; however, currently it is mostly of theoretical interest because the asymptotic running time improvement is achieved only for a very large number of input points [4]. More LSH related work can be found in a recent survey [4]. This survey also observes, that despite decades of research, current solutions still suffer from the "the curse of dimensionality". In fact, for a large enough dimensionality, current solutions provide little improvement over LinearScan, both in theory and in practice [4]. We further note that the technique in this paper is orthogonal to the other LSH variants described above and can be applied in those scenarios.

2.5 Tree-Based Indexing Techniques

When the dimensionality is relatively low (e.g. 10 or 20), tree-based indexing techniques are known to be efficient. Examples include kd-trees [6], R-tree [14], SR-tree [16], cover-trees [8] and navigating-nets [19]. These methods do not scale well with the (intrinsic) dimensionality. Weber et al. [24] show that when the dimensionality exceeds 10, all space partitioning and clustering based indexing techniques degrade to LinearScan. For indexing high dimensional points, B+ tree is also used together with different techniques handling the "dimensionality curse", such as iDistance [25] and LDC [18]. Other tree-based approaches like IQ-tree [7] and A-tree [22] use a smaller vector to represent the data points

approximately which helps to reduce the complexity of the problem. Different from the LSH based approaches where large amount of space is traded for gaining fast response time, the tree-based approaches have less concern on index space while they usually have faster but comparable query time as LinearScan.

Due to the intensive research within the past decades, there are a large body of related literature which cannot be covered here. Samet's book [23] provides a comprehensive survey on this topic.

3 SimPair LSH

Our approach is based on the standard LSH indexing, and takes advantage of existing similar pair information to accelerate the running time of LSH. We thus call it *SimPair LSH*. Unless noted otherwise, LSH denotes the original LSH indexing method in the rest of this paper.

3.1 Key Idea

We observe that LSH retrieves all points stored in the buckets a query point q hashed to. Let the set of points returned by LSH be the candidate set C. Then q is compared with all the points in C as in LinearScan, and the near neighbors are found. To guarantee a low chance of missing a near neighbor in C, a large number of hash tables has to be created which may lead to a large C depending on the query q, and accordingly increase the running time especially when d is large.

The main idea of this paper is to take advantage of a certain number of pair-wise similar points in the data set and store them in memory; in the process of scanning through C, the search algorithm can look up the similar pair list on-the-fly whenever a distance computation between q and a point p in C is done; if a similar pair $(p, p\prime)$ is found in the list, it is very likely that $p\prime$ will also appear in C; based on the known distances $d(q, p)$ and $d(p, p\prime)$ we can infer an upper bound to $d(q, p\prime)$ by using triangle inequality and may skip the distance computation between q and $p\prime$. The reason this idea works is that LSH tends to group similar objects into the candidate set C. Thus the points in C are very likely to be similar to each other. Checking one point p can avoid computing distance for the points similar to p, and therefore saving distance computations.

3.2 The SimPair LSH Algorithm

Our SimPair LSH algorithm works as follows: given a set of points P and all point pairs (including their distances) whose pair-wise distances are smaller than a threshold θ (let the set of all similar pairs be SP). Also given the distance threshold τ determining near neighbors, SimPair LSH then creates L indices as in LSH; whenever a query point q comes, SimPair LSH retrieves all points in the buckets to which q is hashed. Let this set of points be the candidate set C. Instead of scanning through all the points p in C one by one and compute their

distances to q as in LSH, SimPair LSH checks the pre-computed similar pair set SP whenever a distance computation $d(q, p)$ is done. Based on the distance between p and q, SimPair LSH continues in 2 different ways:

- If $d(q, p) <= \tau$, SimPair LSH searches in SP for all points $p\prime$ which satisfies $d(p, p\prime) <= \tau - d(q, p)$; check if $p\prime$ in the candidate set C or not; if yes, then mark $p\prime$ as a near neighbor of q without the distance computation.
- If $d(q, p) > \tau$, SimPair LSH searches in SP for all those points $p\prime$ which satisfies $d(p, p\prime) < d(q, p) - \tau$; check if $p\prime$ in the candidate set C or not; if yes, then remove $p\prime$ from C without the distance computation.

