PSLSH: An Index Structure for Efficient Execution of Set Queries in High-Dimensional Spaces*

Parth Nagarkar
New Mexico State University
nagarkar@nmsu.edu

K. Selçuk Candan
Arizona State University
candan@asu.edu

ABSTRACT

Efficient implementations of range and nearest neighbor queries are critical in many large multimedia applications. Locality Sensitive Hashing (LSH) is a popular technique for performing approximate searches in high-dimensional multimedia, such as image or sensory data. Often times, these multimedia data are represented as a collection of important spatio-temporal features which are extracted by using localized feature extraction algorithms. When a user wants to search for a given entity (object, event, or observation), individual similarity search queries, which collectively form a set query, need to be performed on the features that represent the particular search entity. Existing LSH techniques require that users provide an accuracy guarantee for each query in the set query, instead of an overall guarantee for the entire set query, which can lead to misses or wasteful work. We propose a novel index structure, *Point Set LSH* (PSLSH), which is able to execute a similarity search for a given set of search points in the high-dimensional space with a user-provided guarantee for the entire set query. Experimental evaluation shows significant gains in efficiency and accuracy trade-offs for executing set queries in high-dimensional spaces.

CCS CONCEPTS

• Information systems → Nearest-neighbor search;

KEYWORDS

Nearest-neighbor search, Locality Sensitive Hashing

ACM Reference Format:

Parth Nagarkar and K. Selçuk Candan. 2018. PSLSH: An Index Structure for Efficient Execution of Set Queries in High-Dimensional Spaces. In *The 27th ACM International Conference on Information and Knowledge Management (CIKM '18)*, October 22–26, 2018, Torino, Italy. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3269206.3271691>

1 INTRODUCTION

The similarity search problem in high dimensional spaces is a very well-known problem. Tree-based indexing structures (KD-tree, X-tree, SR-tree, etc.) have been shown to be effective for

low-dimensional spaces [1]. For moderate to high dimensions, tree-based index structures are often out-performed even by linear scans (a problem popularly known as the *curse of dimensionality*). One solution to address this problem for higher dimensions is to look for *approximate* results instead of exact results. Especially given that, in many applications, 100% accuracy is not needed, searches that return points that are *close enough* to the query point rather than the closest ones, are often times, more effective and much faster than solutions that attempt to find exact query results [2].

One of the most popular solutions for approximate searching is called *Locality Sensitive Hashing* (LSH), first proposed in [1]. The idea behind LSH is to map high dimensional data to lower dimensional representations, in such a way that searches are nevertheless reasonably accurate. In the lower dimensional space, which is obtained through *random projections*, data points are mapped to individual buckets based on a hash function. The intuition behind this method is that data points closer in the original space will be mapped to the same buckets with a higher probability in the lower dimensional space than dissimilar points. This may, however, lead to *misses* as well as *false positives*.

Locality Sensitive Hashing has been studied and improved upon extensively in the research community [2, 3, 5–12]. The seminal Locality Sensitive Hashing work [1] aimed to address the (r,c)-Near Neighbors problem: every point p that lies within a distance of r from query point q should be reported with a probability guarantee of at least $1 - \delta$ (where δ is a user-specified error probability), whereas points that lie beyond distance of $c \times r$, for some $c > 1$, from the query point q should have a very low likelihood of being included in the query result. Intuitively, the user provides a *success probability* that decides the balance between the accuracy of the results and query processing speed for a given radius. Higher the target success probability, higher is the accuracy for the results, but slower the query processing speed, and vice-versa. Since using more layers helps eliminate misses, this may increase the number of *candidate points* that need to be enumerated and potentially eliminated during post-processing. It is important to note that the parameters of the index structure need to be selected carefully to achieve the target accuracy for the given target radius.

1.1 Set Queries and Limitations of LSH

While most (if not all) index structures are designed to search for a given point in a vector space, not all data representations match objects to single points. In fact, most modern feature extraction methods map objects of interest to multiple feature points that collectively represent the object of interest, rather than representing the object using a single representative point. In the case of images, for example, such feature sets can be extracted using popular localized feature extraction algorithms such as SIFT [13] or SURF [14].

*This work is supported by NSF Grant #1318788, NSF Grant #1339835, NSF Grant #1610282, NSF Grant #1633381, and EUH2020 #690817.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM '18, October 22–26, 2018, Torino, Italy

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6014-2/18/10...\$15.00

<https://doi.org/10.1145/3269206.3271691>

Naturally, when objects of interest are represented using sets of feature points, similarity searches have to be performed on each of these features in order to find similar objects in the database. Collectively, these individual queries form a set query.

Another real-world application of set queries is in drug activity prediction [4] where one of the objectives is to find similar drug molecules that can and cannot bind with a target protein. In this application, the molecule frequently changes its shape due to biological transformations, and each shape is represented as a point in a high-dimensional space. The objective is to find similar drug molecules, to a given query molecule, that also undergo similar shape transformations. These different shape transformations also collectively form a set query.

Unfortunately, state-of-the-art LSH-based techniques allow users to input a success guarantee for each individual query point, instead of a guarantee for the entire set query. A lower guarantee on these individual queries can lead to overall misses, and a higher guarantee can lead to redundant and wasteful work for the whole set. Returning results (or features) that satisfy all the queries is an expensive process. Returning approximate results can save time as well as return “good enough” results. Given a set query, however, users may be interested in satisfying a target number of query points instead of all points in the set query, in order to get “good enough” results while improving the execution performance. Thus, for processing these set query based similarity searches, a better approach would be to design an index structure that can take a target number of points and a guarantee on the entire set as an input, and return the data points that satisfy at least “target” number of query points in the set query while satisfying the success guarantee, in an efficient and waste-avoiding manner.

1.2 Contributions of this Paper

To deal with this challenge, we design a novel index structure, *Point Set LSH (PSLSH)*, which creates an LSH index structure based on the Hamming distance (called $PSLSH_H$), on top of the existing LSH index structure (based on the Euclidean distance - called $PSLSH_E$). Since, especially for secondary-storage based implementations, the post-processing step (where the precise positions for the enumerated candidates are fetched from storage to identify and eliminate false positives¹) is the most expensive step, the index structure aims to generate as few candidates as possible. In particular, given a total budget of hash functions to use, we present a novel strategy that intelligently decides how many hash functions $PSLSH_E$ and $PSLSH_H$ use to provide the sought after guarantees on the entire set query. In particular, we show how to extend collision counting [12] so that it can effectively remove false positives in a multi-level index structure. We also present novel cost models that can effectively predict the total set query execution time for different strategies, and thus help choose the most efficient design strategy. To the best of our knowledge, this is the first work that presents an index structure that can give guarantees for an entire set query instead of guarantees on individual query points. Experimental evaluations show the effectiveness of $PSLSH$ in avoiding wasted and redundant work while giving a guarantee on the entire set query.

