2018 IEEE 34th International Conference on Data Engineering

GPH: Similarity Search in Hamming Space

Jianbin Qin† Yaoshu Wang† Chuan Xiao‡ Wei Wang† Xuemin Lin† Yoshiharu Ishikawa‡
†University of New South Wales, Australia
{jqin, yaoshuw, weiw, lxue}@cse.unsw.edu.au
‡ Nagoya University, Japan
chuanx@nagoya-u.jp ishikawa@i.nagoya-u.ac.jp

***Abstract*—A similarity search in Hamming space finds binary vectors whose Hamming distances are no more than a threshold from a query vector. It is a fundamental problem in many applications, including image retrieval, near-duplicate Web page detection, and machine learning. State-of-the-art approaches to answering such queries are mainly based on the *pigeonhole principle* to generate a set of candidates and then verify them.**

We observe that the constraint based on the pigeonhole principle is *not always tight* and hence may bring about unnecessary candidates. We also observe that the distribution in real data is often *skew*, but most existing solutions adopt a simple equi-width partitioning and allocate the same threshold to all the partitions, and hence fail to exploit the data skewness to optimize the query processing. In this paper, we propose a new form of the pigeonhole principle which allows variable partition size and threshold. Based on the new principle, we first develop a tight constraint of candidates, and then devise cost-aware methods for dimension partitioning and threshold allocation to optimize query processing. Our evaluation on datasets with various data distributions shows the robustness of our solution and its superior query processing performance to the state-of-the-art methods.

I. Introduction

Finding similar objects is a fundamental problem in database research and has been studied for several decades [31]. Among many types of queries to find similar objects, Hamming distance search on binary vectors is an important one. Given a query q, a Hamming distance search finds all vectors in a database whose Hamming distances to q are no greater than a threshold τ. Answering such queries efficiently plays an important role in many applications, including Web search, image search, and scientific database. For example:

- For image retrieval, images are converted to compact binary vectors and those within a Hamming distance threshold are identified as candidates for further image-level verification [33]. Recently, deep learning has become remarkably successful in image recognition. Learning to hash algorithms that utilize neural networks have been actively explored [15], [17], [7]. In these studies, images are represented by binary vectors and Hamming distance is utilized to capture the dissimilarity.
- For information retrieval, state-of-the-art methods represent text documents by binary vectors through hashing [8]. Google converts Web pages into 64-bit vectors and uses Hamming similarity search to detect near-duplicate Web pages [20].
- For scientific databases, a fundamental task in cheminformatics is to find similar molecules [11], [22]. In this task, molecules are converted into binary vectors, and the Tanimoto similarity is used to measure the similarity between molecules. This similarity constraint can be converted to an equivalent Hamming distance constraint [34].

The naïve algorithm to answer a Hamming distance search query requires access of every vector in the database; hence it is expensive and does not scale well to large datasets. Therefore, there has been much interest in devising efficient indexes and algorithms. Many existing methods [1], [16], [34], [23] adopt the *filter-and-refine* framework to quickly find a set of candidates and then verify them. They are based on the naïve application of the *pigeonhole principle* to this problem: If the n dimensions are partitioned into m *equi-width* parts (in this paper, we assume $n \bmod m = 0$), then a necessary condition for the Hamming distance of two vectors to be within τ is that they must share a part in which the Hamming distance is within $\lfloor \frac{\tau}{m} \rfloor$. This leads to a *filtering condition*, and produces a set of candidate vectors, which are then verified by calculating the Hamming distances and comparing with the threshold. As a result, the efficiencies of these methods critically depend on the candidate size.

However, despite the success and prevalence of this framework, we identify that the filtering condition has two inherent major weaknesses: (1) **The threshold on each partition is not always tight**. Hence, many unnecessary candidates are included. For example, when $m = 3$, the filtering conditions for τ in $[9, 11]$ are the same (Hamming distance $\leq \lfloor \frac{\tau}{m} \rfloor = 3$), and hence will produce the same set of candidates.

(2) **The thresholds on the partitions are evenly distributed**. It assumes a uniform distribution and does not work well when the dataset is skewed. We found that many real datasets are skewed to varying degrees and complex correlations exist among dimensions. Fig. 1 shows that 8 out of 11 real datasets have dimensions with skewness greater than 0.3 [1], and 5 out of the 8 datasets contain a vector whose frequency ≥ 0.1 on a partition, meaning that at least 1/10 data vectors become candidates if the query matches the data vector on this partition.

In this paper, we propose a novel method to answer the Hamming distance search problem and address the above-mentioned weaknesses. We propose a tight form of the pigeonhole principle named *general pigeonhole principle*. Based on the new principle, the thresholds of the m partitions sum up to $\tau - m + 1$, less than τ, thus yielding a stricter filtering condition than the existing methods. In addition, the threshold on each partition is a *variable* in the range of $[-1, \tau]$, where -1 indicates that this

[1]To measure the skewness of the i-th dimension, we calculate the numbers of vectors whose values on the i-th dimension are 0 and 1, respectively, and then take the ratio of their difference and the total number of vectors.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00013

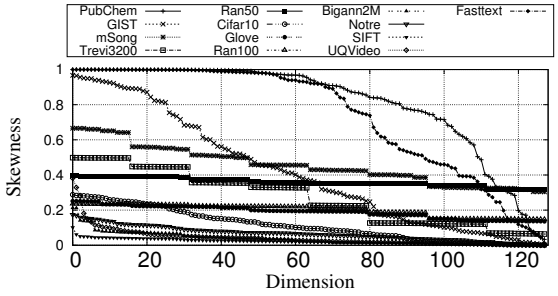


Fig. 1. Skewness ($\frac{|\#1s-\#0s|}{\#data}$) by dimension of datasets in [14].

partition is ignored when generating candidates. This enables us to choose proper thresholds for different partitions in order to improve query processing performance. We prove that the candidate condition based on the general pigeonhole principle is *tight*; i.e., the threshold allocated to each partition cannot be further reduced. To tackle data skewness and dimension correlations, we first devise an online algorithm to allocate thresholds to partitions using a query processing cost model, and then devise an offline algorithm to optimize the partitioning of vectors by taking account of the distribution of dimensions. The proposed techniques constitute the GPH algorithm. Experiments are run on several real datasets with different data distributions. The results show that the GPH algorithm performs consistently well on all these datasets and is faster than state-of-the-art methods by up to two orders of magnitude.

Our contributions can be summarized as follows. (1) We propose a new form of the pigeonhole principle to obtain a tight filtering condition and enable flexible threshold allocation. (2) We propose an efficient online query optimization method to allocate thresholds on the basis of the new pigeonhole principle. (3) We propose an offline partitioning method to address the selectivity issue caused by data skewness and dimension correlations. (4) We conduct extensive experimental study on several real datasets to evaluate the proposed method. The results demonstrate the superiority of the proposed method over state-of-the-art methods.

II. Preliminaries

A. Problem Definition

In this paper, we focus on the similarity search on binary vectors. We can view an object as an n-dimensional binary vector x. $x[i]$ denotes the value of the i-th dimension of x. Let $\Delta(x[i], y[i]) = 0$, if $x[i] = y[i]$; or 1, otherwise. The Hamming distance between two vectors x and y, denoted $H(x, y)$, is the number of dimensions on which x and y differ:

$$H(x, y) = \sum_{i=1}^{n} \Delta(x[i], y[i]).$$

Hamming distance is a symmetric measure. If we regard x (respectively, y) as a yardstick, we can also say that y (respectively, x) has $H(x, y)$ errors with respect to x (respectively, y).

Given a collection of data objects $\mathcal{D}$, a query object q, a Hamming distance search is to find all data objects whose Hamming distance to q is no greater than a threshold τ, i.e., $\{\, x \mid x \in \mathcal{D}, H(x, q) \leq \tau \,\}$.

B. Basic Pigeonhole Principle

Most exact solutions to Hamming distance search are based on the filter-and-refine framework to generate a set of candidates that satisfy a necessary condition of the Hamming distance constraint. The majority of these methods [1], [16], [34], [23] are based on the intuition that if two vectors are similar, there will be a pair of similar partitions from the two vectors. Hence the (basic) pigeonhole principle is utilized by these methods.

Lemma 1 (Basic Pigeonhole Principle): x and y are divided into m partitions. Each partition consists of $\frac{n}{m}$ dimensions. Let x_i and y_i $(1 \leq i \leq m)$ denote each partition in x and y, respectively. If $H(x, y) \leq \tau$, there exists at least one partition i such that $H(x_i, y_i) \leq \lfloor \frac{\tau}{m} \rfloor$.

A data object x satisfying the condition $\exists i$, $H(x_i, q_i) \leq \lfloor \frac{\tau}{m} \rfloor$ is called a *candidate*. Since candidates are verified by computing the Hamming distance to the query, the query processing performance depends heavily on the candidate number.

C. Overview of Existing Approaches

We briefly introduce a state-of-the-art method, Multi-index Hamming (MIH) [23]; other methods based on the basic pigeonhole principle work in a similar way. The n dimensions are divides into m equi-width partitions. In each partition, based on basic pigeonhole principle, it performs Hamming distance search on $n' = \lfloor \frac{n}{m} \rfloor$ dimensions with a threshold $\tau' = \lfloor \frac{\tau}{m} \rfloor$. MIH builds an inverted index offline, mapping each partition of a data object to the object ID. For each partition of the query, it enumerates n'-dimensional vectors whose Hamming distances to the partition are within τ', called *signatures*. It looks up signatures in the index to find candidates and verifies them.

