Generalizing the Pigeonhole Principle for Similarity Search in Hamming Space

Jianbin Qin, Chuan Xiao† , Yaoshu Wang, Wei Wang,
Xuemin Lin, Fellow, IEEE, Yoshiharu Ishikawa, Member, IEEE, and Guoren Wang, Member, IEEE

Abstract—A distance 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, such as image retrieval, near-duplicate Web page detection, and scientific databases. State-of-the-art approaches to Hamming distance search are mainly based on the pigeonhole principle to generate a set of candidates and then verify them. We observe that the constraint by the pigeonhole principle is not always tight and 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 parts, hence failing to exploit the data skewness to optimize query processing. In this paper, we propose a new form of the pigeonhole principle which allows variable partitioning and threshold allocation. Based on the new principle, we develop a tight constraint of candidates and devise cost-aware methods for partitioning and threshold allocation to optimize query processing. In addition, we extend our methods to answer Hamming distance join queries. We also discuss the application of the pigeonhole principle in set similarity search, a problem that can be converted to Hamming distance search equivalently. 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.

Index Terms—Hamming distance, similarity search, pigeonhole principle.

1 Introduction

Finding similar objects is a fundamental problem in database research and has been studied for several decades [47]. Among many types of queries to find similar objects, Hamming distance search on binary vectors is an important one. Given a query $\mathbf{q}$, a Hamming distance search finds all the vectors in a dataset whose Hamming distances to $\mathbf{q}$ 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 [52]. Recently, deep learning has become remarkably successful in image recognition. Learning to hash algorithms that utilize neural networks have been actively explored [10], [23], [25]. 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 [11]. Google converts Web pages into 64-bit vectors and uses Hamming distance search to detect near-duplicate Web pages [28].
- For scientific databases, a fundamental task in cheminformatics is to find similar molecules [17], [31]. 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 [53].

The naïve algorithm to answer a Hamming distance search query requires access and computation 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 [2], [24], [33], [53] 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 of the vectors are vertically partitioned into m *equi-width* parts (we assume $n \bmod m = 0$ in this paper), 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 part is not always tight**. Many unnecessary candidates are included in consequence. 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 produce the same set of candidates.

(2) **The thresholds on the m parts are evenly distributed**. It assumes a uniform distribution and does not work well

† *Corresponding author.*

- *J. Qin and Y. Wang are with Shenzhen Institute of Computing Sciences, Shenzhen University, China. E-mail: {jqin, yaoshuw}@sics.ac.cn.*
- *C. Xiao is with the Graduate School of Information Science and Technology, Osaka University, and the Graduate School of Informatics, Nagoya University, Japan. E-mail: chuanx@ist.osaka-u.ac.jp.*
- *W. Wang and X. Lin are with the School of Computer Science and Engineering, the University of New South Wales, Australia. E-mail: {weiw, lxue}@cse.unsw.edu.au.*
- *Y. Ishikawa is with the Graduate School of Informatics, Nagoya University, Japan. E-mail: ishikawa@i.nagoya-u.ac.jp.*
- *G. Wang is with the School of Computer Science & Technology, Beijing Institute of Technology, China. E-mail: wanggrbit@126.com.*

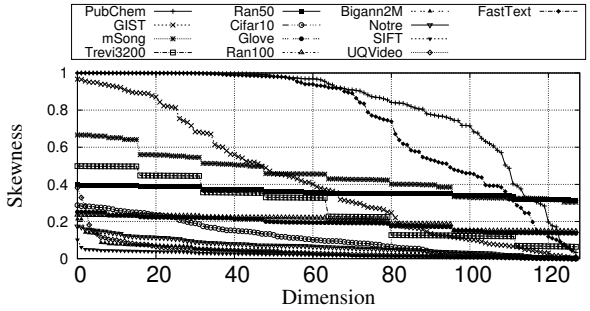


Fig. 1. Skewness ($\frac{|\#1s-\#0s|}{\#vectors}$) by dimension of datasets in [22].

when the dataset is skewed. We find that many real datasets are skewed to varying degrees and complex correlations exist among dimensions. Fig. 1 shows that 8 out of 13 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 part, meaning that at least $1/10$ data vectors become candidates if the query matches the data vector on this part.

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 parts 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 part is a *variable* in the range of $[-1\,..\,\tau]$, where -1 indicates that this part is ignored when generating candidates. This enables us to choose proper thresholds for different parts 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 part cannot be further reduced.

To tackle data skewness and dimension correlations, we first devise an online algorithm to allocate thresholds to m parts 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. We also extend the algorithm to answer the Hamming distance join queries that finds pairs of vectors in a dataset (or two datasets) whose Hamming distances are no greater than a threshold τ. In addition, we discuss the relationship between Hamming distance search and set similarity search as they can be converted to each other and the pigeonhole principle has been utilized to solve the latter. The discussion reveals the fact that the prefix filter, a prevalent approach to set similarity search, is an extension of the general pigeonhole principle. Experiments are run on several real datasets with different data distributions. The results show that the GPH algorithm performs consistently well on 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 for Hamming distance search. (Section 3). (2) We propose an efficient online query optimization method to allocate thresholds on the basis of the new pigeonhole principle (Section 4). (3) We propose an adaptive partitioning method to address the selectivity issue caused by data skewness and dimension correlations (Section 5). (4) We extend our algorithm to answer Hamming distance join queries (Section 7). (5) We discuss the relationship with set similarity search and the application of the (general) pigeonhole principle (Section 8). (6) We conduct extensive experimental study on several real datasets to evaluate the proposed method (Section 9). The results demonstrate the superiority of the proposed method over state-of-the-art methods. Compared to the conference version of this paper [37], Sections 7, 8, and Section 9 (partially) are new materials.

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.

2 Preliminaries

2.1 Problem Definition

An object is represented by an n-dimensional binary vector $\mathbf{x}$. $\mathbf{x}[i]$ denotes the value of the i-th dimension of $\mathbf{x}$. Let $\Delta(\mathbf{x}[i], \mathbf{y}[i]) = 0$, if $\mathbf{x}[i] = \mathbf{y}[i]$; or 1, otherwise. The Hamming distance between two vectors $\mathbf{x}$ and $\mathbf{y}$, denoted by $H(\mathbf{x}, \mathbf{y})$, is the number of dimensions on which $\mathbf{x}$ and $\mathbf{y}$ differ:

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

Hamming distance is a symmetric measure. For brevity, we also say there are $H(\mathbf{x}, \mathbf{y})$ errors between $\mathbf{x}$ and $\mathbf{y}$. The Hamming distance search problem is defined as follows.

Problem 1 (Hamming Distance Search)**.** *Given a collection of data objects $\mathcal{R}$, a query object $\mathbf{q}$, a Hamming distance search is to find all the objects in $\mathcal{R}$ whose Hamming distances to $\mathbf{q}$ are no greater than a threshold τ, i.e., $\{\, \mathbf{x} \mid \mathbf{x} \in \mathcal{R}, H(\mathbf{x}, \mathbf{q}) \leq \tau \,\}$.*

2.2 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 [2], [24], [33], [53] are based on the intuition that if two vectors are similar, there will be a pair of similar parts from the two vectors which are vertically partitioned into m parts. The (basic) pigeonhole principle is utilized by these methods.

Lemma 1 (Basic Pigeonhole Principle)**.** *Binary vectors $\mathbf{x}$ and $\mathbf{y}$ are vertically partitioned into m parts, each having $\frac{n}{m}$ dimensions. Let $\mathbf{x}_i$ ($\mathbf{y}_i$), $1 \leq i \leq m$, denote a part in $\mathbf{x}$ ($\mathbf{y}$). If $H(\mathbf{x}, \mathbf{y}) \leq \tau$, there exists at least one part i such that $H(\mathbf{x}_i, \mathbf{y}_i) \leq \lfloor \frac{\tau}{m} \rfloor$.*

Any data object $\mathbf{x}$ satisfying the condition that $\exists i$, $H(\mathbf{x}_i, \mathbf{q}_i) \leq \lfloor \frac{\tau}{m} \rfloor$ is called a *candidate*. Since these candidates will be verified by computing the exact Hamming distance to the query, the query processing performance depends heavily on the number of candidates.

2.3 Overview of Existing Approaches

We briefly introduce a state-of-the-art method, Multi-index Hamming (MIH) [33]; other methods based on the basic pigeonhole principle work in a similar way. MIH partitions the n dimensions into m equi-width parts. In each part, 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 part of a data object to the object ID. For each part of the query, it enumerates n'-dimensional vectors whose Hamming distances to the part are within τ'. These vectors are called *signatures*. Then it looks up signatures in the index to find candidates and verifies them.

2.4 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 an array of thresholds allocated to each corresponding part. We call the array *threshold array*, and denote the one based on the basic pigeonhole principle by $T_{\text{basic}} = [\lfloor \frac{\tau}{m} \rfloor, \ldots, \lfloor \frac{\tau}{m} \rfloor]$. We also define the *dominance* relationship between threshold arrays. Let n_i denote the number of dimensions in the i-th part. T_1 dominates T_2, or $T_1 \prec T_2$, iff $\forall i \in [1\,..\,m]$, $T_1[i] \leq T_2[i]$ and $[T_1[i], T_2[i]] \cap [-1, n_i - 1] \neq \emptyset$ [2], and $\exists i \in [1\,..\,m], T_1[i] < T_2[i]$.