¹If the computed precise distance is more than the query radius, the candidate is a false positive and it is eliminated.

2 RELATED WORK

One solution to address the *curse of dimensionality* problem is to look for *approximate* results using approximate indexing techniques, such as Locality Sensitive Hashing [1]. While the original LSH scheme [1] was proposed for binary Hamming spaces, the authors then extended the scheme for Euclidean spaces [2], and since then LSH has also been proposed for other distance and similarity measures [15]. While LSH was originally designed to solve the (r,c)-Near Neighbor problem (introduced in Section 3), it has also been used to solve other related problems. In fact, most of the following works (except the notable exceptions of E2LSH², C2LSH [12], and QALSH [16]) have been primarily designed to solve the c-Approximate Nearest Neighbor problem [3, 10] (c-k-NN), where the goal is to find k neighbors such that the distance of the query point and the i th nearest neighbor is at most $(1 + c)$ times the distance from the query point to its true i th nearest neighbor.

2.1 Efficient Variants of LSH

In [11], the authors propose to create a prefix-tree of hash functions for each hash layer in order to return the desired number of top- m results. In [3], the authors propose a probing sequence for the hash buckets such that similar data points lie in neighboring hash buckets, and hence by probing neighboring buckets they are able to retrieve more results, while creating less number of hash layers. In [17], the authors project the original points into a new space using hash functions. Each point in this space is represented using a z-order code, and the points are further retrieved based on the similarities between their z-order codes. In [10], the authors propose SK-LSH, an index structure where similar compound hash keys (CHKs) are stored on the same page on the disk. In order to do so, they propose a distance measure between CHKs, and a linear order on these CHKs, which are then stored in the ascending order on the secondary storage. The main drawbacks of their approach is that they require the user to choose the number of layers and the width of the bucket of a hash function (which are data-dependent), and their formulation does not include the important parameters, error probability (δ) and approximation factor (c). In [18], the authors propose an LSH-based index structure for solving multiple queries with different distance metrics. In C2LSH [12], the authors propose to build a base of LSH functions, and then the points that collide most frequently (based on a count threshold) with the query point in these base functions are chosen as candidate points. By using this concept of “collision counting”, the number of candidate points are reduced without having the need to have large number of layers. This approach has been shown to be more effective by generating less number of candidates, using smaller index structures, than the original LSH index structure [12, 16]. In QALSH [16], the authors propose to build hash functions that are “query-aware”, i.e., the bucket of the hash functions are created based on an input query point. QALSH combines the concept of collision counting with “query-aware” hash buckets.

None of the above mentioned papers are designed to specifically handle set queries. Since they are designed for single queries, they are unable to efficiently reduce the overall number of candidates. In this paper, we build upon the concept of *collision counting* by

²<http://www.mit.edu/~andoni/LSH/>

using effective parameters based on our proposed cost models, and further reduce the number of candidates and the query processing time for processing a set query.

3 BACKGROUND AND PRELIMINARIES

A hash function family H is said to be (r, c, P_1, P_2) -sensitive if it satisfies all following conditions for any two points x and y in a data set $D \subset \mathbb{R}^d$:

- if $|x - y| \leq r$, then $\Pr[h(x) = h(y)] \geq P_1$, and
- if $|x - y| > cr$, then $\Pr[h(x) = h(y)] \leq P_2$.

Here, c is an approximation ratio, P_1 and P_2 are probabilities, and in order for the definition to work, $c > 1$ and $P_1 > P_2$. In the original LSH scheme for Euclidean distance, each hash function is defined as $h_{a,b}(v) = \frac{a \cdot v + b}{w}$, where a is a d -dimensional random vector with entries chosen independently from the standard normal distribution and b is a real number chosen uniformly from $[0, w]$, such that w is the width of the hash bucket [2]. This leads to the following collision probability function [10]:

$$P(r) = \int_0^w \frac{1}{r} \frac{2}{\sqrt{2\pi}} e^{-\frac{t^2}{2r^2}} (1 - \frac{t}{w}) dt.$$

To control false positives, basic LSH concatenates k hash functions to create a compound hash function for a single hash layer. As the number of hash functions increases, the recall of the entire index drops. In order to increase the recall, LSH creates m hash layers each consisting of these k hash functions. Let us assume that the user is interested in objects within distance r . The probability that the two points x and y collide in at least one hash layer (which is the same as the expected recall for the index) is $1 - \delta \geq 1 - (1 - P_1^k)^m$, where δ is a user-provided expected miss probability. In other words, the number, k , of hash functions per layer and the number, m , of layers can together be used to control the accuracy of the index structure.

3.1 Collision Counting Method

In C2LSH [12], it is theoretically shown that two close points x and y (i.e., $|x - y| \leq r$) collide in at least l layers with a probability $1 - \delta$, when the total number, m , of layers in the hash structure is set to be $m = \lceil \frac{\ln(\frac{1}{\delta})}{2(p_1 - p_2)^2} (1 + z)^2 \rceil$ where $z = \sqrt{\ln(\frac{2}{\beta}) / \ln(\frac{1}{\delta})}$, and β is the percentage of points whose distance with a query point is greater than cr . C2LSH sets $\beta = \frac{100}{n}$, where n is the cardinality of the dataset. The authors suggest that only those points that collide above a collision count threshold, l , with the query point are considered as candidate points. The collision count threshold is defined as a ratio of the total number of layers; more specifically, $l = \lceil \alpha \times m \rceil$, where the collision threshold percentage, α , is $\alpha = \frac{zp_1 + p_2}{1 + z}$. In C2LSH, since there is only one hash function per hash layer, the total number of hash functions are equal to the total number of available hash layers. Since we build PLSH on C2LSH, in the rest of the paper, the terms hash functions and hash layers are used interchangeably.

4 PROBLEM SPECIFICATION

Given a multidimensional database $\mathcal{D}$ that contains points that belong to a bounded multidimensional space $\mathcal{S}$, a single query representing a point in $\mathcal{S}$ is called a **point query**.

DEFINITION 4.1 (POSITIVE AND NEGATIVE POINT QUERY). A data point x satisfies a positive point query, q_i (and its corresponding radius r_{q_i}), if $\text{dist}(x, q_i) \leq r_{q_i}$. While a data point x satisfies a negative point query, q_i (and its corresponding radius r_{q_i}), if $\text{dist}(x, q_i) > r_{q_i}$.

Users may also be interested in avoiding particular objects from the database. Hence, we also design set queries such that it can contain point queries that return data points that are not similar to the point query. We call this a negative point query. We refer to a set of point queries as a **set query** Q .

DEFINITION 4.2 (SET QUERY SATISFACTION). Consider a set query Q that consists of s point queries. Each query q_i in Q is represented by its corresponding radius r_{q_i} and a flag f_{q_i} . The flag $f_{q_i} \in (0, 1)$ denotes whether a point query is a positive point query ($f_{q_i} = 1$) or a negative point query ($f_{q_i} = 0$). A point x satisfies q_i ,

- if $f_{q_i} = 1$ and $\text{dist}(x, q_i) \leq r_{q_i}$, or
- if $f_{q_i} = 0$ and $\text{dist}(x, q_i) > r_{q_i}$.