D. Weaknesses of Basic Pigeonhole Principle

Next we analyze the major drawbacks of the filtering condition based on the basic pigeonhole principle. Note that the filtering condition is uniquely characterized by a vector of thresholds allocated to each corresponding partition; we call the vector *threshold vector*, and denote the one used by the basic pigeonhole principle as $T_{\text{basic}} = [\lfloor \frac{\tau}{m} \rfloor, \ldots, \lfloor \frac{\tau}{m} \rfloor]$. We also define the *dominance* relationship between threshold vectors. Let n_i denote the number of dimensions in the i-th partition. T_1 dominates T_2, or $T_1 \prec T_2$, iff $\forall i \in \{1, \ldots, m\}$, $T_1[i] \leq T_2[i]$ and $[T_1[i], T_2[i]] \cap [-1, n_i - 1] \neq \emptyset$, and $\exists i, T_1[i] < T_2[i]$.

- T_{basic} is **not always tight**. By the tightness of a threshold vector T, we mean that (1) (correctness) every vector whose Hamming distance to the query is within the threshold will be found by the filtering condition based on T, and (2) (minimality) there does not exists another vector T' that dominates T yet still guarantees correctness. As the candidate size is monotonic with respect to the threshold, an algorithm based on a threshold vector which dominates T_{basic} will generate fewer or at most equal number of candidates compared with an algorithm based on T_{basic}.

  Example 1: Consider $\tau = 9$ and $m = 3$. The threshold vector T_{basic} is $[3, 3, 3]$. We can find a dominating threshold vector $T = [2, 2, 3]$ which is tight and guarantees both correctness and minimality. Note that there may be multiple

tight threshold vectors for the same τ. E.g., another tight threshold vector for the example can be $[2, 3, 2]$ or $[4, 3, 0]$ [2].

- The filtering condition does **not adapt to the data distribution** in the partitions. Skewness and correlations among dimensions often exist in real data. Equal allocation of thresholds, as done in T_{basic}, may result in poor selectivity for some partitions, hence excessive number of candidates. Several recent studies recognized this issue and proposed several methods to either obtain relatively less skew partitions by partition rearrangement [34] or allocating varying thresholds heuristically to different partitions [10]. In contrast, we propose that skewed partitions can be beneficial and we can reduce the candidate size by judiciously allocating different thresholds to different partitions for *each* query to exploit such skewness, as shown in Example 2.

Example 2: Suppose $n = 8$, $m = 2$, and $\tau = 2$. Consider the four data vectors and the query, and two different partitioning schemes in Table I. Consider the first query and existing method will use $T_{\text{basic}} = [1, 1]$. This will results in all the four data vectors recognized as candidates, but only one (x^1) is the result. If we use the first six dimensions as one partition and the rest two dimensions as the other dimension, and use $T = [2, 0]$, the candidate size will be reduced to 2 (x^1 and x^2).

TABLE I
BENEFITS OF ADAPTIVE PARTITIONING AND THRESHOLDING

	Equi-width Partitioning		Variable Partitioning	
	Partition 1	Partition 2	Partition 1	Partition 2
$x^1 = 00000000$	0000	0000	000000	00
$x^2 = 00000111$	0000	0111	000001	11
$x^3 = 00001111$	0000	1111	000011	11
$x^4 = 10011111$	1001	1111	100111	11
$q^1 = 10000000$	1000	0000	100000	00
	$\tau_1 = 1$	$\tau_2 = 1$	$\tau_1 = 2$	$\tau_2 = 0$

III. GENERAL PIGEONHOLE PRINCIPLE

In this section, we propose a general form of the pigeonhole principle which allows variable thresholds to guarantee the tightness of threshold vectors.

We begin with the allocation of thresholds. Given a threshold vector, we use the notation $\|T\|_1$ to denote the sum of thresholds in all the partitions, i.e., $\|T\|_1 = \sum_{i=1}^{m} T[i]$. The flexible pigeonhole principle is stated below.

Lemma 2 (Flexible Pigeonhole Principle): A partitioning $\mathcal{P}$ divides a n-dimensional vector into m disjoint partitions. x and y are partitioned by $\mathcal{P}$. Consider a vector $T = [\tau_1, \ldots, \tau_m]$ such that τ_i are integers and $\|T\|_1 = \tau$. If $H(x, y) \leq \tau$, there exists at least one partition i such that $H(x_i, y_i) \leq \tau_i$.

Proof: Assume that $\nexists i$ such that $H(x_i, y_i) \leq \tau_i$. Since partitions are disjoint, $H(x, y) = \sum_{i=1}^{m} H(x_i, y_i) > \sum_{i=1}^{m} \tau_i$. Hence $H(x, y) > \tau$, which contradicts that $H(x, y) \leq \tau$. ■

The principle stated by Lemma 2 is more flexible than the basic pigeonhole principle in the sense that we can choose arbitrary thresholds for different partitions. Intuitively, we may tolerate more errors for selective partitions and fewer errors for unselective partitions.

To achieve tightness, we first extend the threshold allocation from integers to real numbers.

Lemma 3: x and y are partitioned by $\mathcal{P}$ into m disjoint partitions. Consider a vector $T = [\tau_1, \ldots, \tau_m]$ in which the thresholds are real numbers. $\|T\|_1 = \tau$. If $H(x, y) \leq \tau$, there exists at least one partition i such that $H(x_i, y_i) \leq \lfloor \tau_i \rfloor$.

Proof: The proof of Lemma 2 also applies to real numbers. Therefore, if $\sum_{i=1}^{m} \tau_i = \tau$ and $H(x, y) \leq \tau$, then $\exists i$, $H(x_i, y_i) \leq \tau_i$. Because τ_i are real numbers and $H(x_i, y_i)$ are integers, $\exists i$, $H(x_i, y_i) \leq \lfloor \tau_i \rfloor$. ■

Definition 1 (Integer Reduction): Given a threshold vector $T = [\tau_1, \tau_2, \ldots, \tau_m]$, we can reduce it to $T' = [\lfloor \tau_1 \rfloor, \lfloor \tau_2 \rfloor, \ldots, \lfloor \tau_m \rfloor]$. This reduction is called *integer reduction*.

It is obvious that the candidate size does not change after an integer reduction, as the Hamming distances must be integers.

When we combine Lemma 3 and the integer reduction technique, they can produce a threshold vector which dominates T_{basic}, as shown in Example 3.

Example 3: Recall in Example 1, T_{basic} is $[3, 3, 3]$ using the basic pigeonhole principle.

To obtain a dominating vector, we can start with a possible threshold vector $T = [2.9, 2.9, 3.2]$. Then by the integer reduction technique, T is reduced to $T' = [2, 2, 3]$. To see this is correct, if $\nexists i$, $H(x_i, y_i) \leq T'[i]$, there will be $3+3+4 = 10$ errors between x and y. Compared to $[3, 3, 3]$, T' is a dominating threshold vector, and the constraints on the first two partitions are stricter.

The above example also shows that the sum of thresholds of partitions can be reduced. The following lemma and theorem show how they work in the general case and the tightness guarantee of the resulting threshold vectors.

Lemma 4 (General Pigeonhole Principle): x and y are partitioned by $\mathcal{P}$ into m disjoint partitions. Consider a threshold vector T composed of integers. $\|T\|_1 = \tau - m + 1$. If $H(x, y) \leq \tau$, there exists at least one partition i such that $H(x_i, y_i) \leq \tau_i$.

Proof: Given a vector $T = [\tau_1, \ldots, \tau_m]$ such that $\|T\|_1 = \tau - m + 1$, we consider another vector $T' = [\tau'_i, \ldots, \tau'_m] = [\tau_1 + 1, \ldots, \tau_{m-1} + 1, \tau_m]$; i.e., it equals to T on the last partition and is greater than T by 1 in the other $m - 1$ partitions. Because $\|T'\|_1 = \|T\|_1 + (m - 1) = \tau$, by Lemma 2, if $H(x, y) \leq \tau$, then $\exists i$, $H(x_i, y_i) \leq \tau'_i$.

For the first $(m - 1)$ dimensions in T', we decrease each of their thresholds by a small positive real number ϵ, and for the last dimension, we increase the threshold by $(m - 1)\epsilon$; i.e., the sum of thresholds does not change. Hence we have a vector $T'' = [\tau''_i, \ldots, \tau''_m] = [\tau_1 + 1 - \epsilon, \ldots, \tau_{m-1} + 1 - \epsilon, \tau_m + (m - 1)\epsilon]$. Because $\|T''\|_1 = \|T'\|_1 = \tau$, by Lemma 3, if $H(x, y) \leq \tau$, then $\exists i$, $H(x_i, y_i) \leq \lfloor \tau''_i \rfloor$. Because

$$\lfloor \tau''_i \rfloor = \begin{cases} \lfloor \tau_i + 1 - \epsilon \rfloor = \tau_i, & \text{if } i < m; \\ \lfloor \tau_i + (m - 1)\epsilon \rfloor = \tau_i, & \text{if } i = m, \end{cases}$$

[2]Please refer to Section III for more explanation of tightness.

if $H(x,y) \leq \tau$, then $\exists i$, $H(x_i, y_i) \leq \tau_i$. ■

One may notice that in the above proof, the partitions we choose to decrease thresholds are not limited to the first $(m-1)$ ones. Therefore, given a threshold vector T such that $\|T\|_1 = \tau$, we may choose any $(m-1)$ partitions and decrease their thresholds by 1. For the resulting vector T', $\|T'\|_1 = \tau - m + 1$. We may use it as a stricter condition to generate candidates and the correctness of the algorithm is still guaranteed. We call the process of converting T to T' *ϵ-transformation*.