- T_{basic} is **not always tight**. By the tightness of a threshold array 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 array 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 array 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 array T_{basic} is $[3, 3, 3]$. We can find a dominating threshold array $T = [2, 2, 3]$ which is tight and guarantees both correctness and minimality. Note that there may be multiple tight threshold arrays for the same τ. E.g., another tight threshold array for the example can be $[2, 3, 2]$ or $[4, 3, 0]$* [3].

- The filtering condition does **not adapt to the data distribution** in the partition. 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 parts, hence excessive number of candidates. Several recent studies recognized this issue and proposed several methods to either obtain relatively less skew parts by partition rearrangement [53] or allocating varying thresholds heuristically to different parts [15]. In contrast, we propose that a skewed partition can be beneficial and we can reduce the candidate size by judiciously allocating different thresholds to different parts in a *query-specific* way 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 vector, and two different partitions in Table 1. Consider the first query. MIH uses $T_{basic} = [1, 1]$. This results in all the four data vectors recognized as candidates, but only one ($\mathbf{x}^1$) is the result. If we use the first six dimensions as one part and the rest two dimensions as the other part, and use $T = [2, 0]$, the candidate size will be reduced to 2 ($\mathbf{x}^1$ and $\mathbf{x}^2$).*

2. This is to make sure at least one threshold is non-trivial.
3. Please refer to Section 3 for more explanation of tightness.

TABLE 1
Benefits of Adaptive Partitioning and Thresholding

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

3 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 arrays.

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

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

Proof. Assume that $\nexists i$ such that $H(\mathbf{x}_i, \mathbf{y}_i) \leq \tau_i$. Since the m parts are disjoint, $H(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{m} H(\mathbf{x}_i, \mathbf{y}_i) > \sum_{i=1}^{m} \tau_i$. Therefore, $H(\mathbf{x}, \mathbf{y}) > \tau$, which contradicts that $H(\mathbf{x}, \mathbf{y}) \leq \tau$. □

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 parts. Intuitively, we may tolerate more errors for selective parts and fewer errors for unselective parts.

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

Lemma 3. *$\mathbf{x}$ and $\mathbf{y}$ are partitioned by $\mathcal{P}$ into m disjoint parts. Consider an array $T = [\tau_1, \ldots, \tau_m]$ in which the thresholds are real numbers. $\|T\|_1 = \tau$. If $H(\mathbf{x}, \mathbf{y}) \leq \tau$, there exists at least one part i such that $H(\mathbf{x}_i, \mathbf{y}_i) \leq \lfloor \tau_i \rfloor$.*

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

Definition 1 (Integer Reduction)**.** *Given a threshold array $T = [\tau_1, \tau_2, \ldots, \tau_m]$, we can reduce it to $T' = [\lfloor \tau_1 \rfloor, \lfloor \tau_2 \rfloor, \ldots, \lfloor \tau_m \rfloor]$.*

It is obvious that the candidate set 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 array 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 array, we can start with a possible threshold array $T = [2.9, 2.9, 3.2]$. Then by the integer reduction, T is reduced to $T' = [2, 2, 3]$. To see this is correct, if $\nexists i$,*

$H(\mathbf{x}_i, \mathbf{y}_i) \leq T'[i]$, *there will be* $3+3+4=10$ *errors between* $\mathbf{x}$ *and* $\mathbf{y}$. *Compared to* $[3,3,3]$, T' *is a dominating threshold array, and the constraints on the first two parts are stricter.*

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

Lemma 4 (General Pigeonhole Principle)**.** $\mathbf{x}$ *and* $\mathbf{y}$ *are partitioned by* $\mathcal{P}$ *into* m *disjoint parts. Consider a threshold array* $T = [\tau_1, \ldots, \tau_m]$ *composed of integers.* $\|T\|_1 = \tau - m + 1$. *If* $H(\mathbf{x}, \mathbf{y}) \leq \tau$, *there exists at least one part* i *such that* $H(\mathbf{x}_i, \mathbf{y}_i) \leq \tau_i$.

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

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 an array $T'' = [\tau''_i, \ldots, \tau''_m] = [\tau_1 + 1 - \epsilon, \ldots, \tau_{m-1} + 1 - \epsilon, \tau_m + (m-1)\epsilon]$. Because $\|T''\|_1 = \|T'\|_1 = \tau$, by Lemma 3, if $H(\mathbf{x}, \mathbf{y}) \leq \tau$, then $\exists i$, $H(\mathbf{x}_i, \mathbf{y}_i) \leq \lfloor \tau''_i \rfloor$. Because

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

if $H(\mathbf{x}, \mathbf{y}) \leq \tau$, then $\exists i$, $H(\mathbf{x}_i, \mathbf{y}_i) \leq \tau_i$. □

One may notice that in the above proof, the parts we choose to decrease thresholds are not limited to the first $(m-1)$ ones. Therefore, given a threshold array T such that $\|T\|_1 = \tau$, we may choose any $(m-1)$ parts and decrease their thresholds by 1. For the resulting array 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 array T based on the general pigeonhole principle, i.e., $\|T\|_1 = \tau - m + 1$, we consider a threshold array T' which is composed of integers and dominates T, i.e., $\forall i \in [1\,..\,m]$, $T'[i] \leq T[i]$ and $[T'[i], T[i]] \cap [-1, n_i - 1] \neq \emptyset$, and $\exists j \in [1\,..\,m]$, $T'[j] < T[j]$. Because $\forall i \in [1\,..\,m]$, $H(\mathbf{x}_i, \mathbf{q}_i) \in [0\,..\,n_i]$ and $[T'[i], T[i]] \cap [-1, n_i - 1] \neq \emptyset$, we may construct a vector $\mathbf{x}$ such that $\forall i \in [1\,..\,m]$, $H(\mathbf{x}_i, \mathbf{q}_i) = \max(0, T'[i] + 1)$. $\forall i \in [1\,..\,m]$, because $T'[i] \leq T[i]$ and $[T'[i], T[i]] \cap [-1, n_i - 1] \neq \emptyset$, $H(\mathbf{x}_i, \mathbf{q}_i) \leq T[i] + 1$. Because $\exists j \in [1\,..\,m]$, $T'[j] < T[j]$, $\exists j \in [1\,..\,m]$, $H(\mathbf{x}_j, \mathbf{q}_j) \leq T[j]$. Because $H(\mathbf{x}_i, \mathbf{q}_i) > T'[i]$ on all the m parts, x is not a candidate by T'. However, $H(\mathbf{x}, \mathbf{q}) = \sum_{i=1}^{m} H(\mathbf{x}_i, \mathbf{q}_i) \leq \|T\|_1 + m - 1 = \tau$, meaning that $\mathbf{x}$ 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 part may become *negative*. For example, $[1,0,0]$ becomes $[0,0,-1]$[4] if the first and third parts are chosen to decrease thresholds. Since $H(\mathbf{x}_i, \mathbf{y}_i)$ is a non-negative integer, $H(\mathbf{x}_i, \mathbf{y}_i) \leq \tau_i$ is always false if τ_i is negative. This fact indicates that the parts with negative thresholds can be **safely ignored** for candidate generation. As will be shown in the next section, this allows us to ignore the parts where the query vector and most of the data vectors 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 2. For* $\mathbf{q}^1$, *we show the threshold arrays based on the flexible pigeonhole principle and the general pigeonhole principle. The candidate sizes are 2 and 1, respectively. For* $\mathbf{q}^2$, *we show two different threshold arrays based on the general pigeonhole principle. The candidate sizes are 4 and 2, respectively.*

TABLE 2
Threshold Array and Candidate Size

	Part 1	Part 2
$\mathbf{x}^1 = 00000000$	000000	00
$\mathbf{x}^2 = 00000111$	000001	11
$\mathbf{x}^3 = 00001111$	000011	11
$\mathbf{x}^4 = 10011111$	100111	11
$\mathbf{q}^1 = 10000000$	100000	00
$\mathbf{q}^2 = 10000011$	100000	11

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

4 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 parts, and (2) how to compute the threshold array T such that $\|T\|_1 = \tau - m + 1$. We will tackle the first issue in Section 5 with an offline solution. Before that, we focus on the second issue in this section and propose an online algorithm.

4.1 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 part of a data object to the object ID. Then for each part of the query, we enumerate signatures to generate candidates.

The query processing cost consists of three parts:

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

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

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

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

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

where n_i denotes the number of dimensions in the i-th part, 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 part.

