Predicting Pair Similarities for Near-Duplicate Detection in High Dimensional Spaces

Marco Fisichella, Andrea Ceroni, Fan Deng, and Wolfgang Nejdl

L3S Research Center, Hannover, Germany
{fisichella,ceroni,deng,nejdl}@L3S.de

Abstract. The problem of near–duplicate detection consists in finding those elements within a data set which are closest to a new input element, according to a given distance function and a given closeness threshold. Solving such problem for high–dimensional data sets is computationally expensive, since the amount of computation required to assess the similarity between any two elements increases with the number of dimensions. As a motivating example, an image or video sharing website would take advantage of detecting near–duplicates whenever new multimedia content is uploaded. Among different approaches, near–duplicate detection in high–dimensional data sets has been effectively addressed by SimPair LSH [11]. Built on top of Locality Sensitive Hashing (LSH), SimPair LSH computes and stores a small set of near-duplicate pairs in advance, and uses them to prune the candidate set generated by LSH for a given new element. In this paper, we develop an algorithm to predict a lower bound of the number of elements pruned by SimPair LSH from the candidate set generated by LSH. Since the computational overhead introduced by SimPair LSH to compute near-duplicate pairs in advance is rewarded by the possibility of using that information to prune the candidate set, predicting the number of pruned points would be crucial. The pruning prediction has been evaluated through experiments over three real–world data sets. We also performed further experiments on SimPair LSH, confirming that it consistently outperforms LSH with respect to memory space and running time.

Keywords: Indexing methods, Query, Near–duplicate detection, Locality Sensitive Hashing, high–dimensional data sets.

1 Introduction

Near–duplicate detection, also known as similarity search, is an old and well–established research topic which finds applications in different areas, from information retrieval to pattern recognition, from multimedia databases to machine learning. The near–duplicate detection problem can be informally stated as follows: given an object and a collection, find those elements in the collection which are most similar to the input one. Objects are usually represented through a set of features, which are exploited to assess similarities among them. Note that a similarity metric has to be defined over the feature space in order to evaluate

H. Decker et al. (Eds.): DEXA 2014, Part II, LNCS 8645, pp. 59–73, 2014.

similarities. Depending on the application domain, objects can be, for instance, documents, images or videos. However, once features have been extracted, they are represented just as points in a multidimensional space, independently from their original nature. In the rest of this paper we will refer to *objects* and *features* respectively as *points* and *dimensions*.

The near–duplicate detection problem, although it has been studied for decades, is still considered a challenging problem. The main reason is the well known *curse of dimensionality* [6]. It has been shown that the computational complexity of the near–duplicate detection exponentially increases with the number of point dimensions [1]. Moreover, any partitioning and clustering based indexing approach degenerates to a linear scan approach when the problem dimensionality is sufficiently high, i.e. more than 10 dimensions. This fact was both theoretically and empirically proved in [23].

In this work, we study near–duplicate detection within the context of *incremental range searches*. Given a point as a query, the problem consists in retrieving all the similar points from the data set before processing the query. We further introduce the term *incremental* to specify that in this scenario the data set is built in an incremental manner: whenever a point has to be inserted in the data set, the similar points previously inserted are retrieved and taken into account before actually performing the insertion. If a similar–enough point is already present in the collection, then the insertion will not be performed. Clearly, such a decision is taken by considering a pre–defined similarity threshold which allows to determine whether two points are similar or not. In addition to the previously mentioned *curse of dimensionality*, which equally affects any instance of the near–duplicate detection problem, a key issue of incremental range searches is fast query response. This is so because the near–duplicate detection is performed online during the insertion process, which has to be completed within the lowest possible amount of time.

Among others, one effective approach to near-duplicate detection in high-dimensional data sets is SimPair LSH [11]. Working on top of LSH, SimPair LSH substantially speeds up the running time of LSH by exploiting a small, constant amount of extra space. The key intuition consists in computing and storing a small set of near–duplicate pairs in advance, using them to prune the candidate set generated by LSH for a given new element. In this paper, we present an algorithm to predict a lower bound of the number of elements pruned by SimPair LSH from the candidate set. This would allow to predict whether the computational overhead introduced by SimPair LSH (see Paragraph 3.4 in [11]) is rewarded by the number of elements pruned. This and other contributions of this paper are summarized as follows:

- We present and evaluate an algorithm to predict a lower bound of the number of elements pruned from the candidate set by SimPair LSH.
- We describe an efficient procedure to maintain and update similar pair information in dynamic scenarios, i.e. when data sets are incrementally built with bursty sequences of queries.

– We perform additional experiments, on 3 real-world data sets, confirming that SimPair LSH consistently outperforms LSH under different aspects.

The remainder of this paper is the following. We first describe related works in Section 2 and we recall the basic behavior of SimPair LSH in section 3. We then present an algorithm to predict the number of points pruned by SimPair LSH from the candidate set in Section 4. In Section 6, we show that SimPair LSH outperforms LSH via extensive experiments. Finally, we give overall comments and conclusions in Section 7.