Theorem 1: The filtering condition based on the general pigeonhole principle is tight.

Proof: The correctness is stated in Lemma 4. We prove the minimality. Given a threshold vector T based on the general pigeonhole principle, i.e., $\|T\|_1 = \tau - m + 1$, we consider a threshold vector T' which dominates T, i.e., $\forall i \in \{1, \ldots, m\}$, $T'[i] \leq T[i]$ and $[T'[i], T[i]] \cap [-1, n_i - 1] \neq \emptyset$, and $\exists j \in \{1, \ldots, m\}$, $T'[j] < T[j]$. Because $\forall i \in \{1, \ldots, m\}$, $H(x_i, q_i) \in [0, n_i]$ and $[T'[i], T[i]] \cap [-1, n_i - 1] \neq \emptyset$, we may construct a vector x such that $\forall i \in \{1, \ldots, m\}$, $H(x_i, q_i) = \max(0, T'[i] + 1)$. $\forall i \in \{1, \ldots, m\}$, because $T'[i] \leq T[i]$ and $[T'[i], T[i]] \cap [-1, n_i - 1] \neq \emptyset$, $H(x_i, q_i) \leq T[i] + 1$. Because $\exists j \in \{1, \ldots, m\}$, $T'[j] < T[j]$, $\exists j \in \{1, \ldots, m\}$, $H(x_i, q_i) \leq T[i]$. Because $H(x_i, q_i) > T'[i]$ on all the partitions, x is not a candidate by T'. However, $H(x,q) = \sum_{i=1}^{m} H(x_i, q_i) \leq \|T\|_1 + m - 1 = \tau$, meaning that x is result of the query. Therefore, the filtering condition based on T' is incorrect, and thus the minimality of T is proved. ■

One surprising but beneficial consequence of the ϵ-transformation is that the resulting threshold of a partition may become *negative*. For example, $[1, 0, 0]$ becomes $[0, 0, -1]$[3] if the first and third partitions are chosen to decrease thresholds. Since $H(x_i, y_i)$ is a non-negative integer, $H(x_i, y_i) \leq T[i]$ is always false if $T[i]$ is negative. This fact indicates that the partitions with negative thresholds can be **safely ignored** for candidate generation. As will be shown in the next section, this allows us to ignore partitions where the query and most of the data are identical. This endows our method the *unique* ability to handle highly skewed data or partitions.

Example 4: Consider the four data vectors and two queries in Table II. For q^1, we show the threshold vectors based on the flexible pigeonhole principle and the general pigeonhole principle. The candidate sizes are 2 and 1, respectively. For q^2, we show two different threshold vectors based on the general pigeonhole principle. The candidate sizes are 4 and 2, respectively.

IV. Threshold Allocation

To utilize the general pigeonhole principle to process queries, there are two key issues: (1) how to divide the n dimensions into m partitions, and (2) how to compute the threshold vector T such that $\|T\|_1 = \tau - m + 1$. We will tackle the first issue in Section V with an offline solution. Before that, we focus on the second issue in this section and propose an online algorithm.

[3]Note that in our method, we only consider the case of -1 for the negative threshold of a partition since the other negative values are not necessary.

TABLE II
Threshold Vector and Their Candidate Sizes

	Partition 1	Partition 2
$x^1 = 00000000$	000000	00
$x^2 = 00000111$	000001	11
$x^3 = 00001111$	000011	11
$x^4 = 10011111$	100111	11
$q^1 = 10000000$	100000	00
$q^2 = 10000011$	100000	11

q^1	$T = [2, 0]$	$Cand = \{x^1, x^2\}$
	$T = [1, 0]$	$Cand = \{x^1\}$
q^2	$T = [1, 0]$	$Cand = \{x^1, x^2, x^3, x^4\}$
	$T = [2, -1]$	$Cand = \{x^1, x^2\}$

A. Cost Model

To optimize the threshold allocation, we first analyze the query processing cost. Like MIH, we also build an inverted index offline to map each partition of a data object to the object ID. Then for each partition of the query, we enumerate signatures to generate candidates.

The query processing cost consists of three parts:

$$C_{query\_proc}(q,T) = C_{sig\_gen}(q,T) + C_{cand\_gen}(q,T) + C_{verify}(q,T),$$

where C_{sig_gen}, C_{cand_gen}, and C_{verify} denote the costs of signature generation, candidate generation, and verification, respectively.

For each partition i, a signature is a vector whose Hamming distance is within τ_i to the i-th partition of query q. Since we enumerate all such vectors, the signature generation cost is

$$C_{sig\_gen}(q,T) = \sum_{i=1}^{m} \binom{n_i}{\tau_i} \cdot c_{enum},$$

where n_i denotes the number of dimensions in the i-th partition, and c_{enum} is the cost of enumerating the value of a dimension in a given vector. If $\tau_i < 0$, the cost is 0 for the i-th partition.

Let S_{sig} denote the set of signatures generated. The candidate generation cost can be modeled by inverted index lookup:

$$C_{cand\_gen}(q,T) = \sum_{s \in S_{sig}} |I_s| \cdot c_{access},$$

where $|I_s|$ denotes the length of the postings list of signature s, and c_{access} is the cost of accessing an entry in a postings list.

The verification cost is

$$C_{verify}(q,T) = |S_{cand}| \cdot c_{verify},$$

where S_{cand} is the set of candidates, and c_{verify} is the cost to check if two n-dimensional vectors' Hamming distance is within τ.

In practice, the signature generation cost is usually much less than the candidate generation cost and the verification cost (see Section VII-B for experiments). So we can ignore the signature

generation cost when optimizing the threshold allocation. In addition, it is difficult to accurately estimate the size of S_{cand} using the lengths of postings lists, because it can be reduced from the minimal k-union problem [29], which is proved to be NP-hard. Nonetheless, $|S_{cand}|$ is upper-bounded by the sum of candidates generated in all the partitions, i.e., $\sum_{s \in S_{sig}} |I_s|$. Our experiments (Section VII-B) show that the ratio of $|S_{cand}|$ and this upper bound depends on data distribution and τ. Given a dataset, the ratio with respect to varying τ can be computed and recorded by generating a number of queries and processing them. Let α denote this ratio. We may rewrite the number of candidates in the form of $\alpha \cdot \sum_{i=1}^{m} CN(q_i, \tau_i)$, where $CN(q_i, \tau_i)$ is the number of candidates generated by the i-th partition of the query q with a threshold of τ_i (when $\tau_i = -1$, $CN(q_i, \tau_i) = 0$). Hence the query processing cost can be estimated as:

$$\widehat{C_{query\_proc}}(q, T) = \sum_{i=1}^{m} CN(q_i, \tau_i) \cdot (c_{access} + \alpha \cdot c_{verify}). \tag{1}$$

With the above cost model, we can formulate the threshold allocation as an optimization problem.

Problem 1 (Threshold Allocation): Given a collection of data objects $\mathcal{D}$, a query q and a threshold τ, find the threshold vector T that minimizes the estimated query processing cost under the general pigeonhole principle; i.e.,

$$\arg\min_{T} \widehat{C_{query\_proc}}(q, T), \quad \text{s.t. } \|T\|_1 = \tau - m + 1.$$

B. Threshold Allocation Algorithm

Since c_{access}, c_{verify}, and α are independent of $CN(q_i, \tau_i)$, we can omit the coefficient $(c_{access} + \alpha \cdot c_{verify})$ in Equation 1 and find the minimum query processing cost with only $CN(q_i, \tau_i)$. The computation of $CN(q_i, \tau_i)$ values will be introduced in Section IV-C. Here we treat $CN(q_i, \tau_i)$ as a black box with $O(1)$ time complexity and propose an online threshold allocation algorithm based on dynamic programming.

Let $OPT[i, t]$ record the minimum query processing cost (omitting the coefficient $(c_{access} + \alpha \cdot c_{verify})$) for partitions $1, \ldots, i$ with a sum of thresholds t. We have the following recursive formula:

$$OPT[i,t] = \begin{cases} \min\limits_{e=-1}^{t+i-1} OPT[i-1, t-e] + CN(q_i, e), & \text{if } i > 1; \\ CN(q_i, t), & \text{if } i = 1. \end{cases}$$

With the recursive formula, we design a dynamic programming algorithm for threshold allocation, whose pseudo-code is shown in Algorithm 1. It first initializes the costs for the first partition (Lines 1 – 2), i.e., $OPT[1, -1], \ldots, OPT[1, \tau]$. Then it iterates through the other partitions and compute the minimum costs (Lines 3 – 10). Note that the negative threshold -1 is also consider for each partition. Finally, we trace the path that reaches $OPT[m, \tau - m + 1]$ to obtain the threshold vector (Lines 11 – 14). The time complexity of the algorithm is $O(m \cdot (\tau + 1)^2)$.