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

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

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

The verification cost is

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

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 9.2 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 [44], which is proved to be NP-hard. Nonetheless, $|S_{cand}|$ is upper-bounded by the sum of candidates generated in all the parts, i.e., $\sum_{\mathbf{s} \in S_{sig}} |I_{\mathbf{s}}|$. Our experiments (Section 9.2) 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 $\alpha \cdot \sum_{i=1}^{m} CN(\mathbf{q}_i, \tau_i)$, where $CN(\mathbf{q}_i, \tau_i)$ is the number of candidates generated by the i-th part of the query $\mathbf{q}$ with a threshold of τ_i (when $\tau_i = -1$, $CN(\mathbf{q}_i, \tau_i) = 0$). The query processing cost can be estimated as:

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

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

Problem 2 (Threshold Allocation)**.** *Given a collection of data objects $\mathcal{R}$, a query $\mathbf{q}$ and a threshold τ, find the threshold array T that minimizes the estimated query processing cost under the general pigeonhole principle; i.e.,*

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

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;
```

4.2 Threshold Allocation Algorithm

Since c_{access}, c_{verify}, and α are independent of $CN(\mathbf{q}_i, \tau_i)$, we can omit the coefficient $(c_{access} + \alpha \cdot c_{verify})$ in Equation 1 and find the minimum query processing cost with only $CN(\mathbf{q}_i, \tau_i)$. The computation of $CN(\mathbf{q}_i, \tau_i)$ values will be introduced in Section 4.3. Here we treat $CN(\mathbf{q}_i, \tau_i)$ 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 the parts $1, \ldots, i$ 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 part (Lines 1 – 2), i.e., $OPT[1, -1], \ldots, OPT[1, \tau]$. Then it iterates through the other parts and compute the minimum costs (Lines 3 – 10). Note that the negative threshold -1 is also considered for each part. Finally, we trace the path that reaches $OPT[m, \tau - m + 1]$ to obtain the threshold array (Lines 11 – 14). The time complexity of the algorithm is $O(m \cdot (\tau + 1)^2)$.

Example 5. *Consider a dataset of 100 binary vectors and we partition them into 4 parts. Given a query $\mathbf{q}$, for each part i, suppose the numbers of candidates (denoted by CN_i) under different thresholds are given 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 array. 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 (underlined) that reaches this value and obtain the threshold array* $[2,0,2,0]$.

4.3 Computing Candidate Numbers

In order to run the threshold allocation algorithm, we need to obtain the candidate numbers $CN(\mathbf{q}_i, \tau_i)$ beforehand. An exact solution to computing $CN(\mathbf{q}_i, \tau_i)$ is to enumerate all possible vectors for the i-th part and then count how many vectors in $\mathcal{R}$ has a Hamming distance within τ_i to the enumerated vector in this part. These numbers are stored in a table. When processing the query, with the given $\mathbf{q}_i$, the table is looked up for the corresponding entry $CN(\mathbf{q}_i, \tau_i)$. 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 $\mathbf{q}_i$ into smaller equi-width sub-parts and estimating $CN(\mathbf{q}_i, \tau_i)$ with the candidate numbers of the sub-parts. We divide $\mathbf{q}_i$ into m_i sub-parts. Each sub-part 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-parts, 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 $\mathbf{q}_{ij}$ denote a sub-part of $\mathbf{q}_i$ and τ_{ij} denote its threshold. Let $G(m_i, \tau_i)$ be the set of threshold arrays 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 $CN(\mathbf{q}_{ij}, \tau_{ij})$ 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 $\mathcal{R}$ has a Hamming distance within τ_{ij} to the enumerated vector in this sub-part. We assume that the candidates in the m_i sub-parts are independent. Then $CN(\mathbf{q}_i, \tau_i)$ can be approximately estimated online with the following equation.

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

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

Let $h_{\tau_i}(\mathbf{x}_i, \theta_i)$ 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 $CN(\mathbf{x}_k, \tau_i)$, we use relative error as our loss function: $J(\mathcal{T}_i, \theta_i) = \sum_{k=1}^{|\mathcal{T}_i|} \{ \frac{CN(\mathbf{x}_k, \tau_i) - h_{\tau_i}(\mathbf{x}_k, \theta_i)}{CN(\mathbf{x}_k, \tau_i)} \}^2$. According to [36], we utilize the approximation $ln(t) \approx t-1$ to estimate $J(\mathcal{T}_i, \theta_i)$:

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

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

5 Dimension Partitioning

To deal with data skewness and dimension correlations, the existing methods for Hamming distance search resort to random shuffle [2] or dimension rearrangement [26], [45], [53]. All of them are aiming towards the direction that the dimensions in each part 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 an algorithm to solve this problem.

5.1 Cost Model

Let P_i denote a set of dimensions in the range $[1\,..\,n]$. Our goal is to find a partition $\mathcal{P} = \{\, P_1, \ldots, P_m \,\}$ 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 $\mathcal{Q} = \{ < \mathbf{q}^1, \tau^1 >, \ldots, < \mathbf{q}^{|\mathcal{Q}|}, \tau^{|\mathcal{Q}|} > \}$, the query processing cost of $\mathcal{Q}$ is the sum of the costs of its constituent queries:

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

where $\widehat{C_{query\_proc}}(\mathbf{q}^i, \tau^i, \mathcal{P})$ is the processing cost of query $\mathbf{q}^i$ with a threshold τ^i, which can be computed using the dynamic programming algorithm proposed in Section 4. Then we can formulate the dimension partitioning as an optimization problem.

Problem 3 (Dimension Partition)**.** *Given a collection of data objects* $\mathcal{R}$ *and a query workload* $\mathcal{Q}$*, find the partition* $\mathcal{P}$ *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 partition problem is NP-hard.*

Proof. We can reduce the dimension partition problem from the number partition problem [4], 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 the two sets

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

```
1  P ← InitialPartition(R, 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;
```

is minimized. Consider a special case $m = 2$ and a $\mathcal{Q}$ of only one query. Let S be a multiset of n positive integers, each representing a dimension in the dimension partition problem. Let $sum(S)$ denote the sum of numbers in S. For $i \in \{1, 2\}$, Let $CN(\mathbf{q}_i, \tau_i) = sum(S_i)^2$, $\forall \tau_i \in [-1 \,..\, \tau]$; i.e., the candidate number in part i equals to the square of the sum of numbers in this part. 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. The special case of the dimension partition problem is thus reduced from the number partition problem. Because the latter is NP-complete, the dimension partition problem is NP-hard. □

5.2 Partitioning Algorithm

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

Algorithm 2 captures the pseudo-code of the heuristic partitioning algorithm. It first generates an initial partition $\mathcal{P}$ of m parts (Line 1). The details of the initialization step will be introduced in Section 5.3. Then the algorithm iteratively improves the current partition by selecting the best option of moving a dimension from one part to another. In each iteration, we pick a dimension from a part P_i (Line 8), try to move it to another part 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, a number of parts may become empty in our algorithm. Hence it is not mandatory to output exactly m parts for an input partition size 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 partition, our query processing algorithm still achieves good performance (Section 9.7). 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 partition using the workload which covers a wide range of thresholds, and then build an index without being aware of the thresholds of real queries beforehand.

5.3 Initial Partitioning

Since the dimension partition algorithm stops at a local optimum, we may achieve a better result with a carefully selected initial partition. The correlation of dimensions play an important role here. Unlike the existing methods which try to make dimensions in each part 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 parts. 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 part, and thus our threshold allocation algorithm can assign a larger threshold to this part and smaller thresholds to the other parts; i.e., choosing proper thresholds for different parts. If the dimensions are uniformly distributed, all the parts will have the same distribution and there is little chance to optimize for specific parts.

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

$$H(\mathcal{R}_{P_i}) = -\sum_{\mathbf{x} \in \mathcal{R}_{P_i}} P(\mathbf{x}) \cdot \log P(\mathbf{x}).$$

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 partition $\mathcal{P}$ is the sum of the entropies of its constituent parts:

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

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

6 The GPH Algorithm

Based on the general pigeonhole principle and the techniques proposed in Sections 4 and 5, 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: the indexing phase (Algorithm 3) and the query processing phase (Algorithm 4). In the indexing phase, it takes as input a dataset $\mathcal{R}$, a query workload $\mathcal{Q}$,and a tunable parameter m for the size

Algorithm 3: GPH-Index($\mathcal{R}, \mathcal{Q}, m$)

```
1 P ← HeuristicPartition(R, Q, m);
2 I ← ∅;      /* I is a hashmap of key-value pair
    ⟨⟨signature,part ID⟩,postings list⟩ */;
3 foreach x ∈ R do
4     for i ← 1 to m do
5         I_{x_i} ← I_{x_i} ∪ { ⟨x, i⟩ };
6 return I;
```

Algorithm 4: GPH-ProcessQuery($\mathcal{R}, I, \mathcal{P}, \mathbf{q}, \tau$)