2 Related Work

As already stated, this paper is a substantial extension of the approach presented in [11], whose description is recalled in Section 3. In the rest of this section we give an overview of the current state of the art related to Locality Sensitive Hashing and near–duplicate detection.

2.1 Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) [16,13] was proposed by Indyk and Motwani and finds applications in different areas including multimedia near–duplicate detection (e.g., [7,21]). LSH was first applied in indexing high–dimensional points for Hamming distance [13], and later extended to L_p distance [9] where L_2 is Euclidean distance, the one used in this paper.

LSH exploits certain hash functions to map each multi–dimensional point into a scalar; the employed hash functions 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: (i) select k hash functions h randomly and uniformly from a LSH hash function family H, and create L hash tables, hereafter called *buckets*; (ii) create an index (a hash table) by hashing all points $\boldsymbol{p}$ in the data set P into different buckets based on their hash values; (iii) when a query point $\boldsymbol{q}$ arrives, use the same set of hash functions to map $\boldsymbol{q}$ into L buckets, one from each hash table; (iv) retrieve all points $\boldsymbol{p}$ from the L buckets, collect them into a candidate set C, and remove duplicate points in C; (v) for each point $\boldsymbol{p}$ in C compute its distance to $\boldsymbol{q}$ and output those points that are similar to $\boldsymbol{q}$.

In LSH, the probability that two points p_1 and p_2 are hashed into the same bucket is proportional to their distance c, and it can be computed as follows:

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

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, namely success probability, under H:

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

2.2 Other LSH-Based Approaches

Since it was proposed, LSH has been extended in different directions. Lv et al. [20] proposed multi-probe LSH, showing experimentally that it significantly reduces space cost while achieving the same search quality and similar time efficiency compared with LSH. The key idea of multi-probe LSH is that the algorithm not only searches for the near–duplicates in the buckets to which the query point $\boldsymbol{q}$ is hashed, but it also searches the buckets where the near–duplicates have slightly less chance to appear. The benefit 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.

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 [3] 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 [1].

Inspired by LSH technique is the distributed similarity search and range query processing in high dimensional data [15]. Authors consider mappings from the multi-dimensional LSH bucket space to the linearly ordered set of peers that jointly maintain the indexed data and derive requirements to achieve high quality search results and limit the number of network accesses. Locality preserving properties of proposed mappings is proved.

In [1], authors observe that despite decades of research, current solutions still suffer from the "curse of dimensionality", i.e. either space or query time exponential effort in the dimensionality d is needed to guarantee an accurate result. In fact, for a large enough dimensionality, current solutions provide little improvement over a brute force linear scanning of the entire data set, both in theory and in practice. To overcome this limitation, in one of the latest work Indyk et al. [2] lead to a two-level hashing algorithm. The outer hash table partitions the data sets into buckets of bounded diameter. Then, for each bucket, they build the inner hash table, which uses the center of the minimum enclosing ball of the points in the bucket as a center point. By this approach, authors claim to achieve a better query time and space consumption with respect to their previous work [16]. In a recent research conducted in [8], authors proposed two new algorithms to speed up LSH for the Euclidean distance. These algorithms are able to reduce the hash index construction time.

Finally, it is important to note that SimPair LSH is orthogonal to LSH and to other LSH variants described above and, more significant, it can be applied in those scenarios, taking advantage of the improvements achieved in those works.

2.3 Similarity Search and Duplicate Detection on Streaming Data

The applications we consider have certain data stream characteristics such as continuous queries and fast response requirements, although they are still mainly

traditional Web applications. Within the past decade, data streaming processing has been a popular topic where the applications include sensor data processing, real-time financial data analysis, Internet traffic monitoring and so on.

Gao et al. [12] and Lian et al. [19] studied the problem of efficiently finding similar time series on streaming data, and they achieved efficiency by accurately predicting future data. Their methods are for time series data and cannot be used for the type of applications we consider. Koudas et al. [18] studied the problem of finding k nearest neighbors over streaming data, but they were concerned about low-dimensional case. Deng and Rafiei [10] studied the problem of detecting duplicates for streaming data, where no similarity is involved.

Recently, Sudaram et al. [4] presented the Parallel LSH (PLSH) workflow, a system to handle near–duplicate queries on large amounts of text data, over one Billion Tweets, based on an efficient parallel LSH implementation.

3 SimPair LSH

In this section we recall the behavior of SimPair LSH, since it is used as a starting point in the rest of the paper. Hereafter, unless noted otherwise, the term *LSH* denotes the original LSH indexing method.