Algorithm 1: DPAllocate(q, m, τ)

```
1  for e = −1 to τ do
2      OPT[1, e] ← CN(q_1, e), PATH[1, e] ← e;
3  for i = 2 to m do
4      for t = −i to τ − i + 1 do
5          c_min = +∞;
6          for e = −1 to t + i − 1 do
7              if OPT[i − 1, t − e] + CN(q_i, e) < c_min then
8                  c_min ← OPT[i − 1, t − e] + CN(q_i, e);
9                  e_min ← e;
10         OPT[i, t] = c_min, PATH[i, t] = e_min;
11 e ← τ − m + 1;
12 for i = m to 1 do
13     T[i] ← PATH[i, e];
14     e ← e − PATH[i, e];
15 return T;
```

Example 5: Consider a dataset of 100 binary vectors and we partition it into 4 partitions. Given a query q, for each partition i, suppose the numbers of candidates (denoted CN_i) under different thresholds are provided in the table below.

	$\tau_i = -1$	$\tau_i = 0$	$\tau_i = 1$	$\tau_i = 2$	$\tau_i = 3$	$\tau_i = 4$
CN_1	0	5	10	15	50	100
CN_2	0	10	80	90	95	100
CN_3	0	5	15	20	70	100
CN_4	0	10	70	80	95	100

We use Algorithm 1 to compute the threshold vector. The $OPT[i, t]$ values are given in the table below.

t =	$i = 1$	$i = 2$	$i = 3$	$i = 4$
-3	0	0	0	5
-2	0	0	5	10
-1	0	5	10	20
0	5	15	20	30
1	10	20	20	30
2	**15**	**25**	35	45
3	50	60	40	45
4	100	110	**45**	**55**

The minimum query processing cost $OPT[4, 4] = 55$. We trace the path (in boldface) that reaches this value and obtain the threshold vector $[2, 0, 2, 0]$.

C. Computing Candidate Numbers

In order to run the threshold allocation algorithms, we need to obtain the candidate numbers $CN(q_i, \tau_i)$ beforehand. An exact solution to computing $CN(q_i, \tau_i)$ is to enumerate all possible vectors for the i-th partition and then count how many vectors in $\mathcal{D}$ has a Hamming distance within τ_i to the enumerated vector in this partition. These numbers are stored in a table. When processing the query, with the given q_i, the table is looked up for the corresponding entry $CN(q_i, \tau_i)$. The time complexity of this algorithm is $O(m \cdot 2^n \cdot 2^\tau)$, and the space complexity is $O(m \cdot 2^n)$. This method is only feasible when n and τ are small. To cope with large n and τ, we devise two approximation algorithms to estimate the number of candidates.

Sub-partitioning. The basic idea of the first approximation algorithm is splitting q_i into smaller equi-width sub-partitions

and estimating $CN(q_i, \tau_i)$ with the candidate numbers of the sub-partitions. We divide q_i into m_i sub-partitions. Each sub-partition has a fixed number of dimensions so that its candidate number can be computed using the exact algorithm in reasonable amount of time and stored in main memory. For the thresholds of the sub-partitions, we may use the general pigeonhole principle and divide τ_i into m_i values such that they sum up to $\tau_i - m_i + 1$. Let q_{ij} denote a sub-partition of q_i and τ_{ij} denote its threshold. Let $G(m_i, \tau_i)$ be the set of threshold vectors of which the total thresholds sum up to no more than $\tau_i - m_i + 1$; i.e., $\{\, [\tau_{i1}, \ldots, \tau_{im_i}] | \tau_{ij} \in [-1, \tau_i] \wedge \sum_{j=1}^{m_i} \tau_{ij} \leq \tau_i - m_i + 1 \,\}$.

We offline compute all the $CN(q_{ij}, \tau_{ij})$ values for all $\tau_{ij} \in [-1, \tau_i]$ using the aforementioned exact algorithm; i.e., enumerate all possible query vectors and then count how many data vectors in $\mathcal{D}$ has a Hamming distance within τ_{ij} to the enumerated vector in this sub-partition. We assume that the candidates in the m_i sub-partitions are independent. Then $CN(q_i, \tau_i)$ can be approximately estimated online with the following equation.

$$\widehat{CN(q_i, \tau_i)} = \sum_{g \in G(m_i, \tau_i)} \prod_{j=1}^{m_i} (CN(q_{ij}, g[j]) - CN(q_{ij}, g[j] - 1)).$$

Machine Learning. We may also use machine learning technique to predict the candidate number for a given $\langle q_i, \tau_i \rangle$. For each τ_i, we regard each dimension of q_i as a feature and randomly generate feature vectors $x_k = [b_1, \ldots, b_{|q_i|}]$. The candidate number $CN(x_k, \tau_i)$ can be obtained by processing x_k as a query with a threshold τ_i. Then we apply the regression model on the training data $\mathcal{T}_i = \{\, \langle x_k, CN(x_k, \tau_i) \rangle \,\}$.

Let $h_{\tau_i}(x_i, \theta_i)$ denote the machine learning model, where θ_i denotes its parameters. Traditional regression models utilize mean squared error as loss function. To reduce the impact of large $CN(x_k, \tau_i)$, we use relative error as our loss function: $J(\mathcal{T}_i, \theta_i) = \sum_{k=1}^{|\mathcal{T}_i|} \{ \frac{CN(x_k, \tau_i) - h_{\tau_i}(x_k, \theta_i)}{CN(x_k, \tau_i)} \}^2$. According to [25], we utilize the approximation $ln(t) \approx t - 1$ to estimate $J(\mathcal{T}_i, \theta_i)$:

$$\begin{aligned} J(\mathcal{T}_i, \theta_i) &= \sum_{k=1}^{|\mathcal{T}_i|} \left\{ 1 - \frac{h_{\tau_i}(x_k, \theta_i)}{CN(x_k, \tau_i)} \right\}^2 \\ &\approx \sum_{i=1}^{|\mathcal{T}_i|} \left\{ \ln \frac{CN(x_k, \tau_i)}{h_{\tau_i}(x_k, \theta_i)} \right\}^2 \\ &= \sum_{i=1}^{|\mathcal{T}_i|} \{ \ln CN(x_k, \tau_i) - \ln h_{\tau_i}(x_k, \theta_i) \}^2. \end{aligned}$$

From the above equation, we can simply convert training data $\langle x_k, CN(x_k, \tau_i) \rangle$ into $\langle x_k, \ln CN(x_k, \tau_i) \rangle$ and then take mean squared error to train an SVM model with RBF kernel.

V. Dimension Partitioning

To deal with data skewness and dimension correlations, the existing methods for Hamming distance search resort to random shuffle [1] or dimension rearrangement [34], [30], [18]. All of them are aiming towards the direction that the dimensions in each partition or the signatures in the index are uniformly distributed, so as to reduce the candidates caused by frequent signatures. In this section, we present our method for dimension partitioning. We devise a cost model of dimension partitioning and convert the partitioning into an optimization problem to optimize query processing performance. Then we propose the algorithm to solve this problem.

A. Cost Model

Let P_i denote a set of dimensions in the range $[1, n]$. Our goal is to find a partitioning $\mathcal{P} = \{\, P_1, \ldots, P_m \,\}$ such that $P_i \cap P_j = \emptyset$ if $i \neq j$, and $\cup_{i=1}^{m} P_i = \{\, 1, \ldots, n \,\}$. Given a query workload $\mathcal{Q} = \{ < q^1, \tau^1 >, \ldots, < q^{|\mathcal{Q}|}, \tau^{|\mathcal{Q}|} > \}$, the query processing cost of the workload is the sum of the costs of its constituent queries:

$$C_{workload}(\mathcal{Q}, \mathcal{P}) = \sum_{i=1}^{|\mathcal{Q}|} \widehat{C_{query\_proc}}(q^i, \tau^i, \mathcal{P}), \qquad (2)$$

where $\widehat{C_{query\_proc}}(q^i, \tau^i, \mathcal{P})$ is the processing cost of query q^i with a threshold τ^i, which can be computed using the dynamic programming algorithm proposed in Section IV. Then we can formulate the dimension partitioning as an optimization problem.

Problem 2 (Dimension Partitioning): Given a collection of data objects $\mathcal{D}$, a query workload $\mathcal{Q}$, find the partitioning $\mathcal{P}$ that minimizes the query processing cost of $\mathcal{Q}$ under the general pigeonhole principle; i.e.,

$$\arg\min_{\mathcal{P}} C_{workload}(\mathcal{Q}, \mathcal{P}).$$

Lemma 5: The dimension partitioning problem is NP-hard.

Proof: We can reduce the dimension partitioning problem from the number partitioning problem [2], which is to partition a multiset of positive integers, S, into two subsets S_1 and S_2 such that the difference between the sums in two sets is minimized. Consider a special case of $m = 2$ and a $\mathcal{Q}$ of only one query. Let S be a multiset of n positive integers, each representing a dimension in the dimension partitioning problem. Let $sum(S)$ denote the sum of numbers in S. For $i \in \{\, 1, 2 \,\}$, Let $CN(q_i, \tau_i) = sum(S_i)^2$, $\forall \tau_i \in [-1, \tau]$; i.e., the candidate number in partition i equals to the square of the sum of numbers in this partition. By Equations 1 and 2, $C_{workload}(\mathcal{Q}, \mathcal{P}) = (sum(S_1)^2 + sum(S_2)^2) \cdot (c_{access} + \alpha \cdot c_{verify})$. $C_{workload}$ is minimized when the difference between $sum(S_1)$ and $sum(S_2)$ is minimized. Hence the special case of dimension partitioning problem is reduced from the number partitioning problem. Because the number partitioning problem is NP-complete, the dimension partitioning is NP-hard. ■

B. Partitioning Algorithm

Seeing the difficulty of the dimension partitioning problem, we propose a heuristic algorithm to select a good partitioning: first generate an initial partitioning and then refine it.