The detailed description is shown in Algorithm 1:

Algorithm 1. SimPair LSH

```
Input: A set P with n d-dimensional points; L in-memory hash tables created
       by LSH; a set SP storing all similar pairs in P whose pair-wise
       distances are smaller than θ; a distance threshold τ defining near
       neighbors; and a query point q
Output: all near neighbors of q in P
begin
    check the L buckets q hashed to and retrieve all the points in those buckets
    as in LSH;
    put all the points into a candidate set C;
    for each point p in C do
        compute the distance between q and p, i.e. d(q,p);
        if d(q,p) < τ then
            output p as a near neighbor of q;
            search in SP for all the points p′ which satisfies d(p,p′) < τ − d(q,p);
            for each point p′ found in SP do
                check if p′ in C or not;
                if found then
                    output p′ as a near neighbor of q and remove it from C;
        if d(q,p) > τ then
            search in SP for all the points p′ which satisfies d(p,p′) < d(q,p) − τ;
            for each point p′ found in SP do
                check if p′ in C or not;
                if found then
                    remove p′ from C;
end
```

The algorithm constructing the LSH indices is the original LSH algorithm. [10] describes how to select L and g_i to guarantee the success probability.

3.3 Algorithm Correctness

Since our algorithm is based on LSH, it is important that the theoretical guarantee still holds for SimPair LSH.

Theorem 1. *SimPair LSH has the same theoretical guarantee as LSH has in terms of the range search problem we study. That is, near neighbors will be returned by SimPair LSH with a user-specified probability by adjusting the parameters (hash functions and number of hash tables) accordingly.*

Proof. Since we consider points in metric space where triangle inequality holds, SimPair LSH guarantees that the points skipped are either true near neighbors or not near neighbors without distance computation.

3.4 Algorithm Effectiveness

The benefit of SimPair LSH compared with LSH is that points in the candidate set returned by LSH can be pruned by checking the similar pair list SP without distance computations. Therefore, it is important to analyze the number of prunes SimPair generates. Also, to obtain the benefit, SimPair LSH has to search in SP and C for the points to be pruned, which can take time although hash indices can be built to speed up each search operation to $O(1)$ time. Next, we analyze the factors affecting the gain and cost.

Pruning analysis. To generate a prune from a point p in C, SimPair first has to find a "close enough" point $p\prime$ of p from SP, where close enough or not depends on $|d(q,p) - \tau|$. If $|d(q,p) - \tau|$ is large, SimPair LSH has a higher chance to find a $p\prime$.

Another factor that can affect the chance of finding $p\prime$ from SP is the size of SP. Clearly, maintaining a large set of SP will increase the chance of finding $p\prime$ of p.

Finding $p\prime$ of p does not necessarily lead to a prune. The condition that a prune occurs is that $p\prime$ appears in C. According to the property of LSH hash functions, points close to q have higher chance appearing in C. In other words, $d(q,p\prime)$ determines the chance of generating a prune. Although $d(q,p\prime)$ can not be known precisely, a bound of this distance can be derived from $d(q,p)$ and the "close enough" threshold $|d(q,p) - \tau|$.

Cost analysis. To gain the pruning, SimPair LSH has to pay certain amount of costs including time and space costs. The time cost mainly comes from the searching processes: find the points "close enough" to p in SP and check those points to see if they are in C or not. By constructing hash indices for SP, searching for p in SP only takes $O(1)$ time; constructing hash indices for SP also takes $O(1)$ time for each object. When a candidate set C of points for the query q is retrieved, all points in the dataset belonging to C are marked both in LSH and SimPair LSH; this is possible since each point in the data set has a Boolean attribute showing if the point is in C or not. The purpose of having this attribute is to remove duplicate points when generating C. Duplicates can appear in C because one point can appear in multiple LSH hash buckets. Note that when the searching is finished, the boolean attributes need to be cleared (for both LSH and SimPair LSH) which takes $O(|C|)$ time when all points in C are also maintained in a linked list. For the sake of pruning, another boolean attribute is needed for each point to indicate if the point has been pruned or not.

With these boolean attributes, searching for $p\prime$ in C takes $O(1)$ time. The time cost is mainly generated by searching $p\prime$ in C since there can be multiple $p\prime$ for each p, and therefore multiple look-ups in C.