3.1 Problem Statement (Incremental Range Search)

Near–duplicate. Given a point $\boldsymbol{q} \in \mathbb{R}^d$, any point $\boldsymbol{p} \in \mathbb{R}^d$ such that $d(\boldsymbol{q}, \boldsymbol{p}) < \tau$ is a near–duplicate of $\boldsymbol{q}$, where $\tau \in \mathbb{R}$ is a similarity threshold. Among the available distance functions d which might be exploited, the Euclidean distance has been chosen in this work, since it has been widely used in different applications. However, SimPair LSH can be easily extended to other distance functions, e.g. L1 and Hamming distance, since LSH can be applied in those cases as well.
Incremental Range Search. Given a point $\boldsymbol{q} \in \mathbb{R}^d$ and a set P of n points $\boldsymbol{p}_i \in \mathbb{R}^d$, find all the near–duplicate of $\boldsymbol{q}$ in P according to a given distance function $d(\cdot, \cdot)$ and a given similarity threshold τ before $\boldsymbol{q}$ is inserted into P.

3.2 The SimPair LSH Algorithm

The main intuition behind SimPair LSH resides in how LSH works. Given a query point $\boldsymbol{q}$ and a set of points $\boldsymbol{p}$, LSH fills a candidate set C for $\boldsymbol{q}$ with *all* the points $\boldsymbol{p}$ stored in the same buckets in which $\boldsymbol{q}$ has been hashed. The near–duplicates of $\boldsymbol{q}$ are then found by comparing it with all the points in C. Such approach suffers from two facts. First, in order to increase the probability of including in C all the near–duplicates of $\boldsymbol{q}$, a large number of hash tables has to be created. This can lead to an higher number of points in C, which means an higher number of comparisons. Second, in case of high-dimensional points, the number of operations required to compare two points increases. SimPair LSH speeds up the search by pre-computing and storing in memory a certain number

of pair-wise similar points in the data set. They are exploited at query time to prune the candidate set C, which results in a lower number of comparisons.

Formally, SimPair LSH algorithm works as follows. Given a set P of n points $\boldsymbol{p}_i \in \mathbb{R}^d$, SimPair LSH creates L buckets as in LSH. In addition, given a similarity threshold θ, the set SP of similar pairs $(\boldsymbol{p}_1, \boldsymbol{p}_2) : d(\boldsymbol{p}_1, \boldsymbol{p}_2) < \theta$ is built and stored along with the computed distances $d(\boldsymbol{p}_1, \boldsymbol{p}_2)$. Whenever a query point $\boldsymbol{q} \in \mathbb{R}^d$ comes, SimPair LSH retrieves all points in the buckets to which $\boldsymbol{q}$ is hashed. Let this set of points be the candidate set C. Instead of linearly scanning through all the points $\boldsymbol{p}$ in C and computing their distances to $\boldsymbol{q}$ as in LSH, SimPair LSH checks the pre-computed similar pair set SP whenever a distance computation $d(\boldsymbol{q}, \boldsymbol{p})$ is done. Depending on the value of $d(\boldsymbol{q}, \boldsymbol{p})$ and on the value of a given similarity threshold τ, SimPair LSH can behave in 2 different ways:

- If $d(\boldsymbol{q}, \boldsymbol{p}) <= \tau$, then SimPair LSH searches in SP for all points $\boldsymbol{p\prime}$: $d(\boldsymbol{p}, \boldsymbol{p\prime}) \leq \tau - d(\boldsymbol{q}, \boldsymbol{p})$. It checks if $\boldsymbol{p\prime}$ is in the candidate set C or not: if yes, then it marks $\boldsymbol{p\prime}$ as a near–duplicate of $\boldsymbol{q}$ without computing the distance $d(\boldsymbol{p\prime}, \boldsymbol{q})$.
- If $d(\boldsymbol{q}, \boldsymbol{p}) > \tau$, SimPair LSH searches in SP for all those points $\boldsymbol{p\prime} : d(\boldsymbol{p}, \boldsymbol{p\prime}) < d(\boldsymbol{q}, \boldsymbol{p}) - \tau$. It checks if $\boldsymbol{p\prime}$ is in the candidate set C or not. If yes, then it removes $\boldsymbol{p\prime}$ from C without the distance computation.

For a more detailed description of SimPair LSH, please refer to [11].

4 Pruning Prediction

In this Section we propose an algorithm to predict a lower bound of the number of elements pruned by SimPair LSH with respect to LSH. As we saw in Section 3, SimPair LSH requires the maintenance and the access to SP. That is translated in a computational overhead which might be compensated by pruning elements in the candidate set. Finally, predicting the prunes in advance gives us the possibility to know if the overhead is rewarded.

4.1 The Intuition

According to the pruning analysis presented in [11], a lower bound of the number of prunes given a query $\boldsymbol{q}$ can be estimated. The key intuition is as follows: take a few sample points $\boldsymbol{p}$ from C and for each $\boldsymbol{p}$ compute $d(\boldsymbol{q}, \boldsymbol{p})$. Based on the sample, estimate the distribution of different $d(\boldsymbol{q}, \boldsymbol{p})$ for all $\boldsymbol{p}$ in C. From $d(\boldsymbol{q}, \boldsymbol{p})$ we can derive an upper bound of $d(\boldsymbol{q}, \boldsymbol{p\prime})$ according to the triangle inequality. Thus, we can estimate a lower bound of the probability that $\boldsymbol{p\prime}$ appears in C and, accordingly, a lower bound of the number of prunes.

