HmSearch: An Efficient Hamming Distance Query Processing Algorithm

Xiaoyang Zhang[†] Jianbin Qin[†] [*] Wei Wang[†] Yifang Sun[†] Jiaheng Lu[‡]

[†]University of New South Wales, Australia
{xyzhang, jqin, weiw, yifangs}@cse.unsw.edu.au

[‡] Renmin University of China, China
jiahenglu@ruc.edu.cn

ABSTRACT

Hamming distance measures the number of dimensions where two vectors have different values. In applications such as pattern recognition, information retrieval, and databases, we often need to efficiently process *Hamming distance query*, which retrieves vectors in a database that have no more than k Hamming distance from a given query vector. Existing work on efficient Hamming distance query processing has some of the following limitations, such as only applicable to tiny error threshold values, unable to deal with vectors where the value domain is large, or unable to attain robust performance in the presence of data skew.

In this paper, we propose HmSearch, an efficient query processing method for Hamming distance queries that addresses the above-mentioned limitations. Our method is based on improved enumeration-based signatures, enhanced filtering, and the hierarchical binary filtering-and-verification. We also design an effective dimension rearrangement method to deal with data skew. Extensive experimental results demonstrate that our methods outperform state-of-the-art methods by up to two orders of magnitude.

1. INTRODUCTION

In this paper, we study the problem of efficiently processing *Hamming distance queries* with a fixed threshold k.

Hamming distance measures the number of dimensions where two vectors have different values. In many applications, we often need to efficient process the *Hamming distance query*, which retrieves vectors in a database that have Hamming distance no more than k from a given query vector. For example,

- In order to identify near-duplicate Web pages, Google uses SimHash to obtain a 64-dimension vector for each web page. Two web pages are considered as near-duplicate if their vectors are within Hamming distance 3 [16].
- Similarity search is widely used in Chemical informatics to search and classify known chemicals, virtually screen of chemicals for drug discovery, and predict and optimize the properties of existing active compounds [10, 20]. A fundamental query is to find all the molecules whose 881-bit fingerprints have Tanimoto similarity no less than t to the fingerprint of a query molecule. As will be shown in section 2, this can be transformed into a Hamming distance query.
- Locality sensitive hashing [12] is a widely used technique to perform approximate similarity search with probabilistic guarantees. Recently, C2LSH [11] is proposed to address the issue of excessive index space required by traditional LSH method without affecting the theoretical guarantees. At the core of the method is a Hamming distance query with a threshold k (computed from several parameters) against vectors of the database objects generated by N LSH functions.

While there are prior studies on efficient query processing methods for Hamming distance search with a fixed threshold k, they suffer from some of the following problems:

- *Limited to tiny k values.* Early solutions based on reduction to exact matching problems only work for a very small k values [16, 24]. Recent proposals [14, 21] are able to process slightly larger k thresholds, but are still fairly limited as the performance deteriorates rapidly with the increase of k due to lack of effective pruning.
- *Unable to handle a large value domain.* Most existing solutions were designed for binary vectors (i.e., value domain size is 2), and they incur huge space usage when the value domain size is large, which is required by data generated by MinHash [5, 25], or by kNN search [21, 11].
- *Unable to handle skewed data.* As we show in Section 5, existing methods will all degenerate to essentially the brute-force linear-scan method in the presence of data skew, which is present in many datasets such as the chemical fingerprints in PubChem.

In this paper, we propose HmSearch, an efficient query processing method for Hamming distance queries that addresses the above-mentioned limitations. Our method is based on partitioning the dimensions into partitions such that the

[*]Corresponding author.

query results must have at least one partition whose Hamming distance is no more than 1 with the corresponding partitions of the query. We can then use either *1-deletion variants* or *1-variants* to efficiently process the special 1-Hamming distance query. We further strengthen the partitioning method by requiring candidates to match more than one variant-based signature under certain circumstances. We also develop a hierarchical binary representation for the data, which enables us to perform filtering and verification simultaneously with almost no additional cost. To deal with data skew, we design an effective dimension rearrangement method. Extensive experimental results demonstrate that our methods outperform the state-of-the-art methods by up to two orders of magnitude, especially for medium-valued k and skewed datasets.