Algorithm 2 captures the pseudo-code of the heuristic partitioning algorithm. It first generates an initial partitioning $\mathcal{P}$ of m partitions (Line 1). The details of the initialization step will be introduced in Section V-C. Then the algorithm iteratively

Algorithm 2: HeuristicPartition($\mathcal{D}, \mathcal{Q}, m$)

```
1  P ← InitialPartition(D, Q, m);
2  c_min ← C_workload(Q, P);
3  f ← true;
4  while f = true do
5      f ← false;
6      foreach P_i ∈ P do
7          foreach d ∈ P_i do
8              P'_i ← P_i \ {d}, P' ← (P \ P_i) ∪ P'_i;
9              foreach P_j ∈ P, j ≠ i do
10                 P'_j ← P_j ∪ {d}, P' ← (P' \ P_j) ∪ P'_j;
11                 if C_workload(Q, P') < c_min then
12                     f ← true;
13                     c_min ← C_workload(Q, P');
14                     P_min ← P';
15     if f = true then
16         P ← P_min;
17 return P;
```

improves the current partitioning by selecting the best option of moving a dimension from one partition to another. In each iteration, we pick a dimension from a partition P_i (Line 8), try to move it to another partition P_j, $j \neq i$ (Line 10), and compute the resulting query processing cost of the workload. We try all possible combination of P_i and P_j, and the option that yields the minimum cost is taken as the move of this iteration (Line 16). The above steps repeat until the cost cannot be further improved by moving a dimension. The time complexity of the algorithm is $O(lmnc)$. l is the number of iterations. c is the time complexity of computing the cost of the workload, $O(|\mathcal{Q}| \cdot m \cdot (\tau+1)^2)$. We also note that due to the replacement of dimensions, partitions may become empty in our algorithm. Hence it is not mandatory to output exactly m partitions for an input partition number m.

For the input query workload $\mathcal{Q}$, in case a historical query workload is unavailable, a sample of data objects can be used as a surrogate. Our experiments show that even if the distribution of real queries are different from the query workload that we use to compute the partitioning, our query processing algorithm still achieves good performance (Section VII-G). We also note that we may assign varying thresholds to the queries in the workload $\mathcal{Q}$. The benefit is that we can offline compute the partitioning using the workload which cover a wide range of thresholds, and then build an index without being aware of the thresholds of real queries beforehand.

C. Initial Partitioning

Since the dimension partitioning algorithm stops at a local optimum, we may achieve a better result with a carefully selected initial partitioning. The correlation of dimensions play an important role here. Unlike the existing methods which try to make dimensions in each partition uniformly distributed, our method aims at the opposite direction. We observe that the query processing performance is usually improved if highly correlated dimensions are put into the same partitions. This is because our threshold allocation algorithm works online and optimizes each query individually. When highly correlated dimensions are put together, more errors are likely to be identified in a partition, and thus our threshold allocation algorithm can assign a larger threshold to this partition and smaller thresholds to the other partitions; i.e., choosing proper thresholds for different partitions. If the dimensions are uniformly distributed, all the partitions will have the same distribution and there is little chance to optimize for specific partitions.

We may measure the correlation of dimensions with entropy. For a partition P_i, we project all the data objects in $\mathcal{D}$ on the dimensions of P_i, and use $\mathcal{D}_{P_i}$ to denote the set of the resulting vectors. The correlation of the dimensions of P_i is measured by:

$$H(\mathcal{D}_{P_i}) = -\sum_{X \in \mathcal{D}_{P_i}} P(X) \cdot \log P(X).$$

According to the definition of entropy, a smaller value of entropy indicates a higher correlation of the dimensions of P_i. The entropy of the partitioning $\mathcal{P}$ is the sum of the entropies of its constituent partitions:

$$H(\mathcal{P}) = \sum_{i=1}^{m} H(\mathcal{D}_{P_i}).$$

Our goal is to find an initial partitioning $\mathcal{P}$ to minimize $H(\mathcal{P})$. To achieve this, we generate an equi-width partitioning in a greedy manner: Starting with an empty partition, we select the dimension which yields the smallest entropy if it is put into this partition. This is repeated until a fixed partition size $\lfloor \frac{n}{m} \rfloor$ is reached, and thereby the first partition is obtained. Then we repeat the above procedure on the unselected dimensions to generate the other $(m-1)$ partitions.

VI. The GPH Algorithm

Based on the general pigeonhole principle and the techniques proposed in Sections IV and V, we devise the GPH (short for the **G**eneral **P**igeonhole principle-based algorithm for **H**amming distance search) algorithm.

The GPH algorithm consists of two phases: indexing phase and query processing phase. In the indexing phase, it takes as input the dataset $\mathcal{D}$, the query workload $\mathcal{Q}$, and a tunable parameter m for the number of partitions. The partitioning $\mathcal{P}$ is generated using the heuristic partitioning algorithm proposed in Section V. Then for each n-dimensional vector x in $\mathcal{D}$, we divided it by $\mathcal{P}$ into m partitions. Then for the projection of x on each partition, the ID of vector x is inserted into the postings list of this projection. In the query processing phase, the query q and the threshold τ are input to the algorithm. It first partitions q by $\mathcal{P}$ into m partitions. Then the threshold vector T is computed using the dynamic programming algorithm proposed in Section IV. For the projection of q on each partition, we enumerate the signatures whose Hamming distances to the projection do not exceed the allocated threshold. Then for each signature, we probe the inverted index to find the data objects that have this signature in the same partition, and insert the vector IDs into the candidate set. The candidates are finally verified using Hamming distance and the true results are returned. We omit the pseudo-code here in the interest of space.

VII. Experiments

We report experiment results and analyses in this section.

A. Experiments Setup

The following algorithms are compared in the experiment.

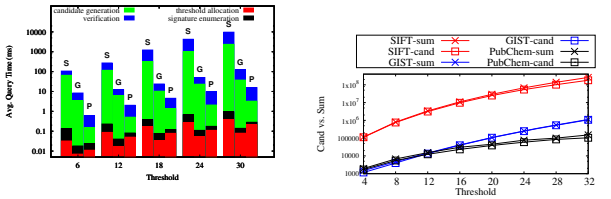
- **MIH** is a method based on the basic pigeonhole principle [23]. It divides vectors into m equi-width partitions and uses a threshold $\lfloor \frac{\tau}{m} \rfloor$ on all the partitions to generate candidates. Its filtering condition is not tight. Signatures are enumerated on the query side. We choose the fastest m setting for this method on each dataset.
- **HmSearch** is a method based on the basic pigeonhole principle [34]. Vectors are divided into $\lfloor \frac{\tau+3}{2} \rfloor$ equi-width partitions. It has a filtering condition in multiple cases but not tight. The threshold of a partition is either 0 or 1.
- **PartAlloc** is a method to solve the set similarity join problem [10]. It divides vectors into $\tau+1$ equi-width partitions and allocate thresholds to partitions with three options: -1, 0, and 1. Its filtering condition is tight. Signatures are enumerated on both data and query vectors. We convert the Jaccard similarity constraint to an equivalent Hamming distance constraint [1]. The greedy method is chosen to allocate thresholds.
- **LSH** is an algorithm to retrieve approximate answers. We convert the Hamming distance constraint to an equivalent Jaccard similarity constraint and then use the minhash LSH [5]. The dimension which yields the minimum hash value is chosen as a minhash. k minhashes are concatenated into a single signature, and this is repeated l times to obtain l signatures. We set $k = 3$ and recall to 95%. $l = \lceil \log_{1-t^k}(1-r) \rceil$, where t is the Jaccard similarity threshold.
- **GPH** is the method proposed in this paper.

Other methods for Hamming distance search, e.g., [16], [13], [20], are not compared since prior work [34] showed they are outperformed by HmSearch. We do not consider the method in [26] because it focuses on small n (≤ 64) and small τ (≤ 4), and it is significantly slower than the other algorithms in our experiments. E.g., on GIST, when $\tau = 8$, its average query response time is 128 times longer than GPH. The approximate method proposed in [24] is only fast for small thresholds. On SIFT, when $\tau \geq 12$, it becomes slower than MIH even if the recall is set to 0.9 [24]. Due to its performance compared to MIH and the much larger threshold settings in our experiments, we do not compare with the method in [24].

We select three publicly available real datasets with different data distributions and application domains.

- **SIFT** is a set of 1 billion SIFT features from the BIGANN dataset [12]. We follow the method used in [23] to convert them into 128-dimensional binary vectors.
- **GIST** is a set of 80 million 256-dimensional GIST descriptors for tiny images [28].
- **PubChem** is a database of chemical molecules. We sample 1 million entries, each of which is a 881-dimensional vector.