Again, by using the small fraction of sample points obtained from C, SimPair LSH can check SP and find if there is any *close enough* point $\boldsymbol{p\prime}$ of $\boldsymbol{p}$ such that $d(\boldsymbol{p\prime}, \boldsymbol{p}) < |d(\boldsymbol{q}, \boldsymbol{p}) - \tau|$. Since the distance $d(\boldsymbol{q}, \boldsymbol{p})$ is known, an upper bound of $d(\boldsymbol{q}, \boldsymbol{p\prime})$ can be derived according to the triangle inequality. Knowing $d(\boldsymbol{q}, \boldsymbol{p\prime})$, one can know the probability that $\boldsymbol{p\prime}$ appears in C, which leads to a prune.

4.2 The Pruning Prediction Algorithm

The algorithm for predicting a lower bound of the number of prunes is described in Algorithm 1. First, let us consider the probability that two points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are hashed into the same bucket. Such a probability, according to Equation 1, is proportional to their distance $d(\boldsymbol{p}_1, \boldsymbol{p}_2)$, which will be referred to as simply d in the rest of the section. Second, in our algorithm, the full distance range is cut into multiple intervals. Then, given a distance value d, the function $I(d)$ can be used to determine which interval d falls in. Counter $Count[]$ is a histogram for storing the number of points in a particular interval. The interval is determined by the collision probabilities of pair–wise distances. For a fixed parameter r in Equation 1 the probability of collision $P(d)$ decreases monotonically with $d = ||\boldsymbol{p}_1\boldsymbol{p}_2||_p$, where p is 2 in our case, where we consider *Gaussian* distribution. The optimal value for r depends on the data set and the query point. However, in [9] was suggested that $r = 4$ provides good results and, therefore, we currently use the value $r = 4$ in our implementation.

Under this hash function setting, there is not much difference for distances d within the range $[0, 1]$ in terms of hash collision probability. Thus, we use fewer intervals. In contrast, for distances within the range $[1, 2.5]$, the hash collision probabilities differ significantly. We use more intervals for this distance range.

The policy that we use to split a distance range into intervals is defined as follows. Fix the number of intervals first (e.g., 100); assign one interval for range $[0, x_1]$ and one for range $[x_2, \infty]$, within both of which the hash collision probabilities are similar; assign the rest of intervals to range $[x_1, x_2]$ by cutting the range evenly. Since the function $I(d)$ is only determined by the hash function, the intervals cutting can be done off–line before processing the data set.

4.3 Sampling Accuracy

SimPair LSH only takes a small fraction (e.g., 10%) of points from C. Increasing the number of points in the sample whose distances to $\boldsymbol{q}$ are within an interval and estimating the value in C will generate some errors.

Lemma 1. *A uniform random sample gives an unbiased estimate for the number of points with certain property, and the relative accuracy is inversely proportional to the number of points, and proportional to the sample size and to the true number being estimated. The standard deviation of the ratio between the estimate and the true value is* $\sqrt{\frac{n}{R}(\frac{1}{x} - \frac{1}{n})}$*, where* n *is the total number of points to be sampled,* R *is the sample size and* x *is the true value to be estimated, i.e. the number of points with the property.*

Proof. Let us assume that R different points are randomly taken into the sample, and each point with the property has a probability $\frac{x}{n}$ to stay in the sample. Let X_i be an indicator random variable indicating if the i-th point being sampled is with the property or not. That is,

$$X_i = \begin{cases} 1, & \text{if the sampled point has the property} \\ 0, & \text{otherwise} \end{cases}$$

Algorithm 1: Prediction of the number of prunes.

Input: A set C with n d-dimensional points $\boldsymbol{p}$; a distance function $d(\cdot,\cdot)$, a distance threshold τ defining near–duplicates; a query point $\boldsymbol{q}$; a distance interval function $I(d)$; the set of similar point pairs SP.
Output: Number of prunes $PruneNumber$
begin
  Construct a sample set S by selecting s points from C uniformly at random;
  for *each point* $\boldsymbol{p}$ *in* S **do**
    Compute the distance $d(\boldsymbol{q},\boldsymbol{p})$;
    search for $\boldsymbol{p'}$ in SP where $d(\boldsymbol{p'},\boldsymbol{p}) < |d(\boldsymbol{q},\boldsymbol{p}) - \tau|$;
    for *each* $\boldsymbol{p'}$ *found* **do**
      increment the counter $Count[I(d(\boldsymbol{q},\boldsymbol{p}) + |d(\boldsymbol{q},\boldsymbol{p}) - \tau|)]$ by 1;
  Scale up the non-zero elements in the counter by a factor of $|C|/|S|$ and store them back to Count[]; PruneNumber = 0;
  for *each non-zero element in Count[i]* **do**
    Let d_i be the maximum distance of interavl i;
    Let $P(d_i)$ be the probability that 2 points with distance d_i are hashed to the same value, according to Equation 1;
    $PruneNumber \mathrel{+}= Count[i] \cdot P(d_i)$;
  Output PruneNumber;