```
1  A ← ∅, R ← ∅;  /* A is a set of candidates */;
2  T ← DPAllocate(q, m, τ);
3  for i ← 1 to m do
4      foreach s s.t. H(s, x_i) ≤ T[i] do
5          foreach ⟨x, j⟩ ∈ I_s and i = j do
6              A ← A ∪ { x };
7  foreach x ∈ A do
8      if H(x, q) ≤ τ then
9          R ← R ∪ x;
10 return R;
```

of partition. The partition $\mathcal{P}$ is generated using the heuristic partitioning algorithm proposed in Section 5 (Line 1). Then for each n-dimensional vector $\mathbf{x}$ in $\mathcal{R}$, we partition it by $\mathcal{P}$ into m parts. For the projection of $\mathbf{x}$ on each part, the ID of $\mathbf{x}$, along with the part ID, is inserted into the postings list of the projection (Line 5). In the query processing phase, the query $\mathbf{q}$ and the threshold τ are input to the algorithm. It first partitions $\mathbf{q}$ by $\mathcal{P}$ into m parts. Then the threshold array T is computed using the dynamic programming algorithm proposed in Section 4 (Line 2). For the projection of $\mathbf{q}$ on each part, we enumerate the signatures whose Hamming distances to the projection do not exceed the allocated threshold (Line 4). Then for each signature, we probe the inverted index to find the data objects that have this signature in the same part (Line 5), and insert their object IDs into the candidate set (Line 6). The candidates are verified using Hamming distance (Line 8) and the results are returned (Line 10).

7 Hamming Distance Join

Hamming distance searches can be invoked in batch mode and become a Hamming distance join.

Problem 4 (Hamming Distance Join)**.** *Given two collection of data objects $\mathcal{R}$ and $\mathcal{S}$, a Hamming distance join is to find all pairs of objects in the two collections whose Hamming distances are no greater than a threshold τ, i.e., $\{ \langle \mathbf{x}, \mathbf{y} \rangle \mid \mathbf{x} \in \mathcal{R}, \mathbf{y} \in \mathcal{S}, H(\mathbf{x}, \mathbf{y}) \leq \tau \}$.*

Hamming distance join is useful for the applications in which we are to identify similar objects from multiple datasets, e.g., detecting suspects from a set of camera snapshots using a repository of the photos of recorded criminals. For such applications, a common scenario is that the dimension partition and the index are sometimes not given beforehand, i.e., the whole task is processed online. We assume such setting for Hamming distance join. In the rest of this section, we first adapt the GPH algorithm to self joins (i.e., $\mathcal{R} = \mathcal{S}$ and $\mathbf{x}.ID < \mathbf{y}.ID$) and then extend the method to process R-S joins.

7.1 Self Join

The self join algorithm is composed of two passes over $\mathcal{R}$. The first pass is to compute the dimension partitioning. The second pass is to perform the join with an index constructed online.

For dimension partitioning, in Section 5 we propose to find an initial partition using entropy and then refine it with a heuristic algorithm. Since the heuristic refinement runs in multiple iterations to compute the cost of the workload and adjust the partition, it is time-consuming and not suitable for an online join task. Thus, we take the initial partition as the result of dimension partitioning for Hamming distance join.

Then we iterate through $\mathcal{R}$ to perform the join (pseudo-code given in Algorithm 5). The inverted index, the statistics for candidate number computation, and the result set are initialized as empty at first (Line 1). Then each object $\mathbf{x}$ is partitioned and processed in two steps:

- The first step is a Hamming distance search for the already-seen objects that are similar to $\mathbf{x}$. We first partition $\mathbf{x}$ and allocate thresholds (Line 3). In Section 4.3, two candidate number computation methods are proposed for threshold allocation. We choose the sub-partitioning method to compute the candidate number for Hamming distance join, because the machine learning method requires an offline training and does not work for an index built online since the number of indexed objects is growing. Then we follow the same method as Hamming distance search to find similar objects to $\mathbf{x}$ (Lines 4 – 11).
- In the second step, we first insert $\mathbf{x}$ to the inverted index (Line 13). Then we update the candidate number statistics (the CN values) by sub-partitioning (Lines 14 – 17): for each $\tau_{ij} \in [-1 \,..\, \tau]$, we enumerate every possible query sub-part $\mathbf{q}_{ij}$ which is within Hamming distance τ_{ij} to $\mathbf{x}_i$, and then increase the counter $CN(\mathbf{q}_{ij}, \tau_{ij})$ by one.

Algorithm 5: GPH-SelfJoin($\mathcal{R}, \mathcal{P}, \tau$)

```
1  I ← ∅, CN(·,·) ← 0, R ← ∅;
2  foreach x ∈ R do
3      T ← DPAllocate(x, m, τ);
4      A ← ∅;
5      for i = 1 to m do
6          foreach s s.t. H(s, x_i) ≤ T[i] do
7              foreach ⟨y, j⟩ ∈ I_s and i = j do
8                  A ← A ∪ { y };
9      foreach y ∈ A do
10         if H(x, y) ≤ τ then
11             R ← R ∪ ⟨x, y⟩;
12     for i = 1 to m do
13         I_{x_i} ← I_{x_i} ∪ { ⟨x, i⟩ };
14         for j = 1 to m_i do
15             for τ_ij = −1 to τ do
16                 foreach q_ij s.t. H(q_ij, x_ij) ≤ τ_ij do
17                     CN(q_ij, τ_ij) ← CN(q_ij, τ_ij) + 1;
18 return R;
```

7.2 R-S Join

For R-S join, we assume that an index is created on $\mathcal{R}$ online and the objects in $\mathcal{S}$ are taken as batch queries.

The dimension partitioning is the same as the self join case. We also choose the sub-partitioning method to estimate the candidate number, and compute the candidate number statistics of $\mathcal{R}$ when building the index of $\mathcal{R}$. Then we scan the objects in $\mathcal{S}$ to find join results (pseudo-code given in Algorithm 6). An optimization is based on the observation that two objects in $\mathcal{S}$ may share the same projection over a part. This may result in the same enumerated signatures and the same candidates generated from this part. To exploit this observation, we process $\mathcal{S}$ in two passes. In the first pass, $\mathcal{S}$ is scanned horizontally and the objects are allocated with thresholds (Lines 2 – 3). In the second pass, $\mathcal{S}$ is scanned vertically. For each part, we group the objects by the projection over this part (Line 5). For each group, since the set of signatures enumerated with a smaller threshold are always contained by that with a larger threshold, we only pick the largest allocated threshold of this group (Line 8) and enumerate signatures using this threshold (Line 9). Then the index is accessed to retrieve candidates (Lines 10 – 12). Finally, the candidates of this group are verified using Hamming distance (Lines 13 – 16).

Algorithm 6: GPH-RSJoin$(\mathcal{R}, \mathcal{S}, \mathcal{P}, \tau)$