As can be seen from Fig. 1, SIFT has the smallest skewness among the three. GIST is a medium skewed dataset. PubChem is a highly skewed dataset. In addition to the three real datasets, we generate a synthetic dataset with varying skewness.



(a) Response Time Decomposed (b) Compare $\sum_{s \in S_{sig}} |I_s|$ and S_{cand}

Fig. 2. Justification of Assumptions

We sample a subset of 100 vectors from each dataset as the query workload for the partitioning of GPH. To generate real queries, for each dataset we sample 1,000 vectors (**differ** from the query workload for partitioning) and take the rest as data objects. We vary τ and measure the query response time averaged over 1,000 queries. For GPH and PartAlloc, threshold allocation time are also included. The τ settings are up to 32, 64, and 32 on the three datasets, respectively. The reason why we set smaller thresholds on PubChem is that due to the skewness, more than 10% data objects are results when $\tau = 32$.

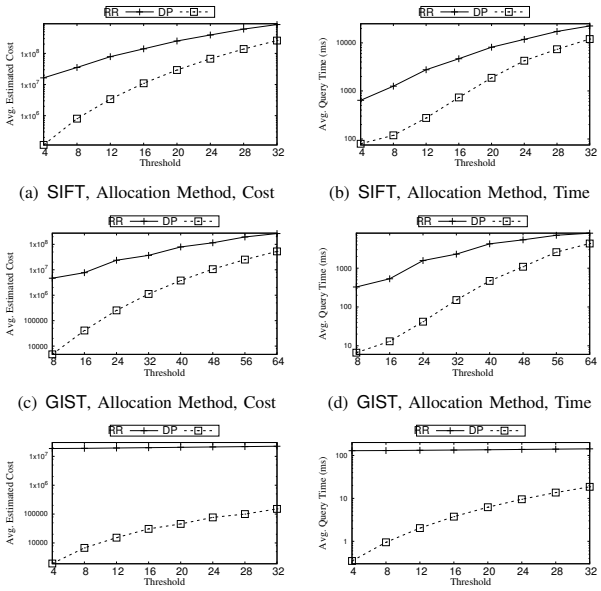
The experiments are carried out on a server with a Quad-Core Intel Xeon E3-1231 @3.4GHz Processor and 96GB RAM, running Debian 6.0. All the algorithms are implemented in C++ in a main memory fashion.

B. Justification of Assumptions

We first justify our assumptions for the cost model of threshold allocation. Fig. 2(a) shows the query processing time of GPH on the three datasets (denoted S, G, and P, respectively). The time is decomposed into four parts: threshold allocation, signature enumeration, candidate generation, and verification. The figure is plotted in **logscale** so that threshold allocation and signature enumeration can be seen. Compared to candidate generation and verification, the time spent on threshold allocation and signature enumeration is negligible ($< 3\%$), meaning that we can ignore them when estimating the query processing cost. Fig. 2(b) shows the sum of candidates generated in all the partitions ($\sum_{s \in S_{sig}} |I_s|$, denoted dataset-sum) and the candidate sizes ($|S_{cand}|$, denoted dataset-cand) on the three datasets. It can be seen that $|S_{cand}|$ is upper-bounded by $\sum_{s \in S_{sig}} |I_s|$. The ratio of them varies from 0.69 to 0.98, depending on dataset and τ. The ratios on different datasets and τ settings are recorded as the value of α in Equation 1 for cost estimation.

C. Evaluation of Threshold Allocation

We evaluate threshold allocation by comparing with a baseline algorithm (denoted RR). RR allocates thresholds in a round robin manner, and the thresholds of all partitions sum up to $\tau - m + 1$. For a fair comparison, we randomly shuffle the dimensions and then use the equi-width partitioning (m is chosen for the best performance) for the competitors in this set of experiments. Figs. 3(a), 3(c), and 3(e) show the query processing costs (in terms of candidate numbers) estimated by DP on the three datasets. We also plot the costs of RR using our cost model.



(a) SIFT, Allocation Method, Cost (b) SIFT, Allocation Method, Time

(c) GIST, Allocation Method, Cost (d) GIST, Allocation Method, Time

(e) PubChem, Allocation Method, Cost (f) PubChem, Allocation Method, Time

Fig. 3. Evaluation of Threshold Allocation

The corresponding query response times are shown in Figs. 3(b), 3(d), and 3(f). The trends of the cost and the time are similar, indicating that the cost model effectively estimates the query processing performance. DP is significantly faster than RR in query processing, and the gap is more remarkable on datasets with more skewness. On PubChem, the time of RR is close to sequential scan due to the skewness. With judicious threshold allocation, the time is reduced by nearly two orders of magnitude.

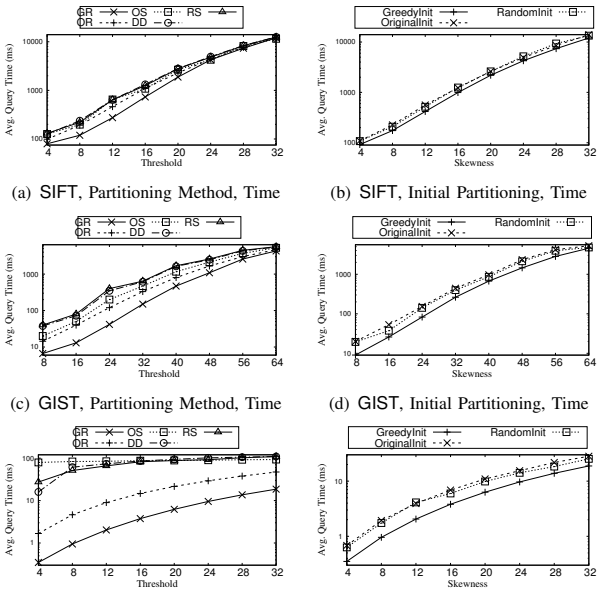
To evaluate the candidate number computation, we compare the sub-partitioning algorithm (denoted SP) and the machine learning algorithm based on SVM model (denoted SVM). To show why we choose SVM as the machine learning model, we also compare with two other learning models: random forest (RF) and a 3-layer deep neural network (DNN). The number of sub-partitions is 2. The size of the training data is 1,000 for the machine learning algorithms. Table III shows the relative errors with respect to the exact method and the times of candidate number computation (in microseconds). Since the performances on the real datasets are similar, we only show the results on the GIST dataset. The relative error of SVM is very small, and it is more accurate and faster than SP. To compare learning models, the relative error of RF is much higher than the other methods. Although DNN estimates candidate numbers slightly more accurately than SVM in some settings, their relative errors are both very small, and the running time of DNN is much more than SVM. In addition, we tried logistic regression and gradient boosting decision tree. Their relative errors are higher than the above methods and hence not shown here. Seeing these results, we choose the machine learning algorithm based on SVM model to estimate candidate numbers in the rest of the experiments.

D. Evaluation of Dimension Partitioning

To evaluate the effect of partitioning, we compare our method (denoted GR) with the following competitors: (1) OR is to use

TABLE III
ESTIMATION WITH VARIOUS MODELS ON GIST (EACH CELL SHOWS PERCENTAGE ERROR AND PREDICTION TIME (μS), SEPARATED BY /)

τ	SP	SVM	RF	DNN
16	1.75%/0.47	**1.64%/0.31**	8.73%/0.40	1.78%/2.64
32	0.37%/0.77	0.28%/**0.28**	12.43%/0.39	**0.19%**/2.60
48	0.15%/2.67	0.10%/**0.43**	9.26%/0.73	**0.08%**/3.83
64	0.07%/3.45	0.06%/**0.29**	3.58%/0.44	**0.03%**/2.44



(a) SIFT, Partitioning Method, Time (b) SIFT, Initial Partitioning, Time

(c) GIST, Partitioning Method, Time (d) GIST, Initial Partitioning, Time

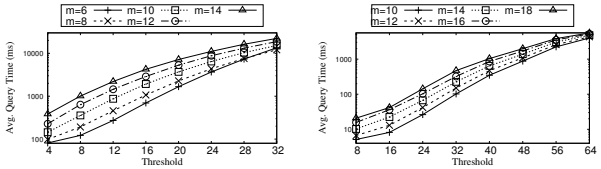
(e) PubChem, Partitioning Method, Time (f) PubChem, Initial Partitioning, Time

Fig. 4. Evaluation of Dimension Partitioning

the original unshuffled order of the dataset. (2) RS is to perform a random shuffle on the original order. (3) OS [34] and DD [30] are two dimension rearrangement methods to make dimensions in each partition uniformly distributed. We run GPH with the above partitioning methods and show the query response times in Figs. 4(a), 4(c), and 4(e). On SIFT, their performances are close. When the dataset has more skewness, the advantage of GR becomes remarkable. It is faster than the runner-up by up to 4 times on GIST and 8 times on PubChem.

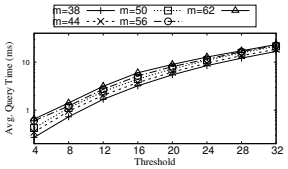
To evaluate the effect of initial partitioning, we run our partitioning algorithm with three initial states: (1) the proposed method which tries to minimize entropy (denoted GreedyInit), (2) equi-width partitioning on the original unshuffled data (denoted OriginalInit), and (3) equi-width partitioning after random shuffle (denoted RandomInit). The corresponding query response times on the three datasets are plotted in Figs. 4(b), 4(d), and 4(f). The trends are similar to the previous set of experiments. On datasets with more skewness, GreedyInit is consistently faster than the other competitors, and the gap to the runner-up can be up to 2 times.