It can be easily shown that $Pr(X_i = 1) = \frac{x}{n}$ and $Pr(X_i = 0) = 1 - \frac{x}{n}$. Given the observed value $Y = \frac{n}{R}\sum_{i=1}^{R} X_i$, since $E[Y] = \frac{n}{R}\sum_{i=1}^{R} E[X_i] = x$ then it is possible to deduce that Y is an unbiased estimate. Thus, we can conclude that:

$$VAR[Y] = \frac{n^2}{R^2} VAR\left[\sum_{i=1}^{R} X_i\right] = \frac{n}{R} x \left(1 - \frac{x}{n}\right)$$

□

5 Similar Pair Maintenance and Updating

Since the gain from SimPair LSH lies on the fact that similar pair information (i.e. SP) is stored in memory, it is important to maintain it properly and efficiently. Moreover, it is crucial to handle sequences of queries by efficiently updating the similarity pair information. In case data are static and not created by the incremental process, one can build the similar pair set SP offline, before queries arrive, by using existing similarity join algorithms (e.g., [17]). Since it is out of the scope of this work, we will not discuss this in detail.

5.1 Maintenance

Data Structure. The set SP can be implemented as a two dimensional linked list. The first dimension is a list of points; the near–duplicates of each point

$\boldsymbol{q}$ in the first-dimension list are stored in another linked list (i.e. the second dimension), ordered by the distances to $\boldsymbol{q}$. Then, a hash index is built on top of the first-dimension linked list to speed up the look-up operations.

Bounding the Size of SP. Since the total number of all similar pairs for a dataset of n objects can be $O(n^2)$, an underlying issue is how to restrict the size of SP. To achieve this purpose we set θ (the similarity threshold for the similar point pairs stored in SP) to τ (the similarity threshold for the similarity search). However, in case the data set size is large, SP can be too big to fit in memory. Thus we impose a second bound on the size of SP: it must not be greater than a constant fraction of the index size (e.g., 10%). To satisfy the latter constraint we can reduce the value of θ within the range $(0, \tau]$. Clearly, a bigger value of θ will generate a larger set of SP increasing the chances of finding $\boldsymbol{p'}$ of $\boldsymbol{p}$ in C, and thus triggering a prune. Another possible solution consists in removing a certain number of similar pairs having the largest distances when the size of SP is above the space bound. We first estimate the number of similar pairs, called k pairs, to be removed based on the space to be released. The k pairs to be removed should be those whose distances are the largest, since they have less chance to generate a prune. Thus, we should find the top–k pairs with the largest distances. To obtain such top–k list, we can create a sorted linked list with k entries, namely L_k; we then take the part with the largest distances of the first second-dimension linked list in SP and fill L_k (assuming L_k is shorter than the first second-dimension linked list; if not, we go to the next second-dimension linked list). Then, we scan the second second-dimension linked list and update L_k. After scanning the second-dimension linked lists, the top–k pairs with the largest distances are found, and we can remove them from SP. Note that real–time updates are not necessary for these operations, and they can be buffered and executed when the data arrival rate is slower.

5.2 Updating

Point Insertions. In a continuous query scenario, each point $\boldsymbol{q}$ issues a query before inserted into the database. That is, as an application requirement, all near–duplicates of each newly arrived point need to be found before the new point updates SP. Hence, to maintain the similar pair list, we only needs to add the near–duplicates of $\boldsymbol{q}$ just found into SP and there is no similarity search involved. To add near–duplicates of $\boldsymbol{q}$ into SP, we first sort the near–duplicates based on their distances to $\boldsymbol{q}$, and store them in a linked list. We then insert $\boldsymbol{q}$ and the linked list into SP, and we update the hash index of SP in the meanwhile. Besides, we need to search for each near–duplicate of $\boldsymbol{q}$ and insert $\boldsymbol{q}$ into the corresponding linked lists of its near–duplicates. Within the linked list into which $\boldsymbol{q}$ is to be inserted, we use a linear search to find the right place $\boldsymbol{q}$ should be put based on the distance between $\boldsymbol{q}$ and the near–duplicate. Note that we could have built indices for the second-dimension linked lists, but this would have increased the space cost. Also, real–time updates of SP are not necessary unlike the query response; we can buffer the new similar pairs and insert them into SP when the data arrival rate is lower.

Point Deletions. When a point $\boldsymbol{q}$ has to be deleted from the data set, we need to update SP. First, we search for all the near–duplicates of $\boldsymbol{q}$ in SP and remove $\boldsymbol{q}$ from each of the second-dimension linked lists of the near–duplicates. We then need to remove the second dimension linked list of $\boldsymbol{q}$ from SP.