```
1  I ← BuildIndex(R, P, τ);
2  foreach x ∈ S do
3      T_x ← DPAllocate(x, m, τ);
4  foreach i = 1 to m do
5      G ← group x ∈ S by x_i;
6      foreach g_i ∈ G do
7          A_x ← ∅, ∀x ∈ S and x_i = g_i;
8          τ_i^max = max{ T_x[i] | x ∈ S, x_i = g_i };
9          foreach s s.t. H(s, g_i) ≤ τ_i^max do
10             foreach ⟨y, j⟩ ∈ I_s and i = j do
11                 foreach x ∈ S s.t. x_i = g_i and
                       T_x[i] ≤ τ_i^max do
12                     A_x ← A_x ∪ { y };
13         foreach x ∈ S s.t. x_i = g_i do
14             foreach y ∈ A_x do
15                 if H(x, y) ≤ τ then
16                     R ← R ∪ ⟨x, y⟩;
17 return R;
```

8 Pigeonhole Principle for Set Similarity Search

In this section, we discuss how the pigeonhole principle is used to solve set similarity search, an equivalent problem of Hamming distance search but defined on sets. For the similarity search problems with other similarity measures, such as string edit distance search and graph edit distance search, we refer readers to a recent study [38] for the application of the pigeonhole principle on these problems.

Problem 5 (Set Similarity Search)**.** *An object is a set of tokens drawn from a finite universe $\mathcal{U}$. Given a collection of objects $\mathcal{R}$, a query set q, find all $x \in \mathcal{R}$ such that $sim(x, q) \geq \tau$.*

$sim(\cdot, \cdot)$ is a set similarity function, e.g., the overlap similarity $O(x, y) = |x \cap y|$ and the Jaccard similarity $J(x, y) = \frac{|x \cap y|}{|x \cup y|}$.

We assume that the overlap similarity is used. To convert it to an equivalent Hamming distance search [5], we first regard a set as a $\mathcal{U}$-dimensional binary vector: the i-th dimension indicates whether the set has the i-th token. Then $O(x, y) \geq \tau \Leftrightarrow H(\mathbf{x}, \mathbf{y}) \leq |x| + |y| - 2\tau$. The main difference of the two problems is that they target different applications and the datasets are different in characteristics. In set similarity search, the number of dimensions (the size of token universe) is usually large (> 10000) and the vectors are usually sparse (< 1000 1s).

Prevalent approaches to set similarity search are based on prefix filter [1], [3], [5], [12], [29], [39], [46], [49], [50] and partition filter [2], [15]. Other methods include enumeration [14], [16], tree indexing [54], and postings list merge [19], [40]. Some of them target the set similarity join problem, the batch version of set similarity search. The methods can be easily adapted to each other. We focus on the two prevalent types of approaches.

8.1 Prefix Filter

The prefix filter [12] works directly on $O(x, y)$:

Lemma 6. *Suppose the tokens in each set is sorted by a total order $\mathcal{O}$. Let the prefix of a set x be the first $(|x| - \tau + 1)$ tokens in x. Consider two objects x and y. If $O(x, y) \geq \tau$, the prefixes of x and y must share at least one token.*

Candidates are generated by the filter and then verified by computing the exact overlap.

The prefix filter is essentially an extension of the general pigeonhole principle with two threshold arrays, though the fact has not been claimed by the aforementioned studies on set similarity search. To see this, we sort the tokens in $\mathcal{U}$ by the order $\mathcal{O}$ and convert sets to binary vectors: A set is regarded as a $|\mathcal{U}|$-dimensional binary vector, with each dimension representing a token. $\mathbf{x}[i] = 1$, if x has the i-th token in $\mathcal{U}$; or 0, otherwise. The overlap similarity $O(x, y) = O(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{|\mathcal{U}|} \mathbf{1}_{\mathbf{x}[i]=\mathbf{y}[i]=1}$, where $\mathbf{1}_{\mathbf{x}[i]=\mathbf{y}[i]=1}$ is the indicator function that returns 1, if $\mathbf{x}[i] = \mathbf{y}[i] = 1$; or 0, otherwise. Then we have the general pigeonhole principle for the $O(\mathbf{x}, \mathbf{y}) \geq \tau$ case (the proof is similar to Lemma 4 and omitted in the interest of space):

Lemma 7. *$\mathbf{x}$ and $\mathbf{y}$ are partitioned into m disjoint parts. Consider a threshold array $T = [\tau_1, \ldots, \tau_m]$ composed of integers. $\|T\|_1 = \tau + m - 1$. If $O(\mathbf{x}, \mathbf{y}) \geq \tau$, there exists at least one part i such that $O(\mathbf{x}_i, \mathbf{y}_i) \geq \tau_i$.*

Let $m = |\mathcal{U}|$. Each part is thus a dimension. We consider the following threshold array $T_{\mathbf{x}} = [\tau_1, \ldots, \tau_m]$ for $\mathbf{x}$:

- $\tau_i = 1$ for the first $|x| - \tau + 1$ dimensions where $\mathbf{x}[i] = 1$.
- $\tau_i = 2$ for the other $\tau - 1$ dimensions where $\mathbf{x}[i] = 1$.
- $\tau_i = 1$, if $\mathbf{x}[i] = 0$.

The first case states that the tokens in the prefix of x must have at least one match to form a candidate. The second case states that the tokens in x but not in the prefix are ignored since $O(\mathbf{x}[i], \mathbf{y}[i]) \leq 1 < 2$. The third case states that the tokens not in x are ignored since $\mathbf{1}_{\mathbf{x}[i]=\mathbf{y}[i]=1} = 0$ when $\mathbf{x}[i] = 0$. The threshold array $T_{\mathbf{y}}$ of $\mathbf{y}$ is constructed in the same way. The prefix filter is therefore converted to the following proposition:

5. Please refer to [50] for the conversion of other similarity measures.

Proposition 1. *If $O(\mathbf{x}, \mathbf{y}) \geq \tau$, there exists at least one $i \in [1 \mathinner{..} m]$, such that $O(\mathbf{x}_i, \mathbf{y}_i) \geq T_{\mathbf{x}}[i]$ and $O(\mathbf{x}_i, \mathbf{y}_i) \geq T_{\mathbf{y}}[i]$.*

Next we prove the proposition is true.

Proof. Because $\|T_{\mathbf{x}}\|_1 = (|x| - \tau + 1) + 2(\tau - 1) + (m - |x|) = \tau + m - 1$, $T_{\mathbf{x}}$ satisfies the condition in Lemma 7. $T_{\mathbf{y}}$ satisfies the condition for the same reason. Because the last two cases are ignored in $T_{\mathbf{x}}$ when generating candidates, there exists at least one i in the first $|x| - \tau + 1$ dimensions of $\mathbf{x}$ where $\mathbf{x}[i] = 1$, such that $O(\mathbf{x}_i, \mathbf{y}_i) \geq T_{\mathbf{x}}[i]$. So it is with $\mathbf{y}$. Without loss of generality, let $i_{\mathbf{x}}$ and $i_{\mathbf{y}}$ denote one of the i values satisfying this condition for $\mathbf{x}$ and $\mathbf{y}$, respectively. Because $O(\mathbf{x}_i, \mathbf{y}_i) = \mathbf{1}_{\mathbf{x}[i]=\mathbf{y}[i]=1}$, we have $\mathbf{x}[i_{\mathbf{x}}] = \mathbf{y}[i_{\mathbf{x}}] = 1$ and $\mathbf{x}[i_{\mathbf{y}}] = \mathbf{y}[i_{\mathbf{y}}] = 1$. If the proposition is false, then $T_{\mathbf{y}}[i_{\mathbf{x}}] = 2$ and $T_{\mathbf{x}}[i_{\mathbf{y}}] = 2$. These two conditions cannot be met at the same time. E.g., if $T_{\mathbf{y}}[i_{\mathbf{x}}] = 2$, then $i_{\mathbf{y}} < i_{\mathbf{x}}$. This means $T_{\mathbf{x}}[i_{\mathbf{y}}]$ belongs to the first case. $T_{\mathbf{x}}[i_{\mathbf{y}}] = 1$ and causes contradiction. □

Variants of prefix filter have been proposed for faster query processing, e.g., extended prefix [46], [50] and k-wise signatures [48]. The latter leverages the pigeonhole principle. Details can be found in a recent study [38].

8.2 Partition Filter

There have been two partition filter methods for set similarity search: PartEnum [2] and PartAlloc [15]. Both convert the problem to an equivalent Hamming distance search and work on $H(\mathbf{x}, \mathbf{y})$. For ease of exposition, we let $\tau' = |x| + |y| - 2\tau$.

PartEnum employs a two-level partitioning. It first partitions each vector into m_1 equi-width parts, each part with a Hamming distance threshold of $\tau_1 = \left\lceil \frac{\tau'+1}{m_1} \right\rceil - 1$. Then it partitions each part into $m_2 > \tau_1$ equi-width sub-parts, so that any two parts with Hamming distance τ_1 must agree on at least $(m_2 - \tau_1)$ sub-parts. The first-level filtering uses the pigeonhole principle with a threshold array $T_1 = [\left\lceil \frac{\tau'+1}{m_1} \right\rceil - 1, \ldots, \left\lceil \frac{\tau'+1}{m_1} \right\rceil - 1]$. It is not always tight because $\|T_1\|_1$ can be up to $\tau' > \tau' - m_1 + 1$, when $\tau' \mod m_1 = 1$. The second-level filtering enumerates every possible $(m_2 - \tau_1)$ parts on both data and query vectors to find candidates. It is a constraint on multiple parts and thus not covered by the pigeonhole principle.