In addition to the time cost, SimPair LSH also has some extra space cost for storing SP compared with LSH. This cost is limited by the available memory. In our approach, we always limit the size of SP based on two constraints: (i) the similarity threshold θ (for the similar point pairs stored in SP) is restricted to the range $(0, \tau]$; (ii) the size of SP must not exceed a constant fraction of the index size (e.g. 10%).

4 Experiments

In this section, we demonstrate the practical performance of our approach on three real-world data sets, testing the pruning effectiveness, pruning costs, real running time together with memory saving and quality of results from SimPair LSH.

4.1 Data Sets

We use three real-world image data sets in our experiments: one directly downloaded from a public website and two generated by crawling commercial multimedia websites.

Flickr images. We sent 26 random queries to Flickr and retrieved all the images within the result set. After removing all the images with less than 150 pixels, we obtained approximately $55,000$ images.

Tiny images. We downloaded a publicly available data set with 1 million tiny images[3]. The images were collected from online search tools by sending words as queries, and the first 30 returned images for each query are stored. Due to the high memory cost of LSH for large data sets, we picked 50 thousand images uniformly at random from this 1 million tiny image data set. This random selection operation also reduced the chance that similar pairs appear in the data set since the images retrieved from the result set of one query have higher chance to be similar to each other.

The reason we used this smaller data set rather than only considering the full set was that we could vary the number of hash tables within a larger range and observe the behavior of the algorithms under different number of hash tables. For example, the largest number of hash tables we used was about 1000; indexing the 1 million data points takes 12GB memory under this setting which was above the memory limit of our machine. (Note that this is an extreme case for experimental purpose and may not be necessary in practice.) If we used 10 hash tables, then the memory consumption will drop to 120MB. We also conducted experiments on the whole 1 million data set setting the number of hash tables

[3] The dataset was collected by A. Torralba and R. Fergus and W. T. Freeman at MIT in 2007; it is available at `http://dspace.mit.edu/handle/1721.1/37291`

to smaller values, so as to see how the algorithms behave with a larger data set size. Due to the space limit, the results from the 1 million data set are reported into the complete version of this paper [1].

Video key-frames. We sent 10 random queries to Youtube and obtained around 200 video clips from each result set, approximately 2100 short videos in total. We then extracted all the frames of the videos and their HSV histograms with dimensionality 162. After that, we extracted key frames of the videos in the following way: sequentially scan the HSV histogram of each frame in a video; if the distance between the current histogram and the previous one in the video is above 0.1 , keep this histogram; otherwise skip it. We set the distance threshold to 0.1 because two images with this distance are similar but one can clearly see their difference based on our observation. In the end, we obtained $165,000$ key-frame images.

For all the images data set described above, we removed duplicates and converted each data set into a d-dimensional vector ($d = 162, 512$) by using the standard HSV histogram methods [13]. Each entry of the vector represents the percentage of pixels in a given HSV interval.

Pair-wise distance distribution. Since the pair-wise distance distribution of data set may affect the result of our experiments, we plotted Figures and found that the 3 data sets had similar curves. Specifically, we cut the distance range into multiple intervals and count the number of points within each interval. Due to the space limit, we put the histograms into the complete version of this paper [1].

4.2 Experimental Setup

All experiments were ran on a machine with an Intel T2500 2GHz processor, 2GB memory under OS Fedora 9. The algorithms were all implemented in C compiled by gcc 4.3.0.

The data points and the LSH indices are both loaded into the main memory. Each index entry for a point takes 12 bytes memory. To test the performance of our approach, we randomly selected a certain number of objects from the data set as query objects, and measure metrics as discussed before. We took the average number of pruned points of all queries, the average percentage of pruned points, and the average number of operations spent on achieving the pruning per point (average number of cost operations per query / average size of candidate sets C of all queries).

4.3 Experiments Testing Pruning Effectiveness and Costs

In this set of experiments, we tested the number of distance computations saved by our approach, the time and space cost to obtain the saving. We used the Flickr data set, and the results from other data sets are also consistent in gerneral.

From our experiments we can see that the algorithm performance is not sensitive to the number of queries, and we fixed the number of queries to 100 in the following experiments.

Since candidate set sizes $|C|$ for some queries is quite small (e.g. < 50), and there is no need to start the pruning process, we set a cut–off threshold T for $|C|$. When $|C| > T$, SimPlair LSH start the pruning process; otherwise, SimPlair LSH does not start the pruning process and degrades to the original LSH. From our experiments we can see that the algorithm performance is not sensitive to T in terms of both the percentage of pruned points and the average number of operations cost per point. In the rest of the experiments, we fixed T to 200.