Buffering SP Updates. As a matter of fact, the nature of the data arrival rate is bursty, i.e. there can be a large number of queries at one time and very few queries at another time. Then, as mentioned earlier, the updates to SP (inserting a point and reducing size) can be buffered and processed later in case the processor is overloaded because of bursts of the data arrival rate. This buffering mechanism guarantees the real–time response to the similarity search query. Note that the operation of deleting a point cannot be buffered since this can lead to inconsistent results.

6 Experiments and Evaluations

We evaluated our work on different real–world data sets and under different evaluation criteria. In the following sections we describe the data sets, as well as the criteria and results of our evaluation. Note that we only report experiments that have not been performed in [11].

The 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. Both the data points and the LSH indices were loaded into the main memory. Each index entry for a point takes 12 bytes memory. The source code of the original LSH was obtained from E2LSH[1] and was used without any modification.

6.1 Data Sets

We experiment the effectiveness of SimPair LSH on the same three real–world image data sets used in [11], which are hereafter summarized.

Flickr Images. We sent 26 random queries to Flickr and we retrieved all the images within the result set. After removing all the results 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 [22]. Due to the high memory cost of LSH for large data sets, we randomly sampled $50,000$ images from the entire dataset. This allowed us to vary the number of hash tables within a larger range. The random sampling operation also reduced the chance that similar pairs appear in the data set, since the images retrieved from the result set of a query have higher chance to be similar to each other.

Video Key-Frames. We sent 10 random queries to Youtube and obtained around 200 video clips from each result set. We then extracted key frames of the

[1] `http://web.mit.edu/andoni/www/LSH/manual.pdf`

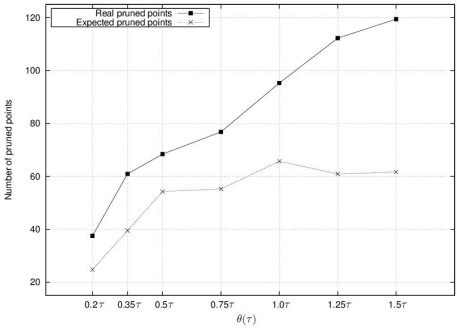


Fig. 1. Predicted prunes vs. $\theta(\tau)$

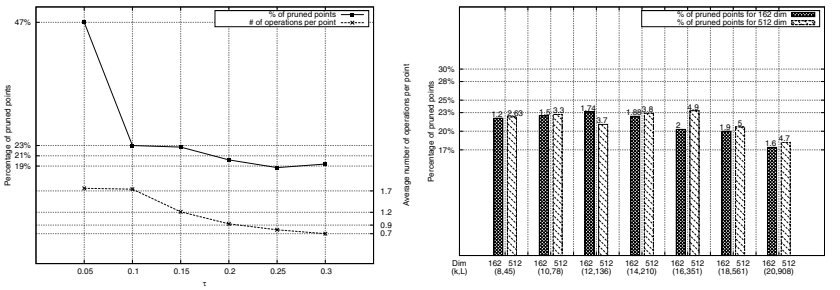
videos in the following way. Sequentially scan HSV histograms (whose dimensionality is equal to 162) of frames in a video: if the euclidean distance between two consecutive histograms is above 0.1, then keep the second histogram; otherwise skip it. In the end, we obtained 165,000 key–frame images. The choice of setting the distance threshold to 0.1 was driven by an empirical tuning. After an human evaluation, the distance value of 0.1 resulted to be the boundary line between equal and different images: image pairs having distance greater than 0.1 were mostly perceived as different, while those having distance lower than 0.1 where mostly categorized as identical. In other words, starting from a distance equal to 0.1, a human can see that two images are similar but not identical.

For all the image data sets described above, we removed duplicates and converted each image within a data set into a d-dimensional vector ($d \in \{162, 512\}$) by using the standard HSV histogram methods [14]. Each element of the vector represents the percentage of pixels within a given HSV interval.

6.2 Results

In this Section we report and discuss the results of our evaluation. In order to validate our approach, we randomly selected 100 objects from the data set as query objects. We tried other values for the number of queries, from 100 to 1000, without noting any change in performances. Thus, we kept it to 100 in the following experiments. The results presented in the rest of this section represent values averaged over the query objects.

Unless explicitly stated, the results that we report in the rest of this section were achieved on the Flickr data set, and the parameters were set as follows: distance threshold for near–duplicates $\tau = 0.1$, dimensionality of the data set $d = 162$, distance threshold used for filling the similar pair list SP $\theta = 0.1$, number of buckets $L = 136$, and success probability $\lambda = 95\%$. For the other data sets we achieved similar results.