PartAlloc is a pigeonhole principle-based algorithm. It partitions a vector into $m = \tau + 1$ equi-width parts. To generate candidates, each part has three options: exact match, differ by one dimension, and be ignored. It is the same as using a threshold array $T = [\tau_1, \ldots, \tau_m]$ such that $\tau_i = -1$, 0, or 1. Its filtering condition is tight as it keeps $\|T\|_1 = \tau - m + 1 = 0$. $\tau_1, \ldots, \tau_m$ are decided by a dynamic programming or a greedy algorithm which optimizes index access.

9 Experiments

9.1 Experiments Setup

The following algorithms are compared in the experiment.

- **MIH** is a method based on the basic pigeonhole principle [33]. It divides vectors into m equi-width parts and uses a threshold $\lfloor \frac{\tau}{m} \rfloor$ on all the parts to generate candidates. Its filtering condition is not tight. Signatures are enumerated on the query side. We utilize the open source of MIH on GitHub [32] and chose the fastest m setting on each dataset.
- **MIH$^+$** is an improved version of MIH [34]. It divides vectors into m equi-width parts, and uses a threshold $\lfloor \frac{\tau}{m} \rfloor$ on the first $\tau - m \lfloor \frac{\tau}{m} \rfloor + 1$ parts and $\lfloor \frac{\tau}{m} \rfloor - 1$ on the other parts. The sum of thresholds is $\tau - m + 1$, and thus the filtering condition is tight. We utilize the open source of MIH on GitHub [32] and chose the fastest m setting on each dataset.
- **HmSearch** is a method based on the basic pigeonhole principle [53]. Vectors are divided into $\lfloor \frac{\tau+3}{2} \rfloor$ equi-width parts. It has a filtering condition in multiple cases but not tight. The threshold of a part is either 0 or 1. This is our previous work and we utilize the existing source code.
- **PartAlloc** is a method to solve the set similarity join problem [15]. It divides vectors into $\tau + 1$ equi-width parts and allocate thresholds to them with three options: -1, 0, and 1. Its filtering condition is tight. Signatures are enumerated on both data and query vectors. The source code is received from the authors of [15]. We convert the Hamming distance constraint to an equivalent Jaccard similarity constraint [2]. The greedy method [15] 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 [8]. 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 to 3 and recall to 95%. $l = \lceil \log_{1-t^k}(1-r) \rceil$, where t is the Jaccard similarity threshold. The algorithm is implemented by ourselves.
- **GPH** is the method proposed in this paper. We implement it on top of the source code of MIH for fair comparison.

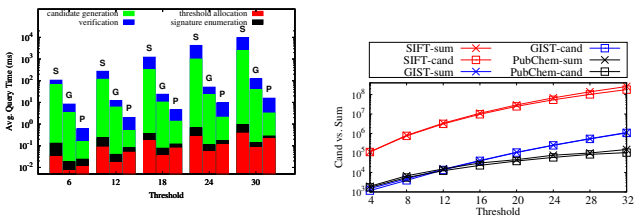
Other methods for Hamming distance search, e.g., [21], [24], [28], are not compared since prior work [53] showed they are outperformed by HmSearch. We do not consider the method in [41] because it focuses on small n (≤ 64) and small τ (≤ 4), and it is significantly slower than the other algorithms in 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 [35] 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 [35]. Due to its performance compared to MIH and the much larger threshold settings in our experiments, we do not compare with the method in [35].

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 [20]. We follow the method used in [33] to convert them into 128-dimensional binary vectors.
- **GIST** is a set of 80 million 256-dimensional GIST descriptors for tiny images [43].
- **PubChem** is a database of chemical molecules [18]. We sample 1 million entries, each of which is a 881-dimensional vector.

SIFT has the smallest skewness among the three. GIST is a moderately skewed dataset. PubChem is a highly skewed dataset. In addition to the three real datasets, we generate a synthetic dataset with varying skewness.

We sample a subset of 100 vectors from each dataset as the query workload for the partitioning of GPH. To generate real queries, from each dataset we sample 1,000 vectors (**differ** from the query workload for partitioning) and take the rest as



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

Fig. 2. Justification of Assumptions

data objects. We vary τ and measure the query response time averaged over 1,000 queries. For GPH and PartAlloc, threshold allocation time is also included. The τ settings are up to 32, 64, and 32 on the three real datasets, respectively. The reason why we set smaller thresholds on PubChem is more than 10% data objects are results when $\tau = 32$ due to the skewness.

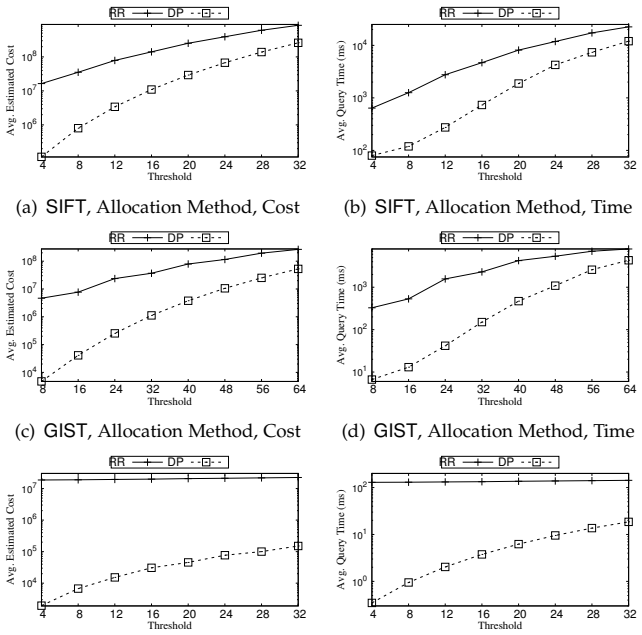
The experiments are carried out on a server with a Quad-Core Intel Xeon E3-1231 @3.4GHz Processor and 96GB RAM, running Debian 6.0. All the algorithms are implemented in C++ in a main memory fashion.

9.2 Justification of Assumptions

We first justify our assumptions for the cost model of threshold allocation. m is chosen for the best performance. Fig. 2(a) shows the query processing time of GPH on SIFT, GIST, and PubChem (denoted by 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 m parts ($\sum_{s \in S_{sig}} |I_s|$, denoted by dataset-sum) and the candidate sizes ($|S_{cand}|$, denoted by 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.

9.3 Evaluation of Threshold Allocation

We evaluate threshold allocation by comparing with a baseline algorithm (denoted by RR). RR allocates thresholds in a round robin manner, and the thresholds of the m parts 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 the dynamic programming algorithm (denoted by DP) on the three datasets. We also plot the costs of RR using our cost model. 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 the



(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

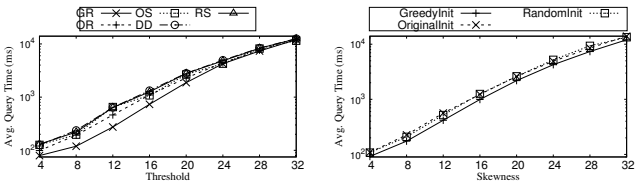
Fig. 3. Evaluation of Threshold Allocation

TABLE 3
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

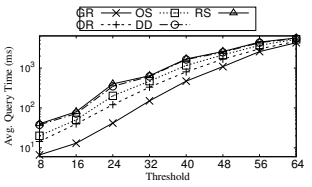
datasets with more skewness. On PubChem, the time of RR is close to sequential scan. With judicious threshold allocation, the time is reduced by nearly two orders of magnitude.

To evaluate the candidate number computation, we compare the sub-partitioning algorithm (SP) and the machine learning algorithm based on an SVM model (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-parts is 2. The size of the training data is 1,000 for the machine learning algorithms. Table 3 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 not shown here. Seeing these results, we choose the SVM model to estimate candidate numbers in the rest of the experiments.

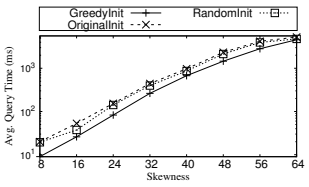


(a) SIFT, Partitioning Method, Time

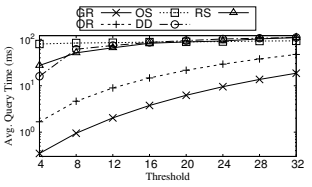
(b) SIFT, Initial Partitioning, Time



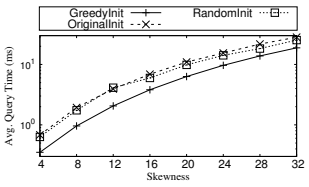
(c) GIST, Partitioning Method, Time



(d) GIST, Initial Partitioning, Time

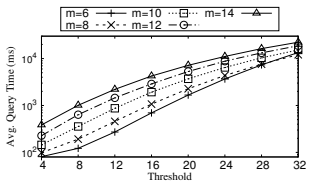


(e) PubChem, Partitioning Method, Time

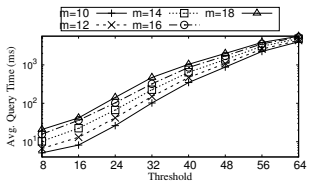


(f) PubChem, Initial Partitioning, Time

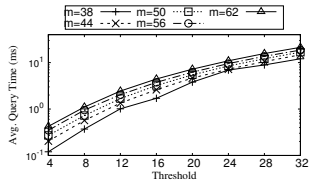
Fig. 4. Evaluation of Dimension Partitioning



(a) SIFT, Effect of m, Time



(b) GIST, Effect of m, Time



(c) PubChem, Effect of m, Time

Fig. 5. Effect of Partition Size

9.4 Evaluation of Dimension Partitioning

To evaluate the effect of partitioning, we compare our method (denoted by GR) with the following competitors: (1) OR is to use the original unshuffled order of the dataset. (2) RS is to perform a random shuffle on the original order. (3) OS [53] and DD [45] are two dimension rearrangement methods to make dimensions in each part 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 minimizes entropy (GreedyInit), (2) equi-width partitioning on the original unshuffled data (OriginalInit), and (3) equi-width partitioning after random shuffle (RandomInit). The 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 100 as the size of $\mathcal{Q}$ in our experiments.

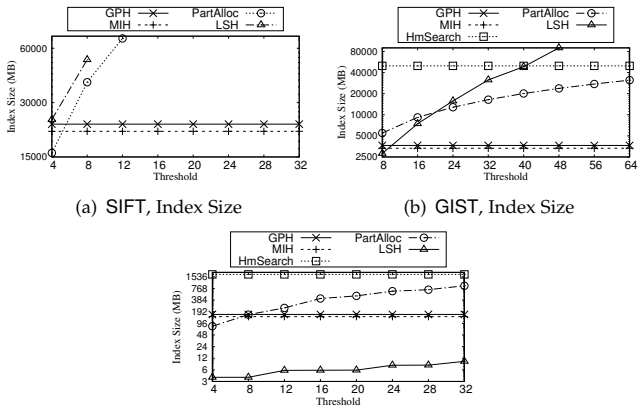
We also study the effect of partition size on the query processing performance. Figs. 5(a) – 5(c) show the query response times on the three datasets by varying the number of parts. The general trend is that a smaller m performs better under small τ settings. When τ increases, the best choice of m slightly increases. The reason is: (1) When τ is small, a small m is good enough. Dividing vectors into unnecessarily large number of parts yields very small parts and increases the frequency of signatures. (2) When τ is large, a small m means more thresholds will be allocated to a part, 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.