As for the query workload $\mathcal{Q}$ to compute dimension partitioning, our results show that the effect of its size on the query processing performance is not obvious.E.g., when $\tau = 64$, the average query processing times vary from 4.19 to 3.97 seconds on GIST, if we increase $|\mathcal{Q}|$ from 100 to 1000. Thus we choose



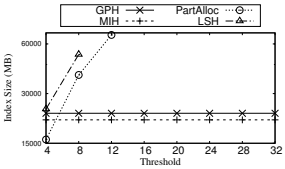
(a) SIFT, Effect of m, Time

(b) GIST, Effect of m, Time

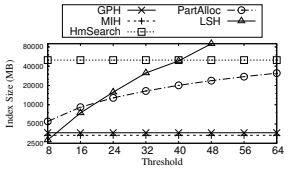


(c) PubChem, Effect of m, Time

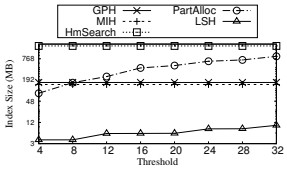
Fig. 5. Effect of Partition Number



(a) SIFT, Index Size



(b) GIST, Index Size



(c) PubChem, Index Size

Fig. 6. Comparison with Alternatives - Index Size

100 as the size of $\mathcal{Q}$ in our experiments.

We also study the effect of partition number on the query processing performance. Figs. 5(a) – 5(c) show the query response times on the three datasets by varying the number of partitions. The general trend is that a smaller m performs better under small τ settings. When τ increases, the best choice of m slightly increases. The reason is: (1) When τ is small, a small m is good enough. Dividing vectors into unnecessarily large number of partitions yields very small partitions and hence increases the frequency of signatures. (2) When τ is large, a small m means more thresholds will be allocated to a partition, and this results in more candidates. Hence a slightly larger m is better in this case. Based on the results, we suggest user choose $m \approx \frac{n}{24}$ for GPH for good query processing performance.

E. Comparison with Existing Methods

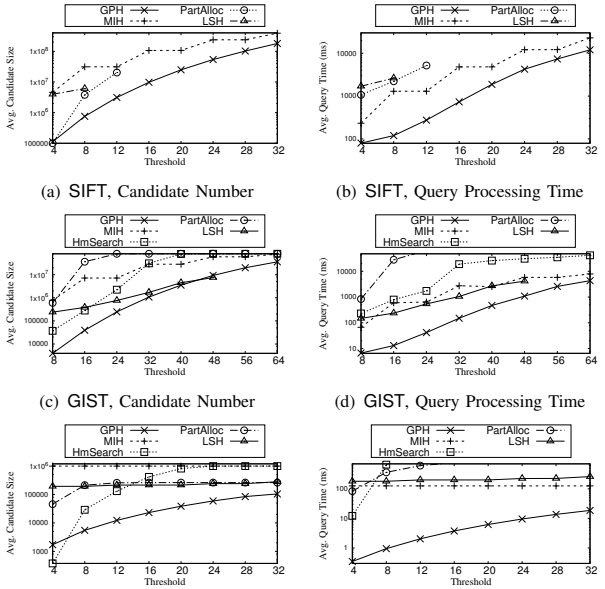
We compare GPH with alternative methods (equipped with the OS partitioning [34]) for Hamming distance search.

Index are compared first. Figs. 6(a) – 6(c) show the index sizes of the algorithms on the three datasets. LSH, HmSearch, and PartAlloc run out of memory for some τ settings on SIFT and GIST. We only show the points when the memory can hold their indexes. GPH consume more space than MIH due to the machine learning-based technique to estimate candidate numbers. Both algorithms consume less space than the other exact competitors. This is expected as GPH and MIH enumerate signatures on query vectors only. HmSearch and PartAlloc enumerate 1-deletion variants on data vectors; i.e., removing an arbitrary dimension from a partition and taking the rest as a signature. The variants are indexed and this will increase their index sizes. PartAlloc and LSH exhibit variable index sizes with respect to τ. LSH has the smallest index size on PubChem, but consumes much more space on the other two datasets. The reason is that PubChem has much more dimensions than the other two datasets. Hence given a τ, the equivalent Jaccard threshold is higher on PubChem, resulting in less number of signatures. The corresponding index construction times on GIST are shown in Table IV. LSH runs out of memory when $\tau = 64$, and thus is shown for the other τ settings. The time of GPH is decomposed into dimension partitioning and indexing. MIH spends the least amount of time on index construction. Despite more time consumption on partitioning, GPH spends less time indexing data objects than the other algorithms. We argue that the partitioning can be done offline and the time is affordable. Because the query workload $\mathcal{Q}$ for partitioning computation consists of queries with varying thresholds, we can run the partitioning once and use the same partitioning for different τ settings in real queries. This is also the reason why GPH has constant partitioning and indexing time irrespective of τ.

TABLE IV
INDEX CONSTRUCTION TIME ON GIST (S)

τ	MIH	HmSearch	PartAlloc	LSH	GPH
16	481	1681	1736	583	5026 + 560
32	481	1689	3244	5221	5026 + 560
48	481	1711	7600	64256	5026 + 560
64	481	1747	9605	N/A	5026 + 560

The candidate numbers are plotted in Figs. 7(a), 7(c), and 7(e). The corresponding query response times are plotted in Figs. 7(b), 7(d), and 7(f). For all the algorithms, candidate numbers and running times increase when τ moves towards larger values, and their trends are similar. Thanks to the tight filtering condition and cost-aware partitioning and threshold allocation, GPH is consistently smaller than MIH and HmSearch in candidate size and faster than the two methods. The only exception is that HmSearch has smaller candidate size when $\tau = 4$ on PubChem, but turns out to be slower than GPH. This is because HmSearch generates many signatures whose postings lists are empty, and this drastically increases signature enumeration and index lookup times. Although PartAlloc has a tight filtering condition and utilizes threshold allocation, it is not as fast as GPH, and even slower than MIH. This result showcases that PartAlloc's partitioning and threshold allocation is not efficient for Hamming distance search, though it pays off on set similarity search. Another interesting observation is that LSH does not perform well on highly skewed data. The reason is that the hash functions may choose highly skewed and correlated dimensions, and thus the selectively of the chosen signatures becomes very bad. On PubChem, LSH's performance is close to a sequential scan. Overall, GPH is the fastest algorithm. The speed-ups against the runner-up algorithms on the three datasets

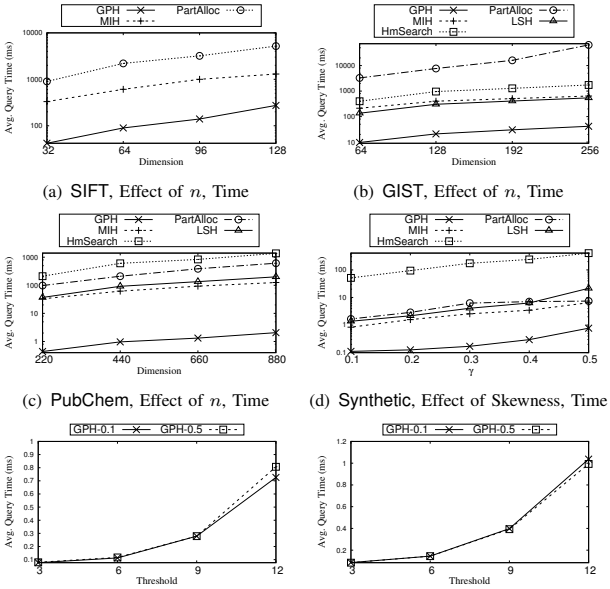


(a) SIFT, Candidate Number (b) SIFT, Query Processing Time

(c) GIST, Candidate Number (d) GIST, Query Processing Time

(e) PubChem, Candidate Number (f) PubChem, Query Processing Time

Fig. 7. Comparison with Alternatives - Candidate Number & Time



(a) SIFT, Effect of n, Time (b) GIST, Effect of n, Time

(c) PubChem, Effect of n, Time (d) Synthetic, Effect of Skewness, Time

(e) Synthetic, $\gamma_{\mathcal{D}} = 0.5$, $\gamma_q = 0.1$, Time (f) Synthetic, $\gamma_{\mathcal{D}} = 0.1$, $\gamma_q = 0.5$, Time

Fig. 8. Varying Number of Dimensions and Skewness

are up to 22, 21, and 135 times, respectively.

F. Varying Number of Dimensions

We compare the five competitors to evaluate their performances when varying the number of dimensions. We sample 25%, 50%, 75%, and 100% dimensions from the three datasets and run the experiment. $\tau = 12$, 24, and 12 for the 100% sample on the three datasets, respectively, and we let τ change linearly with the number of sampled dimensions. Figs. 8(a) – 8(c) show the query response times of the algorithms on the three datasets. We observe that the times of all the algorithms increase with n. There are two factors: (1) Although τ and n increase proportionally, the number of results increases with n due to dimension correlations. Hence we have more candidates to verify. (2) The verification cost increases with n because more dimensions are compared. Nonetheless, GPH is always the fastest algorithm among the competitors, especially on the more skewed PubChem.