Our contributions can be summarized as follows:

- We propose a versatile method to process Hamming distance queries under a wide spectrum of settings including error threshold k, and domain size of dimension values. It is also robust against data skew thanks to the dimension rearrangement technique.
- We compare the proposed method with state-of-the-art methods in an extensive experimental study. The results demonstrate that our method can outperform existing ones by up to two orders of magnitude.

The rest of the paper is organized as follows. Section 2 defines the problem and introduces the preliminaries. Section 3 introduces the variant-based signatures for Hamming distance query with threshold 1. Section 4 presents our HmSearch method with tighter pruning and a filtering-and-verification procedure based on hierarchical binary representation of the data. Section 5 presents our technique of rearranging the dimensions to handle data skew. Experimental results are presented in Section 6 followed by related work in Section 7. Section 8 concludes the paper.

2. PRELIMINARIES

Problem Definition. Since Hamming distance is defined on vectors of same number of dimensions, we consider all data and query vectors having N dimensions in this paper. The i-th dimension is denoted as D_i. $V[D_i]$ represents the i-th dimension value of a vector V. Without loss of generality, we assume the domain of possible values for D_i is the same and is denoted as Σ.

Let $\Delta(x, y) = 0$ if $x = y$ and 1 otherwise. The Hamming distance between two vectors S and T is defined as:

$$H(S,T) = \sum_{i=1}^{N} \Delta(S[i], T[i])$$

If we consider S as a yardstick, we can also say T has $H(S,T)$ *error(s)* with respect to S.

Given a dataset $\mathcal{V}$ of vectors, a Hamming distance query of a query vector Q and threshold k retrieves all vectors in the dataset with Hamming distance to Q no more than k, or

$$\{ v_i \in \mathcal{V} \mid H(v_i, Q) \le k \}$$

Such a query is also known as the k-query due to [17].

Table 1: Notations

Symbol	Definition
N	Dimensionality of all the vectors
k	Hamming distance threshold
n	Number of vectors in the database
Σ	The domain for all values of the vector
$\#$	Deletion marker
$[1_1, 2_2]$	We use dimension ID in the subscript to distinguish values
v^i	The i-th partition of vector v
I_{sig}	Postings list of signature sig
$x_{(i)}$	The i-th bit (from left to right) of the binary representation of an integer x (e.g., $5_{(3)}$ is 1)

Relationship with Tanimoto Similarity. In Chemical Informatics, molecules can be represented by *binary* vectors, which are called fingerprints [10]. One of the most popular measures to measure the similarity between fingerprints is the Tanimoto similarity [19]. Let S and T be binary vectors. $set(S)$ is the set representation of S; i.e., $set(S) \overset{\text{def}}{=} \{ D_i \mid V[D_i] \neq 0 \}$. The Tanimoto similarity, or essentially the Jaccard similarity, is defined as:

$$T(S,T) = \frac{|set(S) \cap set(T)|}{|set(S) \cup set(T)|}$$

We can derive the following equivalence between a constraint based on the Tanimoto similarity and that based on the Hamming distance:

$$T(Q,S) \ge t \iff H(Q,S) \le \frac{1-t}{1+t} \cdot (|set(S)| + |set(Q)|)$$

If we perform a search using a Tanimoto similarity threshold of t, we can derive the threshold of a Hamming distance query with threshold $k_Q = \frac{1-t}{t} \cdot |set(Q)|$. This is because for any result S satisfying $T(Q,S) \ge t$, we know that $|set(S)| \in [t \cdot |set(Q)|, |set(Q)|/t]$.

Notations. We list notations used in the paper in Table 1.

3. VARIANT-BASED SIGNATURES

In this section, we first introduce the definitions of *1-variant* and *1-deletion-variant*, then illustrate how to use these two variants to answer the Hamming distance query for $k = 1$ (also called **1-query**), respectively. As we will see shortly in Section 4.1, the general case Hamming distance search can be reduced to this special case via partitioning.

3.1 Variants and Deletion Variants

The *1-variant* of a vector v with respect to the value domain Σ is any vector v' in Σ^N such that $H(v, v') \le 1$. All the 1-variants of v are denoted collectively as $\text{1-Var-Set}(v)$. v is by definition its own 1-variant. The 1-variant can be computed easily by substituting another value from Σ for $v[i]$. For any $V_i \in \mathcal{V}$, the total number of 1-variants is therefore $1 + (|\Sigma| - 1)N$.