9.5 Comparison with Existing Methods

We compare GPH with alternative methods (equipped with the OS partitioning [53]) for Hamming distance search.

Index are compared first. Figs. 6(a) – 6(c) show the index sizes of the algorithms on the five 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. MIH$^+$ reports the same index size as MIH and thus is not plotted. GPH consumes 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 part 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 datasets. The reason is that PubChem has much more dimensions than the other datasets. 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 4. LSH runs out of memory when $\tau = 64$, and thus is only 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 for partition computation consists of queries with varying thresholds, we can run the partitioning once and use the same partition for different thresholds in real queries. This is also the reason why GPH has constant index construction time irrespective of τ.

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



(a) SIFT, Index Size (b) GIST, Index Size

(c) PubChem, Index Size

Fig. 6. Comparison with Alternatives - Index Size

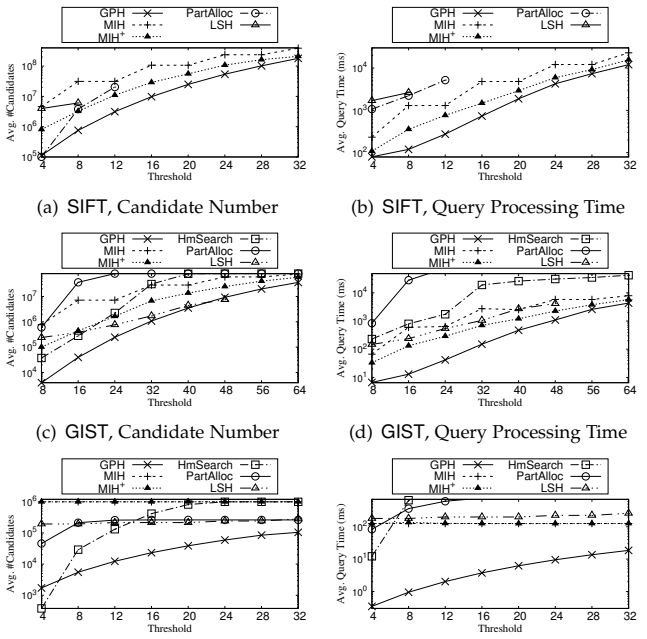
TABLE 4
Index Construction Time on GIST (s)

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

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 the other methods in candidate size and faster in query processing. 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, rendering the selectively of the chosen signatures very bad. On PubChem, LSH's performance is close to a sequential scan. GPH is always the fastest. The speed-ups against the runner-up algorithms on the three datasets are up to 3, 10, and 123 times, respectively.

9.6 Varying Dimensionality

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 then 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

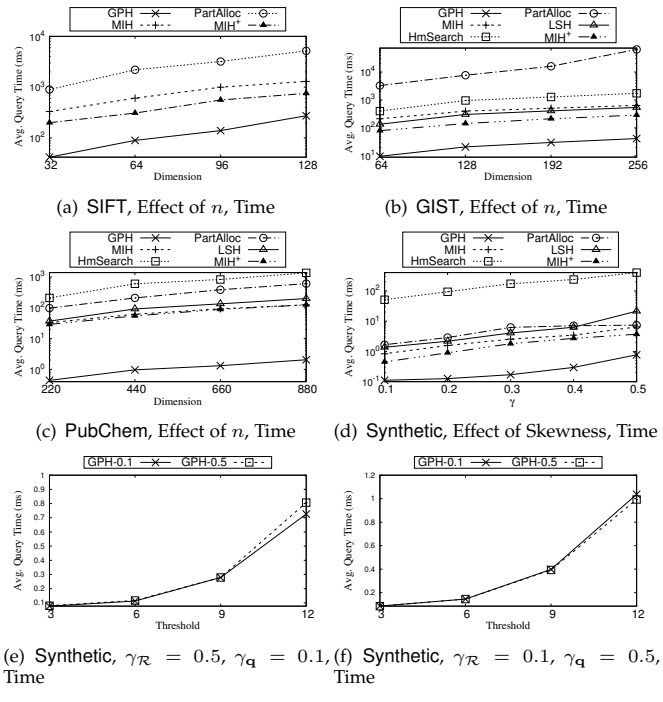


(a) SIFT, Candidate Number (b) SIFT, Query Processing Time

(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, $\gamma_{\mathcal{R}} = 0.5$, $\gamma_{\mathbf{q}} = 0.1$, Time (f) Synthetic, $\gamma_{\mathcal{R}} = 0.1$, $\gamma_{\mathbf{q}} = 0.5$, Time

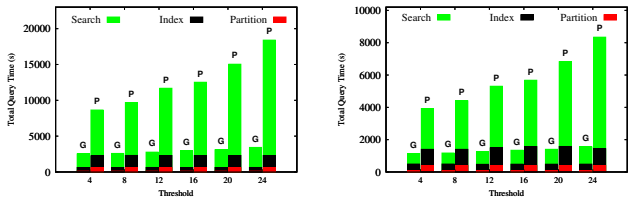
Fig. 8. Varying Number of Dimensions and Skewness

to verify. (2) The verification cost increases with n because more dimensions are compared. Nonetheless, GPH is always the fastest, especially on the highly skewed PubChem.

9.7 Varying Skewness

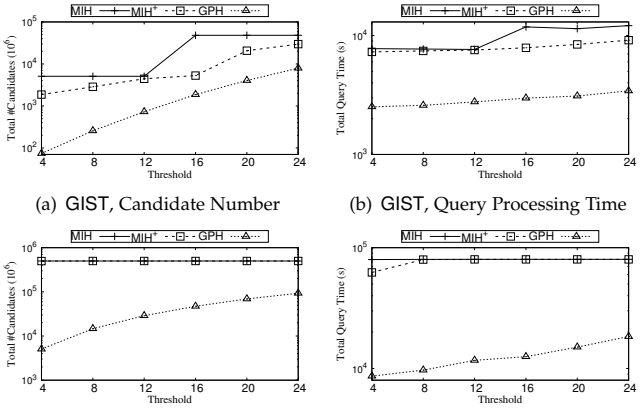
We study the performance by varying skewness [6]. As seen from Fig. 1, the relationship between skewness and dimensions is approximately linear (except PubChem) on most datasets. On the basis of this observation, the synthetic dataset is generated as follows: The number of dimensions is 128. The mean

6. See the footnote in Section 1 for the measurement of dataset skewness.



(a) Self Join (b) R-S Join

Fig. 9. Join Time Breakdown



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

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

Fig. 10. Self Join

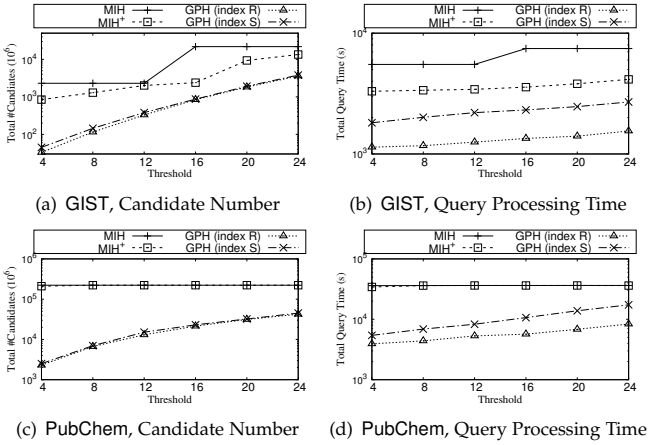
skewness is controlled by a parameter γ, and the skewnesses of the 128 dimensions range from 0 to 2γ. We set $\tau = 12$. The query processing times are plotted in Fig. 8(d). The general trend is that all the algorithms become slower on more skewed data. This is expected as signatures become less selective. Thanks to variable partitioning and threshold allocation, GPH is the fastest among the competitors.

To demonstrate the robustness of GPH, we show that even if the distribution of real queries is different from the sample for the partitioning, our method retains good performance. We generate a synthetic dataset with a γ of 0.5, and then compute a partition with two query workloads: $\gamma = 0.5$ (denoted by GPH-0.5) and $\gamma = 0.1$ (denoted by 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 even if the partitioning is done with a workload whose distribution is different from real queries, the query processing performance is almost the same. A slight change is noticed only when τ is as large as 12: the query processing speed drops by 11.1% and 4.4%, respectively.

9.8 Experiments on Hamming Distance Join

To study the performance of Hamming distance join, we conduct experiments on GIST and PubChem. 1 million objects are sampled from the original corpora as $\mathcal{R}$ for self join. Then another 2 million objects are sampled as $\mathcal{S}$ for R-S join.

We decompose the query processing time into three parts: partitioning, searching for join results, and updating index and the statistics for candidate number computation. Figs. 9(a) and 9(b) show the decomposed times for self join and R-S



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

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

Fig. 11. R-S Join

join, respectively. G denotes GIST and P denotes PubChem. For both joins, we observe: The partitioning time is the same across different thresholds as we only consider the entropy to compute the partition. The indexing time is also approximately constant. Most query processing time is dedicated to searching, whose time keeps increasing with the threshold. PubChem consumes more query processing time due to the more skewness. The difference between the two joins is that on R-S join, less percentage of overall time is spent on searching, because this step is optimized by the vertical scan over $\mathcal{S}$.

We adapt MIH and MIH$^+$ for Hamming distance join and compare GPH with it. Other methods are not considered as they have been shown much slower on Hamming distance search, and all of them follow the same index nested loops join style as MIH and MIH$^+$ for Hamming distance join. It is very unlikely that these methods become faster than the selected competitors. For self join, the candidate numbers and the running times on the two datasets are plotted in Figs. 10(a) – 10(d). On both datasets, GPH produces the least numbers of candidates, and the candidates are much fewer than MIH and MIH$^+$. For MIH and MIH$^+$, almost all the objects on PubChem become candidates, while GPH successfully handles this high skewness case with the threshold allocation. The different in candidate size results in the significant gaps in running times: GPH is up to 4 and 10 times faster than the runner-up, MIH$^+$, on GIST and PubChem, respectively. For R-S join, the candidate numbers and the running times are plotted in Figs. 11(a) – 11(d). For GPH, we also study which relation, $\mathcal{R}$ (smaller) or $\mathcal{S}$ (larger), should be indexed for better performance. For MIH and MIH$^+$, the results are similar to those we have witnessed in the experiments of self join. For GPH, the candidate number is much less than MIH and MIH$^+$, and it is insensitive to which relation is indexed. However, indexing the smaller relation is faster, because more shared computation can be exploited by the vertical scan of the larger relation. The speed-up of GPH (indexing $\mathcal{R}$) over MIH$^+$ is 5 times on GIST and 9 times on PubChem.

10 Related Work

The notion of Hamming distance search was first proposed in [30]. 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 $\tau = 1$ [6], [7], [27], [51]. Among them, the method in [27] 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 [7] 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 [13] 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 $\tau' < \tau$. In [24], [33], [42], vectors are divided into a number of parts such that query results must have at least one exact match with the query in one of the parts. The idea of recursive partitioning was covered in [28]. Before that, a two-level partitioning idea was adopted by the PartEnum method [2]. Song *et al*. [41] proposed to enumerate the combinations within threshold τ' in each part to avoid the verification of candidates. Ong and Bober [35] proposed an approximate method utilizing variable length hash keys. In [53], vectors are divided into $\lfloor \frac{\tau+3}{2} \rfloor$ parts, and the threshold of a part can be 0 or 1. Deng *et al*. [15] also proposed to use different thresholds, which are computed by a dynamic programming or a greedy 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 [2] in the original dimensions to avoid highly correlated dimensions in the same parts. The second is to perform a dimension rearrangement [26], [45], [53] to minimize the correlation between dimensions in each part. 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 appears in [9], stating 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 principle.