G. Varying Skewness

We study the performance by varying skewness [4]. As seen from Fig. 1, the relationship between skewness and dimensions is approximately linear (except PubChem) on most datasets. On the basis of this observation, the synthetic dataset is generated as follows: The number of dimensions is 128. The mean skewness is controlled by a parameter γ, and the skewnesses of the 128 dimensions range from 0 to 2γ. We set $\tau = 12$. The query processing times are plotted in Fig. 8(d). The general trend is that all the algorithms become slower on more skewed data. This is expected as signatures become less selective. Nonetheless, thanks to variable partitioning and threshold allocation, GPH is the fastest among the five competitors.

To demonstrate the robustness of GPH, we show that even if the distribution of real queries is different from the sample to compute partitioning, our method retains good performance. We generate a synthetic dataset with a γ of 0.5, and then compute partitioning with two query workloads: $\gamma = 0.5$ (denoted GPH-0.5) and $\gamma = 0.1$ (denoted GPH-0.1), respectively. Then we run a set of queries with a γ of 0.1. The gap between GPH-0.5 and GPH-0.1 can be regarded as the extent to which GPH's performance deteriorates in the presence of a different query distribution. Then we set γ to 0.1 for the synthetic dataset and run the experiment again. Results are plotted in Figs. 8(e) – 8(f). It can be seen that although GPH computes partitioning with a workload whose distribution is different from real queries, the query processing performance is almost the same. A slight difference can be noticed only when τ is as large as 12, where the query processing speed drops by 11.1% and 4.4%, respectively.

[4]See the footnote in Section I for the measurement of dataset skewness.

VIII. Related Work

The notion of Hamming distance search was first proposed in [21]. Due to its wide range of applications, the problem has received considerable attention in the last few decades.

A few studies focused on the special case when $\tau = 1$ [3], [4], [19], [32]. Among them, the method by [19] indexes all the 1-variants of the data vectors to answer the query in $O(1)$ time and $O(\binom{n}{\tau})$ space. A data structure was proposed in [4] to answer this special case in $O(1)$ time using $O(n \log m)$ space by a cell probe model with word size m.

For the general case of Hamming distance search, the method by [9] is able to answer Hamming distance search in $O(m + \log^{\tau}(nm) + occ)$ time and $O(n \log^{\tau}(nm))$ space, where occ is the number of results. In practice, many solutions are based on the pigeonhole principle to convert the problem to sub-problems with a threshold τ', where $\tau' < \tau$. In [27], [16], [23], vectors are divided into a number of partitions such that query results must have at least one exact match with the query in one of the

partitions. The idea of recursive partitioning was covered in [20]. Before that, a two-level partitioning idea was adopted by the PartEnum method [1]. Song *et al*. [26] proposed to enumerate the combinations within threshold τ' in each partition to avoid the verification of candidates. Ong and Bober [24] proposed an approximate method utilizing variable length hash keys. In [34], vectors are divided into $\lfloor \frac{\tau+3}{2} \rfloor$ partitions, and the threshold of a partition can be either 0 or 1. Deng *et al*. [10] also proposed to use different thresholds on partitions, including -1, 0, and 1, and the thresholds are computed by the allocation algorithm.

To handle the poor selectivity caused by data skewness and dimension correlations, existing work mainly focused on two strategies. The first is to perform a random shuffle [1] in original dimensions to avoid highly correlated dimensions in same partitions. The second is to perform a dimension rearrangement [34], [30], [18] to minimize the correlation between dimensions in each partition. These methods are able to answer queries efficiently on slightly skewed datasets, but the performances deteriorate on highly skewed datasets.

We note that a strong form of the pigeonhole principle was introduced in [6] which states that given n positive integers $q_1, \ldots, q_m$, if $(\sum_{i=1}^{m} q_i - m + 1)$ objects are distributed into m boxes, then either the first box contains at least q_1 objects, ..., or the n-th box contains at least q_n objects. Although the general pigeonhole principle proposed in this paper coincides with the above strong form, by integer reduction and ϵ-transformation, the general pigeonhole principle is not limited to positive integers (this is the reason why GPH performs well on skewed data) and the tightness of threshold allocation is proved, hence providing a deeper understanding of the pigeonhole principle.

IX. Conclusion and Future Work

In this paper, we proposed a new approach to similarity search in Hamming space. Observing the major drawbacks of the basic pigeonhole principle adopted by many existing methods, we developed a new form of the pigeonhole principle, based on which the condition of candidate generation is tight. The cost of query processing was modeled, and then an offline dimension partitioning algorithm and an online threshold allocation algorithm were devised on top of the model. We conducted experiments on real datasets with various distributions, and showed that our approach performs consistently well on all these datasets and outperforms state-of-the-art methods.

Our future work includes extending general pigeonhole principle to other similarity constraints. Another direction is to explore the techniques to dealing with the parallel case.

Acknowledgements. J. Qin, Y. Wang, and W. Wang are partially supported by ARC DP170103710, and D2DCRC DC25002 and DC25003. C. Xiao and Y. Ishikawa are supported by JSPS Kakenhi 16H01722. X. Lin is supported by NSFC 61672235, ARC DP170101628 and DP180103096. We thank the authors of [10] for kindly providing their source codes.

References

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[2] J. M. Borwein and D. H. Bailey. *Mathematics by experiment - plausible reasoning in the 21st century*. A K Peters, 2003.

[3] G. S. Brodal and L. Gasieniec. Approximate dictionary queries. In *CPM*, pages 65–74, 1996.

[4] G. S. Brodal and S. Venkatesh. Improved bounds for dictionary look-up with one error. *Inf. Process. Lett.*, 75(1-2):57–59, 2000.

[5] A. Z. Broder. On the resemblance and containment of documents. In *SEQS*, 1997.

[6] R. Brualdi. *Introductory Combinatorics*. Math Classics. Pearson, 2017.

[7] Z. Cao, M. Long, J. Wang, and P. S. Yu. Hashnet: Deep learning to hash by continuation. In *ICCV*, pages 5609–5618, 2017.

[8] S. Chaidaroon and Y. Fang. Variational deep semantic hashing for text documents. In *SIGIR Conference*, pages 75–84, 2017.

[9] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.

[10] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.

[11] D. R. Flower. On the properties of bit string-based measures of chemical similarity. *Journal of Chemical Information and Computer Sciences*, 38(3):379–386, 1998.

[12] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. *CoRR*, abs/1102.3828, 2011.

[13] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.

[14] W. Li, Y. Zhang, Y. Sun, W. Wang, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement (v1.0). *CoRR*, abs/1610.02455, 2016.

[15] K. Lin, H. Yang, J. Hsiao, and C. Chen. Deep learning of binary hash codes for fast image retrieval. In *CVPR Workshops*, pages 27–35, 2015.

[16] A. X. Liu, K. Shen, and E. Torng. Large scale hamming distance query processing. In *ICDE*, pages 553–564, 2011.

[17] H. Liu, R. Wang, S. Shan, and X. Chen. Deep supervised hashing for fast image retrieval. In *CVPR Conference*, pages 2064–2072, 2016.

[18] Y. Ma, H. Zou, H. Xie, and Q. Su. Fast search with data-oriented multi-index hashing for multimedia data. *TIIS*, 9(7):2599–2613, 2015.

[19] U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *Inf. Process. Lett.*, 50(4):191–197, 1994.

[20] G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *WWW*, pages 141–150, 2007.

[21] M. Minsky and S. Papert. *Perceptrons - an introduction to computational geometry*. MIT Press, 1987.

[22] R. Nasr, R. Vernica, C. Li, and P. Baldi. Speeding up chemical searches using the inverted index: The convergence of chemoinformatics and text search methods. *J. Chem. Inf. Model*, 2012.

[23] M. Norouzi, A. Punjani, and D. J. Fleet. Fast search in hamming space with multi-index hashing. In *CVPR*, pages 3108–3115, 2012.

[24] E. Ong and M. Bober. Improved hamming distance search using variable length hashing. In *CVPR Conference*, pages 2000–2008, 2016.

[25] H. Park and L. Stefanski. Relative-error prediction. *Statistics & Probability Letters*, 40(3):227 – 236, 1998.

[26] J. Song, H. T. Shen, J. Wang, Z. Huang, N. Sebe, and J. Wang. A distance-computation-free search scheme for binary code databases. *IEEE Trans. Multimedia*, 18(3):484–495, 2016.

[27] Y. Tabei, T. Uno, M. Sugiyama, and K. Tsuda. Single versus multiple sorting in all pairs similarity search. *Journal of Machine Learning Research - Proceedings Track*, 13:145–160, 2010.

[28] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(11):1958–1970, 2008.

[29] S. A. Vinterbo. A note on the hardness of the k-ambiguity problem. Technical report, Harvard Medical School, 06 2002.

[30] J. Wan, S. Tang, Y. Zhang, L. Huang, and J. Li. Data driven multi-index hashing. In *ICIP Conference*, pages 2670–2673, 2013.

[31] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014.

[32] A. C.-C. Yao and F. F. Yao. Dictionary look-up with one error. *J. Algorithms*, 25(1):194–202, 1997.

[33] W. Zhang, K. Gao, Y. Zhang, and J. Li. Efficient approximate nearest neighbor search with integrated binary codes. In *ICMM Conference*, pages 1189–1192, 2011.

[34] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu. Hmsearch: an efficient hamming distance query processing algorithm. In *SSDBM*, page 19, 2013.