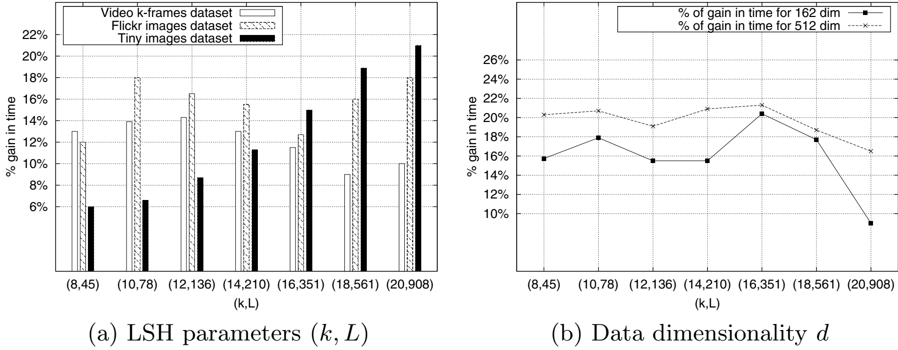
Due to the space limit, the results of testing the number of queries and T are not shown here, but can be found in the complete version of the paper [1].

4.4 Experiments Testing the Query Response Time

In this set of experiments, we report the query response time of the original LSH indexing and our approach. The LSH code were obtained from Andoni [3], and we conducted the experiments for LSH without changing the original source code.

Hash function time costs. Note that during query time, generating the hash values of each query also takes time where the amount depends on the number of hash functions or hash tables used. In the case that the candidate set size is relatively small and the number of hash functions is large, the hashing process can take as high as the time spent on scanning the candidate set. Since the time spent on generating the hash values are exactly the same for both SimPair LSH and the original LSH, and the percentage of this portion varies significantly with the parameter setting of L and the size of C which depends on the queries, we only report the time spent on finding near neighbors from C to see the difference between our approach and LSH better. When the size of C is relatively large, the fraction of hash function time cost is relatively small. But in the worst case where hashing queries take the same amount of time as scanning C, the time difference between two approach will be half of the numbers reported below.

Varying k and L. We varied the hash function parameter k and the number of hash tables L to see how these parameters affect the query time. The threshold τ was set to 0.1, success probability P was set to 95%. Note that once k were fixed, the number of hash tables L was also fixed to guarantee the required success probability. Other parameters are the same as in the previous experiments. The results on the 3 data sets are shown in Figure 1a. Y-axis is the response time saved by SimPair LSH computed as follows: (LSH Time - SimPair LSH Time)/(LSH time). From the figure we can see that SimPair LSH consistently outperforms LSH under different settings of K and L. The extra memory consumptions of the full similar pair set SP when $\theta = 0.1$ are $73.7MB$, $12.5MB$ and $3.7MB$ respectively for the video key frame, Tiny images and Flickr image data sets. Recall that SP size is bound to the 10% of the index size, thus when L



(a) LSH parameters (k, L) (b) Data dimensionality d

Fig. 1. LSH parameters (k, L) and data dimensionality vs. running time

is small, we use a smaller θ. Since LSH can also save running time by increasing the number of hash tables (varying K and L), we tested how much additional memory LSH needs to gain the same amount of time in the next experiments.

Extra space cost comparison showing the significance of our time gain. To achieve the response time gain, in addition to the memory cost of the LSH indices, the extra cost SimPair LSH takes is the memory spent on the similar pair list that we restricted at most to a constant fraction of the LSH indices (10% in our experiments). To achieve approximately the same running time gain, LSH can also increase the memory consumption by increasing k and L without resorting to our approach. Therefore, we compared the memory consumption of the two approaches to achieve roughly the same query time improvement. In fact, the memory cost of LSH can be computed from L: each hash table stores the identifiers of all n points, and each identifier takes 12 bytes as implemented by Andoni and Indyk [3]; therefore, the LSH space cost is $12nL$.

Hence, to see the size of extra space LSH needs, it suffices to check the value of L. Since the size of C dominates the time LSH scans through the candidate set, the running time being saved can be represented by the reduction of $|C|$. Recall that bigger values of L correspond to the decrease in size of C.