A data point x satisfies a given set query Q if x satisfies at least θ point queries (where $1 \leq \theta \leq s$) of Q .

In Section 6.2, we briefly describe how values of r_{q_i} are chosen in our analysis. While PLSH is also designed to handle negative point queries in a set query, in this paper, we only focus on positive point queries due to space limitations.

DEFINITION 4.3 (SET QUERY GUARANTEE). Every data point x that satisfies at least θ point queries in a given set query Q , should be reported with a probability of at least $1 - \delta$ probability (where δ is a user-specified probability).

In this paper, we answer two key questions:

1. Given a budget of total hash functions, how is the index designed such that the number of candidates are reduced while satisfying the user-input guarantee, $1 - \delta$, for the set query?
2. As noted earlier, the precise distance computation is an expensive process. Once the candidates from individual point queries are computed, the next step is to remove those points that do not satisfy at least θ queries. In the recently introduced *collision counting* approach, only the points that collide are considered as candidates. However, the existing collision threshold [12] does not consider the case when the set query is allowed a certain number of misses and false positives. The second question that we answer is: how can we account for a set query that has misses and false positives?

5 POINT SET LSH (PSLSH)

In this section, we describe the proposed index structure, *Point Set LSH* (PSLSH). In order to provide an accuracy guarantee for the entire set query, PLSH introduces an additional index structure based on the Hamming distance on top of the existing LSH index structure. We first explain the naive solution, that involves two steps, to our problem before describing the design of PLSH.

Step 1: for each point query q_i in the set query Q , determine a success guarantee $1 - \delta$, and find the candidates for each q_i in Q ,

Step 2: for each candidate point x , find all point queries satisfied by x (as explained in Section 4). Let us consider a list V_Q that contains the flags of each query in the given set query (i.e. the i th value of V_Q (V_Q^i) would be equal to the flag value of query q_i). Let us also consider that for every point x in the database, we have a list V_x where the i th value of V_x (V_x^i) = 1 if $\text{dist}(x, q_i) \leq r_i$, or $V_x^i = 0$ if $\text{dist}(x, q_i) > r_i$. In order for point x to be considered in the final result set of Q , x has to satisfy at least θ query points in Q .

There are two main problems with the naive approach:

- a) The user is unable to input an overall guarantee for the set query,

Figure 1: Architecture of PSLSH (for diff. Splitting Factors)

and instead has to input a guarantee on individual point queries, b) Underestimating the guarantee on individual point queries can lead to overall misses, while overestimating the guarantee on individual point queries can lead to redundant and wasteful work.

The condition in Step 2 above is equivalent to the following: point x is to be considered in the final results if the Hamming distance between V_Q and V_x is $\leq s - \theta$. Given this observation, we introduce an LSH index structure based on the Hamming distance, in order to give a guarantee on the set query.

5.1 Design of PSLSH

Similar to the original LSH [1], PSLSH is built for a user-specified m number of layers. In PSLSH, an additional index structure based on the Hamming distance (called PSLSH_H) is created in order to provide a guarantee on the entire set query, and in the process, avoid removing the false positives after the candidates for each query are found. Given a set query (that consists of s point queries), PSLSH randomly chooses ρ point queries to be processed (which is similar to randomly choosing a dimension in the Hamming space [1]). Figure 1 illustrates the design of PSLSH. Each of the ρ point queries are further processed using LSH-indexes based on the Euclidean distance (called PSLSH_E). Given a total budget of hash layers, the goal is to appropriately allocate these hash layers to PSLSH_E and PSLSH_H. In other words, we want to split the m total layers into ρ “sub-indexes” (hence, ρ is called the *Splitting Factor*), such that each sub-index will process a point query using $\frac{m}{\rho}$ layers. PSLSH_E generates candidates from individual point queries which are further pruned by using our novel modified Hamming Collision Threshold (Sec. 5.3.1). Figure 1 (a) shows the scenario where $\rho = 1$, i.e. one point query is randomly chosen to be processed and there is only one sub-index. Each of the m (which in Figure 1 = 10) layers are assigned to this sub-index. Similarly, when the splitting factor is 5 (Figure 1 (b)), there are 5 point queries to be processed on 5 sub-indexes (where each of the sub-indexes have 2 layers assigned to it). The following discussion is based on the observation that as the number of layers assigned to PSLSH_E decrease, the accuracy of PSLSH_E decreases [12]. Similarly, as the number of layers assigned to PSLSH_H increases, the accuracy of PSLSH_H increases.

Algorithm 1 Distribution of Layers to Sub-indexes

```

1: Input:
   • Total number of layers ( $m$ )
   • Splitting Factor ( $\rho$ )
2: Output: Mapping of layers to sub-indexes (i.e.  $\forall i \in \rho m_\rho^i$ )
3: procedure MAPLAYERS
4:   for sub-index  $i = 1; i \leq \rho; i++$  do
5:     if  $m \bmod \rho == 0$  then
6:        $m_\rho^i = \frac{m}{\rho}$ 
7:     else
8:       if  $i \leq m \bmod \rho$  then
9:          $m_\rho^i = \lfloor \frac{m}{\rho} \rfloor + 1$ 
10:      else
11:         $m_\rho^i = \lfloor \frac{m}{\rho} \rfloor$ 
12:      end if
13:    end if
14:  end for
15: end procedure

```

- **When $\rho = 1$** , there will be m layers assigned to PSLSH_E, whereas only 1 layer assigned to PSLSH_H. In this option, PSLSH_E will have high accuracy, but the accuracy of PSLSH_H will be low.
- On the contrary, **when $\rho = m$** , PSLSH_E will only have 1 layer per sub-index resulting in low accuracy, and PSLSH_H will have m layers resulting in high accuracy.

Since Hamming layers are chosen virtually, they do not incur any additional storage costs. Also, since choosing of the optimal splitting factor is only dependent on the data characteristics (and oblivious to the query workload characteristics), this process is done during index creation. When $s = \theta = 1$, i.e. there is only one query in the query workload, then PSLSH will act as C2LSH.

Since false positives are not removed from the candidates generated from PSLSH_E, one has to appropriately modify the collision threshold for PSLSH_H. Also, since the input to PSLSH_H is the output of PSLSH_E, it can further contain false negatives. The collision threshold for PSLSH_H is based on ρ , and the value of ρ determines the positive and negative collision probabilities for PSLSH_H. The theoretical analysis behind the modified positive and negative collision probabilities for the Hamming distance are presented in Section 5.3. By using this modified collision threshold for the Hamming distance, the candidates needed to find the final results are further reduced, thus improving the execution time of the given set query.