11 Conclusion

We proposed a new approach to Hamming distance search. Observing the major drawbacks of the basic pigeonhole principle adopted by 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 extended our methods to Hamming distance joins, and discussed the application of the pigeonhole principle in set similarity search. 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.

References

[1] D. C. Anastasiu and G. Karypis. L2AP: fast cosine similarity search with prefix L-2 norm bounds. In *ICDE*, pages 784–795, 2014.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

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

[5] P. Bouros, S. Ge, and N. Mamoulis. Spatio-textual similarity joins. *PVLDB*, 6(1):1–12, 2012.

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

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

[8] A. Z. Broder. On the resemblance and containment of documents. In *SEQS*, pages 21–29, 1997.

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

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

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

[12] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

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

[14] D. Deng, A. Kim, S. Madden, and M. Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. *PVLDB*, 10(10):1082–1093, 2017.

[15] 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.

[16] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *SIGMOD*, pages 905–920, 2018.

[17] 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.

[18] N. C. for Biotechnology Information. The PubChem Project. https://pubchem.ncbi.nlm.nih.gov/, 2017.

[19] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity metricselection queries. In *ICDE*, pages 267–276, 2008.

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

[21] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[22] 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.

[23] 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.

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

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

[26] 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.

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

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

[29] W. Mann and N. Augsten. PEL: position-enhanced length filter for set similarity joins. In *Proc. GvD (Foundations of Databases)*, pages 89–94, 2014.

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

[31] 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.

[32] M. Norouzi. Multi-index Hashing. https://github.com/norouzi/mih, 2014.

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

[34] M. Norouzi, A. Punjani, and D. J. Fleet. Fast exact search in hamming space with multi-index hashing. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(6):1107–1119, 2014.

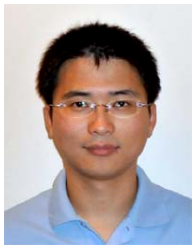
[35] E. Ong and M. Bober. Improved hamming distance search using variable length hashing. In *CVPR*, pages 2000–2008, 2016.

[36] H. Park and L. Stefanski. Relative-error prediction. *Statistics & Probability Letters*, 40(3):227 – 236, 1998.
[37] J. Qin, Y. Wang, C. Xiao, W. Wang, X. Lin, and Y. Ishikawa. GPH: Similarity search in hamming space. In *ICDE*, pages 29–40, 2018.
[38] J. Qin and C. Xiao. Pigeonring: A principle for faster thresholded similarity search. *PVLDB*, 12(1):28–42, 2018.
[39] L. A. Ribeiro and T. Härder. Generalizing prefix filtering to improve set similarity joins. *Inf. Syst.*, 36(1):62–78, 2011.
[40] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, pages 743–754, 2004.
[41] 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.
[42] 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.
[43] 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.
[44] S. A. Vinterbo. A note on the hardness of the k-ambiguity problem. Technical report, Harvard Medical School, 06 2002.
[45] J. Wan, S. Tang, Y. Zhang, L. Huang, and J. Li. Data driven multi-index hashing. In *ICIP*, pages 2670–2673, 2013.
[46] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.
[47] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014.
[48] P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, and Y. Ishikawa. Local similarity search for unstructured text. In *SIGMOD*, pages 1991–2005, 2016.
[49] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang. Leveraging set relations in exact set similarity join. *PVLDB*, 10(9):925–936, 2017.
[50] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15:1–15:41, 2011.
[51] A. C.-C. Yao and F. F. Yao. Dictionary look-up with one error. *J. Algorithms*, 25(1):194–202, 1997.
[52] W. Zhang, K. Gao, Y. Zhang, and J. Li. Efficient approximate nearest neighbor search with integrated binary codes. In *ACM Multimedia*, pages 1189–1192, 2011.
[53] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu. Hmsearch: an efficient hamming distance query processing algorithm. In *SSDBM*, pages 19:1–19:12, 2013.
[54] Y. Zhang, X. Li, J. Wang, Y. Zhang, C. Xing, and X. Yuan. An efficient framework for exact set similarity search using tree structure indexes. In *ICDE*, pages 759–770, 2017.

Jianbin Qin is a research scientist at Shenzhen Institute of Computing Sciences, Shenzhen University. He received B.E. degree from Northeastern University, China in 2007 and Ph.D. degree from the University of New South Wales in 2013. His research interests include data integration, textual databases, and information retrieval.

Chuan Xiao is a specially appointed associate professor with the Graduate School of Information Science and Technology, Osaka University, and a guest associate professor with the Graduate School of Informatics, Nagoya University. He received B.E. degree from Northeastern University, China in 2005 and Ph.D. degree from the University of New South Wales in 2010. His research interests include data cleaning, data integration, textual databases, and graph databases.

Yaoshu Wang is a researcher at Shenzhen Institute of Computing Sciences, Shenzhen University. He received the Ph.D. degree from the University of New South Wales in 2018. His research interests include similarity search, data integration, and textual databases.



Wei Wang received the Ph.D. degree from the Hong Kong University of Science and Technology in 2004. He is a professor with the University of New South Wales. His research interests include data integration, information retrieval, and query processing and optimization.

Xuemin Lin is the Scientia Professor with the University of New South Wales, and a concurrent professor with the School of Software, East China Normal University. Before joining UNSW, he held various academic positions with the University of Queensland and the University of Western Australia. He received Ph.D. degree from the University of Queensland in 1992 and B.Sc. degree from Fudan University in 1984. He is a Fellow of IEEE. His research interests lie in data streams, approximate query processing, spatial data analysis, and graph visualization.

Yoshiharu Ishikawa is a professor with the Graduate School of Informatics, Nagoya University. He received B.S., M.E., and Dr. Eng. degrees from the University of Tsukuba in 1989, 1991, and 1995, respectively. His research interests include spatio-temporal databases, mobile databases, sensor databases, data mining, information retrieval, and e-science.

Guoren Wang is a professor with the School of Computer Science & Technology, Beijing Institute of Technology, China. Before joining BIT, he was a professor with Northeastern University, China. He received Ph.D. degree from Northeastern University, China in 1996. His research interests include graph databases, query processing and optimization, and XML data management.