The Figure 2 is based on the Flickr image data set. The x-axis presents the memory consumption of hash tables needed to index all the points in the dataset for diverse settings of k and L; such a space cost is computed from formula $12nL$ as discussed above, where $n = 55,000$. Since SimPair LSH uses the indices of LSH, this memory utilization is common for both algorithms. The y-axis shows the percentage gain in time. The numbers on top of the bars show the extra memory cost required to reach the gain in time reported. It is important to note that the extra memory cost for SimPair LSH remains constant to $3.7MB$, the similar pair list size, for all the diverse setting of L, while LSH needs significantly more extra memory, with increasing L, to achieve similar response time as our method. We can conclude that LSH needs significantly more memory to achieve even less response time saving as SimPair LSH does. For example, to have a gain in running time of 17% when $L = 78$, LSH needs $38MB$ extra memory.

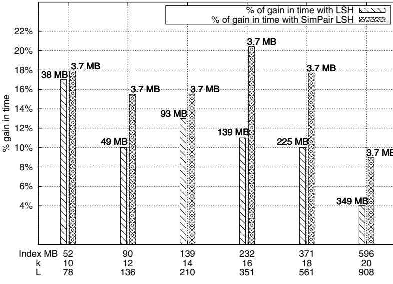


Fig. 2. Running time saved vs. LSH memory consumption

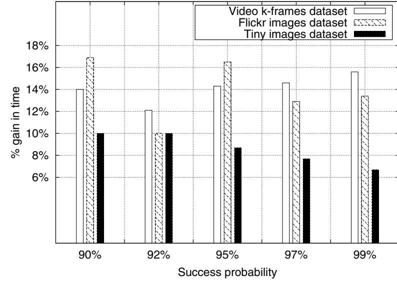


Fig. 3. Success probability P vs. running time

In contrast, SimPair LSH only needs $3.7MB$ storing the similar pair set to gain more than 17% response time; that makes our algorithm save 10 times the space cost used from LSH. Note that when the number of hash tables is increased, the time spent on computing the L hash values will also increase proportionally, which means the real running time saving is actually smaller than 17% for LSH. When L is large, even more extra memory is needed to gain the same amount of running time. For example, when the hash table size increased by around $225MB$, the time cost decreases only about 10%; in this case our method saves roughly 60 times the space cost used from LSH.

Comparing with the figure shown in the previous experiment, by using the same amount of extra memory ($3.7MB$), SimPair LSH gains slightly more percentage for different settings of k and L. Clearly, SimPair LSH is more space efficient in terms of saving running time.

Varying the success probability P. We varied P from 90% to 99%, and set $k = 12$ and L changes accordingly to see how P affects the real running time. Other parameters are the same as the previous experiment. The results are shown in Figure 3. From the figure again we can see that SimPair LSH outperforms LSH in terms of running time consistently. For different data sets, P has different impact on the saving time. However, the general trends seem to indicate that the impact is not significant.

Varying the dimensionality d. We ran experiments on Flickr image data set with different dimensionality d: 162 and 512, and set $P = 95\%$ to see how d affects real running time saved. Other parameters are the same as the previous experiment. The results are shown in Figure 1b. The y-axis shows the percentage of running time saved as in the previous experiments. From this figure we can see that for a higher dimensionality, the percentage of real time saved is higher in general. This is because the gain in time each prune brings is relatively higher compared with the cost of each prune when the dimensionality is higher.

In addition, we also ran experiments on the full 1 million tiny image data set as mentioned earlier, and the results were consist with what we have shown. Due to the space limit, please see details in the complete version of this paper [1].

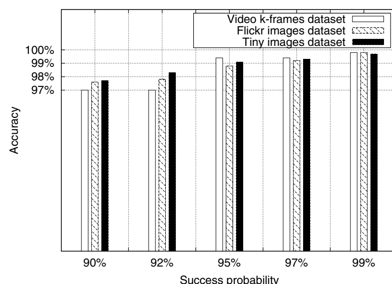


Fig. 4. Sucess probability θ vs. Recall (Quality of results)

4.5 Quality of Results

In this set of experiments, we tested the recall or false negatives of both methods. If a user set P to 90%, it means that he/she can tolerate missing at most 10% near neighbors. The results in Fig. 4 shows that the real recall value is clearly higher than the user specified probability.