5.2 Distribution of Layers to ρ Sub-Indexes

In PSLSH, the m layers of the LSH index are split into ρ sub-indexes. If m is not completely divisible by ρ , then there can be an uneven distribution of these layers to the individual sub-indexes. One naive solution would be to assign $\lfloor \frac{m}{\rho} \rfloor$ layers to each of the $\rho - 1$ sub-indexes, and the remaining layers ($m - (\rho - 1) \cdot \lfloor \frac{m}{\rho} \rfloor$) are assigned to the ρ th index. This can lead to unpredictable behavior of δ_{FNE} and δ_{FPE} as the last sub-index can have significantly more layers to process the sub-queries resulting in a smaller δ_{FNE} and δ_{FPE} relative to the other layers. In order to avoid this, a layer distribution

strategy is presented in Alg. 1. The remaining layers are distributed among the ρ indexes (Lines 8-11) such that the maximum difference in the number of layers mapped to any sub-index is always 1.

5.3 Theoretical Analysis of the Positive and Negative Colliding Probabilities

In PLSH, we virtually add a layer of LSH based on the Hamming distance (PSLSH_H) in order to give a guarantee on the entire set query. But this process is not trivial, since in the original formulation [1], the input data does not consider the possibility of having false positives and false negatives. In our design of PLSH, once the candidates for each of the ρ point queries are found, the exact Euclidean distance of each candidate point with its corresponding query (in order to remove the false positives) is not computed. These candidates can also have misses resulting in false negatives. These candidates are an input to the PLSH_H. Hence, the original LSH formulation for the Hamming distance needs to be extended such that it accounts for the input data to possibly contain false positives and false negatives. First, we briefly describe the original LSH formulation as described in [1]. Let us denote the probability that two close points in the Hamming space are hashed to the same bucket as P_{1H} , and the probability that two distant objects are hashed to the same bucket as P_{2H} . For a given Hamming radius r_H , a d_H -dimensional Hamming space, and an approximation factor c_H , $P_{1H} = 1 - \frac{r_H}{d_H}$ and $P_{2H} = 1 - \frac{c_H r_H}{d_H}$.

In PLSH, the number of Hamming dimensions is equal to the number of queries in the set query (i.e. s), and since our goal is to satisfy at least θ queries, the Hamming radius $r_H = s - \theta$. However, unlike the original Hamming LSH formulation, where a hash collision indicates an evidence of a match, the input to PLSH_H (which are the candidates generated by PLSH_E for each of the ρ queries) can contain false negatives (also called misses) (i.e. V_x^i - as earlier defined in this section - should have been equal to 1 but is equal to 0 due to an error at the Euclidean layer) or false positives (i.e. V_x^i should have been equal to 0 but is returned as 1 due to an error of Euclidean LSH). Let us denote the error probability of a false negative generated by PLSH_E as δ_{FNE} , and the error probability that a false positive is generated by PLSH_E as δ_{FPE} . These potential false positives and false negatives can lead to the following four possible cases:

- **agrPRIN**: V_Q^i and V_x^i should be returned with the same value, i.e. they agree in principle,
- **disPRIN**: V_Q^i and V_x^i should be returned with different values, i.e. they disagree in principle,
- **agrPRAC**: V_Q^i and V_x^i are returned with the same value, i.e. they agree in practice,
- **disPRAC**: V_Q^i and V_x^i are returned with different values, i.e. they disagree in practice.

In order to find P_{1H} and P_{2H} while accounting for the false positives and false negatives at the Euclidean layer, we need to compute the following four probabilities:

- V_Q^i and V_x^i agree in practice and also agree in principle;
 $P(\text{agrPRAC} \cap \text{agrPRIN})$,
- V_Q^i and V_x^i agree in practice but disagree in principle;
 $P(\text{agrPRAC} \cap \text{disPRIN})$,

- V_Q^i and V_x^i disagree in practice but agree in principle;
 $P(\text{disPRAC} \cap \text{agrPRIN})$, and
- V_Q^i and V_x^i disagree in practice and also disagree in principle;
 $P(\text{disPRAC} \cap \text{disPRIN})$.

Let us denote the probability that two close points are hashed to the same bucket (while accounting for false positives and false negatives) as P'_{1H} .

$$P'_{1H} = P(\text{agrPRAC} \cap \text{agrPRIN}) + P(\text{agrPRAC} \cap \text{disPRIN}) + P(\text{disPRAC} \cap \text{agrPRIN}) + P(\text{disPRAC} \cap \text{disPRIN}) \quad (1)$$

In the original Hamming LSH formulation, $P(\text{agrPRAC}|\text{agrPRIN}) + P(\text{disPRAC}|\text{agrPRIN}) = 1$ and $P(\text{agrPRAC}|\text{disPRIN}) + P(\text{disPRAC}|\text{disPRIN}) = 0$. However, in PLSH, since Euclidean LSH, which is the input to the Hamming LSH, can introduce false negatives and false positives, we have $P(\text{agrPRAC}|\text{agrPRIN}) + P(\text{disPRAC}|\text{agrPRIN}) \leq 1$ and $P(\text{agrPRAC}|\text{disPRIN}) + P(\text{disPRAC}|\text{disPRIN}) \geq 0$.

Let us denote the number of true positives in the candidates generated by PLSH_E (for a single query q_i) as TP_E , the number of false positives generated as FP_E , the number of true negatives and false negatives generated as TNE and FNE respectively.

- $P(\text{agrPRAC}|\text{agrPRIN})$, the probability of true positives occurring, $= \frac{TP_E}{TP_E + FNE} = 1 - \delta_{FNE}$, and
- $P(\text{agrPRAC}|\text{disPRIN})$, the probability of false positives occurring, $= \frac{FP_E}{FP_E + TNE} = \delta_{FPE}$

Note that, from [1], we know that $P(\text{agrPRIN}) \leq 1 - \frac{r_H}{s}$ and $P(\text{disPRIN}) > \frac{r_H}{s}$ (where r_H is the Hamming radius). Due to the opposing inequalities, the two terms in Equation 1 cannot be simply added. Instead, since r_H has an integer value, each of the probabilities of the possible values of Hamming radius can be summed up to get the following:

$$P'_{1H} = \left[\sum_{h=0}^{r_H-1} P(h|h \leq r_H) \cdot \left(1 - \frac{h}{s}\right) \cdot (1 - \delta_{FNE}) \right] + \left[\sum_{h=0}^{r_H-1} P(h|h \leq r_H) \cdot \left(\frac{h}{s}\right) \cdot (\delta_{FPE}) \right]$$

where $P(h|h \leq r_H) = \frac{1}{r_H}$.

$$P'_{1H} = \frac{1}{r_H} \cdot (1 - \delta_{FNE}) \left[\sum_{h=0}^{r_H-1} \left(1 - \frac{h}{s}\right) \right] + \frac{1}{r_H} \cdot (\delta_{FPE}) \left[\sum_{h=0}^{r_H-1} \left(\frac{h}{s}\right) \right] \quad (2)$$

The negative collision probability (P'_{2H}) for a $c_H > 1$ can be similarly computed, but we omit it due to space limitations.