(a) Number of prunes and costs vs. τ (b) Number of prunes and costs vs. d

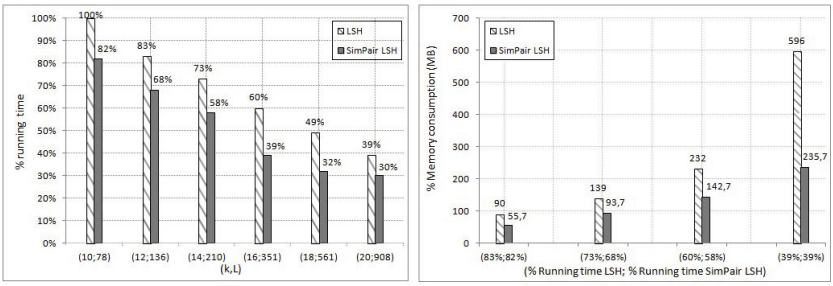
Fig. 2. Number of prunes with different values of τ and d

Pruning Prediction. We test our pruning prediction algorithm (Section 4) by predicting the number of prunes with different values of similar pair threshold θ. The parameter settings are as follows: distance threshold for near–duplicates $\tau = 0.1$, the dimensionality of the data set $d = 162$, $k = 20$, $L = 595$, and the success probability $\lambda = 90\%$. The predicted lower bound of pruned points with different θ is shown in Figure 1, where the *y-axis* shows the average number of points pruned per query, and the *x-axis* shows the values of θ based on τ.

It is possible to observe that the pruning prediction always underestimates the actual number of pruned points, providing a reliable lower bound. Moreover, the difference between lower bound and real number of pruned points is almost constant (between 20%-30%) for low values of θ, i.e. $\theta < \tau$. This means that the trend of the prediction is similar to the real profile of pruned points: apart from a scale value (i.e. the difference between prediction and real number), the actual number of pruned points can be estimated quite precisely. These facts do not hold for higher values of θ ($\theta > \tau$), where the difference between lower bound prediction and real number of pruned points is more varying. This analysis motivates our choice of setting $\theta = \tau$ in the rest of the experiments.

Number and Cost of Prunes. We computed the number of distance computations saved by SimPair LSH, along with the corresponding time and space costs, by varying different parameters that were not considered in [11].

We varied the value of τ to see how it affects the pruning and costs. The other parameters have the values specified at the beginning of the section. The results are shown in Figure 2a, where the *y–axises* have the same meaning as in the previous figures and the *x–axis* represents the values of τ. It is possible to observe from the figure that τ has no significant impact on the pruning effectiveness. The high percentage of pruned points when $\tau = 0.05$ is because the size of the candidate set is small, and the variance of the percentage is higher. As previously mentioned, we fixed $\tau = 0.1$ in the rest of the experiments.



(a) % Running time vs. LSH parameters (k, L)

(b) Memory consumption (MB) vs. % running time for LSH and SimPair LSH

Fig. 3. % Running time (based on LSH greatest time) and memory consumption

We experimented two different vector dimensionalities d, 162 and 512, to investigate the influence of d in the pruning and its costs. The other parameters are the same as in the previous experiments. The results are shown in Figure 2b, where the left *y–axis* shows the percentage of pruned points as in the previous experiments. The numbers on top of the bars show the average operation cost per point. From the figure, it is possible to conclude that the operation cost is larger when the dimensionality is higher. This is partially because the number of similar pairs in SP is relatively larger when d is large, which has a similar effect as increasing θ (when d is large, there are more similar pairs although the similarity threshold does not change).

Time-Space Joint Analysis. The gain in time that SimPair LSH achieved on different data sets has been already showed in [11]. However, LSH can save running time as well by increasing the number of hash tables, i.e. increasing the k and L parameters. In this section we want to show that, to achieve the same gain in time, the additional space required by LSH for higher values of k and L is greater than the space required by SimPair LSH. We show this fact by reporting experiments done on the Flickr image data set, having $n = 55,000$.

The amount of memory required by LSH can be derived from L. Since each hash table stores the identifiers of the n points in P, each one occupying 12 bytes (as implemented in E2LSH), the LSH space cost is equal to $12nL$. Thus, in order to compute the extra space needed by LSH to achieve roughly the same improvement in running time of SimPair LSH, it is sufficient to check the value of L. Since the size of C dominates the time required by LSH to scan through the candidate set, the running time being saved can be represented as the reduction of $|C|$. Recall that bigger values of L correspond to the decrease in size of C. Regarding SimPair LSH, it requires additional space for keeping the set of similar point pairs SP, besides the memory for storing the LSH indices. As already specified, we restricted it to be at most a constant fraction of the LSH indices (10% in our experiments).