5 Conclusions

In this paper, we study the problem of range search in an incremental manner based on a well-known technique, Locality Sensitive Hashing. We propose a new approach to improve the running time of LSH. The idea is take advantage of certain number of existing similar point pairs, and checking this similar pair set on-the-fly during query time. Since the look-up time cost is much cheaper than the distance computation, especially when the dimensionality is high, our SimPair LSH approach consistently outperforms the original LSH method, with the cost of a small amount of extra space. To gain the same amount of running time, LSH needs significantly more space than SimPair LSH (e.g. 10 to 100 times more). The superiority of SimPair LSH over the original LSH is confirmed by our thorough experiments conducted on 3 real-world image data sets. Furthermore, SimPair LSH preserves the theoretical guarantee on the recall of the search results. Last, SimPair LSH is easy to implement based on LSH.

References

1. Complete version of this paper can be found at, `https://www.l3s.de/web/upload/documents/1/SimSearch_complete.pdf`
2. Andoni, A., Indyk, P., Patrascu, M.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: FOCS, pp. 459–468 (2006)
3. Andoni, A., Indyk, P.: E^2 LSH0.1 User Manual. `http://web.mit.edu/andoni/www/LSH/manual.pdf` (2005)
4. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. CACM 51(1) (2008)

5. Bawa, M., Condie, T., Ganesan, P.: Lsh forest: self-tuning indexes for similarity search. In: WWW, pp. 651–660 (2005)
6. Bentley, J.L.: Multidimensional binary search trees used for associative searching. CACM 18(9) (1975)
7. Berchtold, S., Böhm, C., Jagadish, H.V., Kriegel, H.-P., Sander, J.: Independent quantization: An index compression technique for high-dimensional data spaces. In: ICDE, pp. 577–588 (2000)
8. Beygelzimer, A., Kakade, S., Langford, J.: Cover trees for nearest neighbor. In: ICML, pp. 97–104 (2006)
9. Chum, O., Philbin, J., Isard, M., Zisserman, A.: Scalable near identical image and shot detection. In: CIVR, pp. 549–556 (2007)
10. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: SCG, pp. 253–262 (2004)
11. Foo, J.J., Sinha, R., Zobel, J.: Discovery of image versions in large collections. In: Cham, T.-J., Cai, J., Dorai, C., Rajan, D., Chua, T.-S., Chia, L.-T. (eds.) MMM 2007. LNCS, vol. 4352, pp. 433–442. Springer, Heidelberg (2006)
12. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB, pp. 518–529 (1999)
13. Gonzalez, R.C., Woods, R.E.: Digital Image Processing, 3rd edn. Prentice Hall, Englewood Cliffs (2007)
14. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD (1984)
15. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
16. Katayama, N., Satoh, S.: The sr-tree: An index structure for high-dimensional nearest neighbor queries. In: SIGMOD (1997)
17. Ke, Y., Sukthankar, R., Huston, L.: An efficient parts-based near-duplicate and sub-image retrieval system. In: ACM Multimedia, pp. 869–876 (2004)
18. Koudas, N., Ooi, B.C., Shen, H.T., Tung, A.K.H.: Ldc: Enabling search by partial distance in a hyper-dimensional space. In: ICDE, pp. 6–17 (2004)
19. Krauthgamer, R., Lee, J.R.: Navigating nets: simple algorithms for proximity search. In: SODA, pp. 798–807 (2004)
20. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In: VLDB, pp. 950–961 (2007)
21. Panigrahy, R.: Entropy based nearest neighbor search in high dimensions. In: SODA, pp. 1186–1195 (2006)
22. Sakurai, Y., Yoshikawa, M., Uemura, S., Kojima, H.: The a-tree: An index structure for high-dimensional spaces using relative approximation. In: VLDB, pp. 516–526 (2000)
23. Samet, H.: Foundations of Multidimensional and Metric Data Structures, August 8, 2006. Morgan Kaufmann, San Francisco (2006)
24. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: VLDB, pp. 194–205 (1998)
25. Yu, C., Ooi, B.C., Tan, K.-L., Jagadish, H.V.: Indexing the distance: An efficient method to knn processing. In: VLDB (2001)