5.3.1 Calculation of Hamming Collision Threshold. By using Equation 2, the collision counting method (as proposed in [12] and described in Section 3) for PLSH_H can now be extended in order to handle the scenario where false positives and false negatives are introduced to the input data. For a user-provided failure probability threshold δ_H , only the candidates that collide with V_Q at least l_H times are considered (where $l_H = \lceil \rho \times \alpha_H \rceil$, $\alpha_H = \frac{z_H P'_{1H} + P'_{2H}}{1 + z_H}$,

$z_H = \sqrt{\ln(\frac{2}{\beta_H})/\ln(\frac{1}{\delta_H})}$, β_H is the percentage of points whose distance with the set query is greater than $c_H r_H$, and δ_H is the error percentage for the set query). As we experimentally show in Section 6, by using this modified collision threshold for PLSH_H, we are able to effectively reduce the final candidates and the overall set query processing time. This requires a way to distribute hash functions among the Euclidean and Hamming layers optimally.

5.4 Optimal Splitting Factor

As explained in Section 5.1, for a splitting factor ρ , the number of layers assigned to PLSH_E are (approximately) $\frac{m}{\rho}$, while the number of layers assigned to PLSH_H are ρ . Intuitively, for a very small ρ (e.g. $\rho = 2$), the number of candidates generated by PLSH_H are high because the number of layers assigned to PLSH_H are low, and hence the collision threshold is not able to effectively prune the false positives, which increases the total set query execution time. On the contrary, when ρ is very large (e.g. ρ close to m), then the number of layers assigned to PLSH_E are low, which causes PLSH_E to generate excessive false positives. Also, as ρ increases, more point queries have to be processed which increases the time taken by PLSH to identify the candidates by accessing the hash layers. Hence, we need to find the optimal value of the *Splitting Factor* ρ such that the total number of candidates generated by PLSH_E and PLSH_H are minimized, which in turn minimizes the set query execution time. Let us denote the set query execution time as $Time_Q$. This is dominated by two main sub-costs:

Index Access cost: This cost includes the time taken to access the index structure in order to find the candidates for all queries. This cost is denoted by $Time_{IA}$. $Time_{IA}$ is directly proportional to the number of point queries executed (i.e. the total number of sub-indexes). As the number of queries processed increases, the number of index accesses also increase, which increases the $Time_{IA}$, and vice-versa. Assuming λ_{IA} is an index access cost multiplier, function of the data size, we have $Time_{IA} = \lambda_{IA} \times f(\rho)$.

Candidate Processing cost: This cost (denoted by $Time_{CP}$) includes the time needed to bring each candidate into the memory from the secondary storage (IO), compute the exact distance between the candidate and each query in the set query, and compute whether the candidate satisfies at least θ queries of the given set query. The number of candidates generated are related to the number of queries processed, but it is not a straightforward conclusion as further explained in Section 5.4.1. Assuming λ_{CP} is a candidate processing cost multiplier, we have $Time_{CP} = \lambda_{CP} \times g(\rho)$. Our goal is to find ρ such that $Time_Q$ (where $Time_Q = Time_{IA} + Time_{CP}$) is minimized, i.e.

$$\begin{aligned} &\text{minimize} && (\lambda_{IA} \times f(\rho)) + (\lambda_{CP} \times g(\rho)) \\ &\text{subject to} && 1 \leq \rho \leq m. \end{aligned}$$

5.4.1 Relationship between ρ and $Time_{CP}$. $Time_{CP}$ is proportional to the number of candidates generated by PLSH. Each candidate point has to be brought from the secondary storage into the main memory for distance computation. Intuitively, as ρ increases (i.e. more hash functions are assigned to PLSH_H than PLSH_E), more candidates would be generated by PLSH_E but less candidates would be generated by PLSH_H. The main challenge then

is to choose the splitting factor (ρ) such that the least number of candidates are generated by both PLSH_E than PLSH_H.

The total number of candidates generated by PLSH are dependent on the collision thresholds used by PLSH_E and PLSH_H (i.e. l_E and l_H respectively). As explained in Sections 3.1 and 5.3.1, $l_E = \lceil m_E \times \alpha_E \rceil$ and $l_H = \lceil \rho \times \alpha_H \rceil$. In the following discussion, we use l'_E to denote $m_E \times \alpha_E$. Thus, we have $l_E \geq l'_E$. The goal is to estimate $Time_{CP}$ for different values of ρ . Since $Time_{CP}$ is directly proportional to the candidates generated, we need to estimate the number of candidates generated by PLSH for different ρ .

Let us consider two index structures PLSH_{E_i} and PLSH_{E_j} with different splitting factors, ρ_i and ρ_j . Let us denote their corresponding collision thresholds as l_{E_i} and l_{E_j} respectively. Let us also denote the number of candidates generated by PLSH_{E_i} as $cand(PLSH_{E_i})$, and by PLSH_{E_j} as $cand(PLSH_{E_j})$. Let us consider the similar notations (l_{H_i} , l_{H_j} , $cand(PLSH_{H_i})$, and $cand(PLSH_{H_j})$) for PLSH_H. In the collision counting approach (defined in [12]), the collision threshold has to be an integer since it counts the number of times a data point collides with a given query point. When the ceiling function is used on l'_E , the collision counting approach uses a more constrained collision threshold l_E (than l'_E). As the difference between $l_E - l'_E$ increases, this over-constraint increases misses (while still satisfying the user-input error guarantee) but also reduces false positives. Thus, the value of $l_E - l'_E$ needs to be taken into consideration when predicting the number of candidates generated by PLSH for a particular splitting factor.

OBSERVATION 5.1. *If $l_{E_i} - l'_{E_i} > l_{E_j} - l'_{E_j}$, then $cand(PLSH_{E_i}) < cand(PLSH_{E_j})$. The same holds true for PLSH_H. If $l_{H_i} - l'_{H_i} > l_{H_j} - l'_{H_j}$, then $cand(PLSH_{H_i}) < cand(PLSH_{H_j})$.*

As explained in Section 5.1, PLSH splits the total number of layers in ρ sub-indexes. If m is divisible by ρ , each ρ sub-index have exactly $\frac{m}{\rho}$ layers, otherwise, different sub-indexes need to have different numbers of layers. Let m_o be the number of layers assigned to the o th sub-index (and l_{E_o} be the collision threshold for m_o layers). In order to predict the number of candidates generated by PLSH_E, we define

$$Diff_{E\rho} = 1 - \frac{\sum_{o=1}^{\rho} l_{E_o} - l'_{E_o}}{\rho}$$

where for a given ρ , $Diff_{E\rho}$ defines the average of the difference between l_E and l'_E over the ρ sub-indexes. Similarly, for PLSH_H, we define $Diff_{H\rho} = 1 - (l_{H\rho} - l'_{H\rho})$. Since for any given ρ , there is only one collision threshold for PLSH_H (and hence an average is not needed unlike PLSH_E).