Figure 3a shows the running time of both LSH and SimPair LSH for different values of k and L. The running times are showed as percentage values with respect to the running time required by LSH with parameters $(10; 78)$. As already showed in [11], SimPair LSH always outperforms LSH, which anyway achieves decreasing running time for increasing values of k and L. In particular, there always exists in principle a choice of k and L which allows LSH to achieve a running time similar to the one taken by SimPair LSH for lower values of k and L. For instance, the running time of SimPair LSH with parameters $(10; 78)$ is 82%, while a similar value (83%) can be obtained by LSH with parameters $(12; 136)$. Since increasing the parameter values leads to an increase in the required space, it is definitely worth comparing the memory consumption of LSH and SimPair LSH for achieving almost the same running time. This is shown in Figure 3b. Different running times of both LSH and SimPair LSH are grouped in couples along the x–axis, while the corresponding memory costs to achieve them are plotted along the y–axis. The running time values are taken from Figure 3a and they are coupled so that similar values of different approaches belong to the same couple. As an example, the running time equal to 82% in the $(83\%, 82\%)$ couple is the one achieved by SimPair LSH with parameters (10;78), while 83% is the running time achieved by LSH with parameters (12;136). Figure 3b shows that, for every running time couple, LSH always leads to an higher memory consumption (caused by higher values of k and L). For instance, SimPair LSH requires 142.7 MB to achieve a running time equal to 58%, while LSH needs 232 MB to reach a similar running time (60%). This fact is accentuate in the (39%,39%) couple, where the space required by SimPair LSH is the 39.8% of the one taken by LSH.

7 Conclusions

In this paper, we presented and evaluated an algorithm to predict a lower bound of the number of elements pruned by SimPair LSH. In our experiment, the pruning prediction algorithm always provides a reliable lower bound of the actual number of pruned points. Moreover, the difference between lower bound and real number of pruned points is almost constant for a sub set of setups, meaning that the profile of the prediction is similar to the real profile of pruned points apart from a scale value. Thanks to that, our algorithm allows to predict whether the computational overhead introduced by SimPair LSH is rewarded by the number of elements pruned.

Moreover, we described the procedure to maintain and update similar pair information, and we performed further experiments on SimPair LSH, confirming the robustness of SimPair LSH and its superiority with respect to LSH under different setups. In particular, in our evaluations we experimented that LSH can save running time by increasing the number of hash tables, i.e. increasing the k and L parameters. Since increasing the parameter values k and L leads to an increase in the required space, we compared the memory consumption of LSH and SimPair LSH for achieving almost the same running time. Our conclusion is that LSH requires significantly more space than SimPair LSH (e.g., from 40% to 60% more of memory consumption).

Acknowledgement. This work was partly funded by the DURAARK (GA No:600908) project under the FP7 pro- gramme of the European Commission.

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM 51(1) (2008)
2. Andoni, A., Indyk, P., Nguyen, H.L., Razenshteyn, I.: Beyond locality-sensitive hashing. CoRR, abs/1306.1547 (2013)
3. Andoni, A., Indyk, P., Patrascu, M.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: FOCS (2006)
4. Bahmani, B., Goel, A., Shinde, R.: Efficient distributed locality sensitive hashing. In: CIKM (2012)
5. Bawa, M., Condie, T., Ganesan, P.: Lsh forest: self-tuning indexes for similarity search. In: WWW (2005)
6. Bellman, R.E.: Adaptive control processes - A guided tour (1961)
7. Chum, O., Philbin, J., Isard, M., Zisserman, A.: Scalable near identical image and shot detection. In: CIVR (2007)
8. Dasgupta, A., et al.: Fast locality-sensitive hashing. In: KDD (2011)
9. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: SCG (2004)
10. Deng, F.: Approximately detecting duplicates for streaming data using stable bloom filters. In: SIGMOD (2006)
11. Fisichella, M., Deng, F., Nejdl, W.: Efficient incremental near duplicate detection based on locality sensitive hashing. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010, Part I. LNCS, vol. 6261, pp. 152–166. Springer, Heidelberg (2010)
12. Gao, L., Wang, X.S.: Continuous similarity-based queries on streaming time series. IEEE Trans. on Knowl. and Data Eng. 17(10) (2005)
13. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB (1999)
14. Gonzalez, R.C., Woods, R.E.: Digital Image Processing, 3rd edn. (2006)
15. Haghani, P., Michel, S., Aberer, K.: Distributed similarity search in high dimensions using locality sensitive hashing. In: EDBT (2009)
16. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC (1998)
17. Jacox, E.H., Samet, H.: Metric space similarity joins. ACM Trans. Database Syst. 33(2) (2008)
18. Koudas, N., Chin, B., Kian-lee, O., Zhang, T.R.: Approximate nn queries on streams with guaranteed error/performance bounds. In: VLDB (2004)
19. Lian, X., Chen, L.: Efficient similarity search over future stream time series. IEEE Trans. on Knowl. and Data Eng. 20(1) (2008)
20. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe lsh: efficient indexing for high-dimensional similarity search. In: VLDB (2007)
21. Teixeira, T., et al.: Scalable locality-sensitive hashing for similarity search in high-dimensional, large-scale multimedia datasets. CoRR, abs/1310.4136 (2013)
22. Torralba, A., Fergus, R., Freeman, W.: Tech. rep. mit-csail-tr-2007-024. Technical report, Massachusetts Institute of Technology (2007)
23. Weber, R., Schek, H.-J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: VLDB (1998)