OBSERVATION 5.2. *For two splitting factors i and j , the total candidates generated by PLSH _{$\rho=i$} and PLSH _{$\rho=j$} depend on $Diff_{E\rho=i}$, $Diff_{H\rho=i}$, $Diff_{E\rho=j}$, and $Diff_{H\rho=j}$.*

The total number candidates generated by PLSH is dependent on the total number of candidates generated by PLSH_E and PLSH_H. If $Diff_{E\rho}$ is close to 1, then PLSH_E will produce less number of candidates than when $Diff_{E\rho}$ is close to 0. Similarly, when $Diff_{H\rho}$ is close to 1, then PLSH_H will produce less number of candidates than when $Diff_{H\rho}$ is close to 0. Thus, PLSH will generate less candidates when both $Diff_{E\rho}$ and $Diff_{H\rho}$ are

close to 1. But it will not necessarily generate less candidates when only $Diff_{E\rho}$ or $Diff_{H\rho}$ are close to 1, and the other is close to 0. Thus, scenarios when there is a large difference between $Diff_{E\rho}$ and $Diff_{H\rho}$ need to be penalized. Hence, we define another metric, $\phi_{Diff\rho}$, which takes the harmonic mean between $Diff_{E\rho}$ and $Diff_{H\rho}$ (and thus further penalizes the scenarios where there is a large difference between $Diff_{E\rho}$ and $Diff_{H\rho}$) in order to predict the number of candidates PLSH generates,

$$\phi_{Diff\rho} = \frac{2}{\frac{1}{Diff_{E\rho}} + \frac{1}{Diff_{H\rho}}} \quad (3)$$

While Observations 5.1 and 5.2 can be shown empirically, due to space limitations, in Section 6.4, we only show the effectiveness of the proposed $\phi_{Diff\rho}$ in estimating the number of candidates.

Mapping $\phi_{Diff\rho}$ to the Number of Candidates. In order to estimate the number of candidates generated by different splitting factors, there has to be a mapping between at least one $\phi_{Diff\rho}$ and the number of candidates generated by that particular ρ . In order to find this mapping, during runtime, PLSH finds the number of candidates generated for $\rho = 1$ by executing one query from the given set query. Note that, this is an inexpensive step because there is no need to build any additional layers or index structures and there is no need for a disk access either (since we only need to find the number of candidates generated and we do not need to remove any false positives - which is usually the most expensive part of the query execution time). Let us denote the number of candidates generated by PLSH for ρ as $cand(PSLH_{\rho})$. The number of candidates for $\rho > 1$ can then be estimated by considering the change between ϕ_{Diff} for two adjacent splitting factors.

$$\forall_{i=2}^m cand(PSLH_{\rho=i}) = \frac{\phi_{Diff_{\rho=i}}}{\phi_{Diff_{\rho=i-1}}} \times cand(PSLH_{\rho=i-1})$$

In Section 6.4, the accuracy of the proposed method of estimating the number of candidates for different splitting factors is evaluated. Once the number of candidates for different splitting factors are estimated, $Time_{CP}$ can then be estimated for different splitting factors. In Section 6, this time to calculate candidates for $\rho = 1$ is included in the overall query processing time. Note that, the values of ϕ_{Diff} for different splitting factors are not dependent on a single query, but are dependent on the collision thresholds for PLSH_E and PLSH_H. Since these values can be pre-calculated for a given dataset, the values of ϕ_{Diff} for different splitting factors can also be pre-calculated for a given dataset.

5.5 Space and Query Time Complexities

The total space complexity needed for PLSH is similar to C2LSH. PLSH requires space to store m layers (where $m = O(\log n)$ [12]) and the dataset. For a dataset with d dimensions and n data points, the space consumption for the data is $O(dn)$. In each hash layer, there are n data point IDs. Hence, the total space consumption for PLSH is $O(dn + n \log n)$.

PLSH has three major contributors to the execution time: (a) Locating the buckets for each of the ρ queries. Since the total number of buckets (for all ρ queries) needed to be located is always equal to m ($= O(\log n)$), the cost of locating m buckets for d -dimensional objects is $O(d \log n)$ [12]. (b) The second cost includes the cost of collision counting for PLSH_E and PLSH_H. Collision counting has

to be done for at most n data points over m layers, and collision counting for PLSH_H can be done while doing the collision counting for PLSH_E. Thus, the cost for collision counting is $O(n \log n)$ [12]. (c) The third cost includes the distance computation necessary to remove false positives to get the results for the set query execution. Let us consider $cand_Q$ is the number of candidates that PLSH generates. The cost of computing the distance of $cand_Q$ with s queries is $O(cand_Q ds)$. Thus, the total query time cost is $O(d \log n + n \log n + cand_Q ds)$.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of the proposed index structure, PLSH. For these evaluations, several real data sets with different characteristics were used, under different system parameters. Results presented in this paper were obtained using the Chameleon testbed³ supported by the National Science Foundation. M1.large instances, that ran an Ubuntu 16.04 operating system, consisting of an 8GB RAM and an 80GB non-volatile storage were used. All results presented are an average of 20 runs and 25 randomly selected set queries.

Use of C2LSH: PLSH builds upon the concept of collision counting [12]. Under the guidance of the authors of C2LSH [12], their source code (that could only run top-k nearest neighbor (c-k-NN) queries) was modified to run radius search (r,c)-NN queries. Since the same collision counting approach is used in C2LSH and QALSH [16], PLSH can work with either. In our evaluations, we observed that C2LSH produces fewer candidates than QALSH for (r,c)-NN queries and therefore, we use C2LSH as the basis for PLSH. E2LSH (the popular implementation of adhoc-LSH) is the only other implementation that directly aims at solving the (r,c)-NN problem. Adhoc-LSH is known to suffer from radius tuning problems, large memory footprint, and high IO costs [17], and hence not compared with. Since there is no work that directly aims at solving our problem, PLSH was evaluated against these alternatives:

Exact Search: In this brute force sequential scan approach, for each point query in the set query, we calculate the exact distance between each object in the dataset to check whether the object lies within the query radius.

In each of the following alternatives, the state-of-the-art C2LSH implementation is used for finding the candidates of the point queries.

LSH-Pre: Each point query in the set query is solved using C2LSH and the candidates for each of the point queries are found. Then the false positives that do not satisfy each of the point queries (by finding the exact distance between each candidate point and the corresponding point query) are removed, and then the points that satisfy at least θ point queries in the set query are selected.

LSH-Post: The candidates for each point query in the set query are found, candidates that satisfy at least θ point queries in the set query are picked, and finally the false positives that do not satisfy the set query are eliminated.

LSH-Hamming: This alternative is similar to PLSH, except the proposed modified Hamming Collision Threshold (explained in Section 5.3.1) is not used.

³<https://www.chameleoncloud.org/>

Figure 2: [Data=P53] (a) Splitting Factor (ρ) vs theoretical/observed P'_{1H} and P'_{2H} , (b) $\phi_{Diff\rho}$ vs the number of estimated and observed candidates for different Splitting Factors, (c) Set query execution time and observed recall of PLSH for different Splitting Factors, (d) Set Query Execution Time Breakdown

Parameter	Value range
Target Recall (δ_H)	0.85; 0.9 ; 0.95;
# of Queries in Query Workload (s)	30; 40 ; 50;
Ratio of Satisfied Queries (θ/s)	0.65; 0.75 ; 0.85;
Number of Layers (m)	125; 150 ; 200;

Table 1: Parameters and Default Values (in bold)

6.1 Datasets

For the experiments, the following four datasets were used. These datasets have varying number of points as well as varying number of dimensions (attributes). Similar to the setup in [12], the dimension values are normalized to be integers in the range of [0,10000].

Epidemic: This dataset consists of 193,185 128-dimensional SIFT points that are extracted from time-series simulation data depicting the SEIR model. The simulation ensembles for the SEIR model are generated by using the Spatiotemporal Epidemiological Modeler [21] as described in [22]. The page size is set to 4KB.

Mnist⁴: Consists of 60K objects of 50-dimensions. Page size = 4KB.

Sift⁵: Consists of 1,000,000 128-dimensional SIFT points. The page size is set to 4KB.

P53⁶: Consists of 31,008 objects of 5409-dimensions. The page size is set to 64KB.

6.2 Evaluation Criteria and Parameters

We evaluate the effectiveness of PLSH by comparing the total set query execution time to those of the alternative strategies. In this section, due to space constraints, we do not study the index cost. Since all alternatives are implemented on top of C2LSH, they have the same index size and creation time cost as C2LSH [12]. All implementations are disk-based, and hence we do not report memory usage. Since there is no explicit caching mechanism, we read a data object from the disk into the memory each time it is needed. Table 1 shows the parameters and their range values that are used in the evaluation of PLSH. The default values are presented in bold. In the experiments, $c_H = 2$, δ_{FPE} and δ_{FNE} values are chosen by using the Epidemic dataset. We show the effectiveness of these models by comparing the estimated values with the observed values for the P53 dataset. The observed values for the remaining datasets also follow similar behavior.

Distribution of Query Radii: The radii of the point queries are chosen such that the number of results for the point queries are

between 0.1% and 2% of the dataset size, and the number of results of the set queries are between 0.1% and 2% of the dataset size.

6.3 Analysis of Cost Models

Proposed Theoretical vs Observed Collision Probabilities: Figure 2 (a) shows how the proposed theoretical positive and negative collision probabilities effectively estimate the observed positive and negative collision probabilities for PLSH_H.

Estimated vs Observed Number of Candidates for Different Splitting Factors: Fig 2 (b) shows the effectiveness of $\phi_{Diff\rho}$ in predicting the behavior of the number of candidates generated by different splitting factors. It validates Observation 5.2 (explained in Section 5.4.1) that $\phi_{Diff\rho}$ is dependent on the values of l'_E and l'_H . Since different splitting factors can have different values of l'_E and l'_H , each splitting factor generates varying number of candidates. By using $\phi_{Diff\rho}$, we are able to estimate the number of candidates that different splitting factors will generate. In Figure 2 (b), we also compare the estimated number of candidates against the observed number of candidates for varying splitting factors.

6.4 Predicting the Optimal Splitting Factor

Once we can estimate the number of candidates, PLSH can choose the most effective splitting factor. The splitting factor that was chosen by PLSH always was the most efficient splitting factor when compared with the first 20 splitting factors. For number of layers (<200 ; which are enough number of layers), splitting factors above (approximately) 20 are not very effective because large number of false positives and misses are generated by PLSH_E (since very few layers are assigned to PLSH_E). Hence, in the following discussions, the most efficient splitting factor is found from the first 20 splitting factors. Figure 2 (c) shows the set query execution time for the first 20 splitting factors for the default settings (as shown in Table 1) for the P53 dataset. Figure 2 (b) shows that the splitting factor of 10 generates the least candidates, and Figure 2 (c) shows that the splitting factor of 10 is also the fastest among all splitting factors. For moderate-dimensional datasets (Epidemic, Mnist, and Sift), PLSH chooses a smaller splitting factor (within ranges $\rho = 1$ to $\rho = 3$) since it is beneficial to process less point queries (even at the expense of more candidates), whereas for high-dimensional datasets (P53), PLSH chooses a splitting factor that returns the least number of candidates since $Time_{CP}$ is the most dominating cost. In each scenario, PLSH was able to choose the most effective splitting factor. Due to space constraints, we only show the set query execution time and recall of different splitting factors for the P53 dataset. Due to the accuracy and effectiveness of the design of PLSH, PLSH can determine the most efficient splitting factor, while satisfying the user-provided success guarantee.

⁴<http://yann.lecun.com/exdb/mnist/>

⁵<http://corpus-texmex.irisa.fr/>

⁶<http://archive.ics.uci.edu/ml/datasets/p53+Mutants>

Figure 3: Comparison of PSLSH (execution time and recall) against its alternatives (for different datasets)

Figure 4: [Data=P53] (a) Time to fetch candidates, (b) Time to calc. exact distances between candidates and point queries

6.5 Discussion of the Performance Results

In this section, we analyze the execution performance of PSLSH against its alternatives. Figure 3 shows the execution times for different set queries for different datasets for the default settings. PSLSH is faster than all alternatives across all scenarios while satisfying the input target recall. PSLSH is able to reduce the query processing cost by appropriately choosing between different splitting factors by leveraging its novel cost models. While LSH-Hamming is the second-fastest alternative for the moderate-dimensional datasets (Epidemic, Mnist, and Sift), its performance degrades significantly for the high-dimensional dataset (P53). For P53, LSH-Post is the second-fastest alternative, because for moderate-dimensional datasets, $Time_{IA}$ starts becoming dominating as more point queries are processed. Hence, for moderate-dimensional datasets, PSLSH (and LSH-Hamming) chooses a smaller splitting factor whereas for high-dimensional datasets, PSLSH chooses a splitting factor since $Time_{CP}$ is the most dominating cost. While the exact search is comparable for moderate-dimensional datasets, it is magnitudes slower than all methods for the high-dimensional dataset. This is because the exact search has to perform the Euclidean distance check against all data objects, and as the dimensions increase, the distance check time becomes increasingly impractical. Figure 2 (d) shows the breakdown of the set query execution time for the moderate-dimensional dataset (Epidemic) and the high-dimensional dataset (P53). For Epidemic dataset, $Time_{IA}$ dominates the $Time_{CP}$ cost (for LSH-Pre). PSLSH (and LSH-Hamming) are able to choose the right trade-off between $Time_{IA}$ and $Time_{CP}$ by choosing a smaller splitting factor, whereas for the P53 dataset, $Time_{CP}$ dominates the $Time_{IA}$. Due to the effectiveness of our Hamming Collision threshold, PSLSH always generates less candidates than LSH-Hamming (since PSLSH can prune) and hence is faster in all scenarios. Since the performance of all strategies on moderate-dimensional datasets is similar, from Figure 6, due to space limitations, we only present the results for the Epidemic dataset among the moderate-dimensional datasets. Also, due to space limitations, we do not show the performance for the exact search which is always slower than LSH-Hamming and PSLSH for moderate-dimensional datasets and slower than all methods for high-dimensional datasets.

Number of Disk Accesses: Figure 5 shows the number of disk accesses required to execute a set query for PSLSH and its alternatives. LSH-Pre brings the candidates for each point query from the secondary storage. This can result in the same data point being brought into the memory multiple times, and hence it requires the most number of disk accesses. For LSH-Post, LSH-Hamming, and PSLSH, the number of disk accesses is equal to the number of candidates generated since these strategies only bring the candidates into the main memory once all point queries are processed. An interesting observation here is that even though the number of disk accesses for LSH-Pre are, in most cases, more than 50 times higher than that of PSLSH, the execution time (Figure 3) is, in most cases, only 10-15 times higher than that of PSLSH. As explained in Section 5.4, $Time_{CP}$ cost consists of the IO time (Figure 4 (a)) and the time to calculate the exact distance (Figure 4 (b)). While the IO time (P53 dataset) of LSH-Pre is ~ 100 times that of PSLSH (since the disk accesses are also ~ 100 times that of PSLSH), the time to calculate distance for LSH-Pre is ~ 2.56 times that of PSLSH. This is because, once PSLSH brings each candidate into the memory, it has to calculate the distance of the candidate point with each point query to remove false positives that do not satisfy θ queries. Hence, for the P53 dataset, while LSH-Pre requires 92279 distance computations (Fig. 5), PSLSH requires $846 \times 40 = 33840$ distance computations.

Impact of Varying Target Recall: One advantage of LSH based index structures is that it provides the user the ability to control the target recall and thus explicitly trade-off accuracy against execution time. Figure 6 shows the impact of varying target recall for different datasets. In these figures, the secondary axis shows the achieved recall for the different alternative strategies. As these figures show, LSH-Pre and LSH-Post are unable to adapt to the varying recall. Both strategies return very high recall even when the user seeks significantly lower target recall rates, which results in wasted work and slower execution times. PSLSH, instead, is able to adapt appropriately according to the input target recall for the set query while satisfying the target recall in all cases.

Impact of Varying Number of Layers: Figure 7 shows that, for varying layers, PSLSH is still faster than its alternatives for different number of layers, across all datasets, due to its ability to choose an optimal splitting factor.

Impact of Varying Number of Point Queries in the Set Query: Figure 8 shows that PSLSH can still perform better than its competitors even when the number of point queries in the set query vary. For moderate-dimensional datasets, as consistent with the previous results, PSLSH is much faster than LSH-Pre and LSH-Post. For high-dimensional datasets, LSH-Post is the fastest among the alternative strategies, but still slower than PSLSH. As the number of point queries in the set query vary, the time gain of PSLSH over LSH-Post remains consistently high.

Figure 5: Comparison of PSLSH (disk accesses) against its alternatives (for different datasets)

Figure 6: Impact of varying target recall (0.85 and 0.95)

Figure 7: Impact of varying number of layers (125 and 200)

Figure 8: Impact of varying number of point queries (30 and 50)

Due to space limitations, we do not present the results for varying % of required satisfied point queries in a given set query. We again observe that PSLSH is the fastest strategy across all datasets while satisfying the target recall.

7 CONCLUSION

In this paper, we presented a novel index structure, *Point Set LSH* (PSLSH) for efficiently executing a set query while providing guarantees. Often times, points that satisfy certain number of queries are needed to answer similarity search queries. Traditional LSH-based index structures enable accuracy guarantees on individual point queries instead over data that is represented in the form of sets of multidimensional features. This can lead to potential misses and false positives, which lead to higher query processing times. In this work, we introduced a novel index structure, PSLSH, which intelligently combines Hamming distance based and Euclidean distance based LSH to solve the above challenge. Along with our novel cost models, we also presented the design and the theoretical analysis of PSLSH. The experimental analysis proves the effectiveness of PSLSH for different datasets under different settings.

REFERENCES

- [1] P. Indyk, et al. Approximate nearest neighbors: Towards removing the curse of dimensionality. STOC 1998.
- [2] M. Datar, et al. Locality-sensitive hashing scheme based on p-stable distributions. SCG 2004.
- [3] Q. Lv, et al. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. PVLDB 2007.
- [4] O. Maron, et al. A Framework for Multiple-instance Learning. NIPS 1997.
- [5] W. Zhang, et al. Data-oriented locality sensitive hashing. MM 2010.
- [6] J. Gao, et al. DSH: Data Sensitive Hashing for High-dimensional K-NN search. SIGMOD 2014.
- [7] Y. Sun, et al. SRS: Solving C-approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. PVLDB 2014.
- [8] A. Andoni, et al. Optimal Data-Dependent Hashing for Approximate Near Neighbors. STOC 2015.
- [9] A. Joly, et al. A posteriori multi-probe locality sensitive hashing. MM 2008.
- [10] Y. Liu, et al. Sk-lsh: An efficient index structure for approximate nearest neighbor search. PVLDB 2014.
- [11] M. Bawa, et al. Lsh forest: Self-tuning indexes for similarity search. WWW 2005.
- [12] J. Gan, et al. Locality-sensitive hashing scheme based on dynamic collision counting. SIGMOD 2012.
- [13] D. Lowe. Distinctive image features from scale-invariant keypoints. IJCV 2004.
- [14] H. Bay, et al. Surf: Speeded up robust features. ECCV 2006.
- [15] J. Wang, et al. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.
- [16] Q. Huang, et al. Query-aware locality-sensitive hashing for approximate nearest neighbor search. PVLDB 2015.
- [17] Y. Tao, et al. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. TODS 2010.
- [18] Y. Zheng, et al. LazyLsh: Approximate nearest neighbor search for multiple distance functions with a single index. SIGMOD 2016.
- [19] A. Dasgupta, et al. Fast locality-sensitive hashing. SIGKDD 2011.
- [20] J. Gao, et al. Selective Hashing: Closing the Gap between Radius Search and k-NN Search. SIGKDD 2015.
- [21] S. Edlund, et al. The spatiotemporal epidemiological modeler. IHI 2010.
- [22] S. Liu, et al. epidms: Data management and analytics for decision-making from epidemic spread simulation ensembles. The Journal of Infectious Diseases 2016.