Let $\Sigma^* = \Sigma \cup \{ \# \}$. The *1-deletion-variants* of a vector v are all the vectors obtained by substituting the deletion marker $\#$ for $v[i]$. They are also collectively denoted as $\text{1-Del-Var-Set}(v)$. The total number of 1-deletion-variants is N.

All the above different kinds of variants are referred to as *variants* generically.

Example 1. *Consider $v = [1_1, 2_2, 1_3]$ and $\Sigma = \{1, 2, 3\}$. All of its* 1-variants *are:* $[1_1, 2_2, 1_3]$, $[2_1, 2_2, 1_3]$, $[3_1, 2_2, 1_3]$, $[1_1, 1_2, 1_3]$, $[1_1, 3_2, 1_3]$, $[1_1, 2_2, 2_3]$, $[1_1, 2_2, 3_3]$. *All of its* 1-deletion-variants *are* $[\#, 2_2, 1_3]$, $[1_1, \#, 1_3]$, $[1_1, 2_2, \#]$.

3.2 1-Query Processing using variants

3.2.1 1-Query Processing using 1-Variants

The following Lemma gives us a necessary and sufficient condition for two vectors to be within Hamming distance of 1 based on *1-variants*.

Lemma 1. *Consider two vectors S and T. $H(S,T) \leq 1$ if and only if 1-Var-Set$(S) \cap \{T\} \neq \emptyset$.*

According to Lemma 1, we can use the following procedure to answer 1-queries.

- *Indexing.* We generate all the 1-variants for every vector in the database and index the variants using an inverted index I.
- *Query Processing.* We directly look up the query in the index. The returned results are exactly the query results.

The index space complexity of this method is $O(|\Sigma| \cdot N \cdot n)$. The query time complexity is $O(1 + occ)$, where occ denotes the number of query results.

3.2.2 1-Query Processing using 1-Deletion-Variants

Many of the existing Hamming distance query processing methods assume a binary value domain, hence *1-variants* based methods are usually preferred to *1-deletion-variants* based methods (to be introduced below), as the former achieves $O(1)$ query time at the cost of just doubling the index space. However, when $|\Sigma|$ is large (e.g., Σ can be as large as 172 for vectors generated by MinHash [5]), *1-variants*-based methods will incur excessive amount of space usage (and building time) for the index, which is not practical or competitive. Instead, 1-deletion variants will be a good choice under such circumstances.

The following Lemma gives us a necessary and sufficient condition for two vectors to be within Hamming distance of 1 based on the intersection of their *1-deletion-variants* sets.

Lemma 2. *Consider two vectors S and T. $H(S,T) \leq 1$ if and only if 1-Del-Var-Set$(S) \cap$ 1-Del-Var-Set$(T) \neq \emptyset$.*

Note that even if $S = T$, they will have N common 1-deletion-variants anyway.

According to Lemma 2, we can use the following procedure to answer 1-queries.

- *Indexing.* We generate all the 1-deletion-variants for every vector in the database and index the variants using an inverted index I.
- *Query Processing.* We generate all the 1-deletion-variants of the query and look them up in the index. The returned results are merged and become the query results.

The index space complexity of this method is $O(N \cdot n)$. The query time complexity is $O(N + N \cdot occ)$.

Example 2. *Continuing Example 1, we index all the 1-deletion-variants of v (and other vectors in the database). To process the query $Q = [1_1, 2_2, 3_3]$, we first generate all q's 1-deletion-variants: $[\#, 2_2, 3_3]$, $[1_1, \#, 3_3]$, and $[1_1, 2_2, \#]$; and we look them up in the inverted index and merge the returned results. v will be found in the postings list of $I_{[1_1, 2_2, \#]}$.*

4. THE HmSearch ALGORITHM

In this section, we first introduce how to reduce the general Hamming distance problem to Hamming distance problem with $k = 1$, so that the *variants*-based methods introduced in Section 3 can be employed to answer each reduced query. Then we present HmSearch, our proposed query processing method with advanced threshold-based pruning and a technique to perform pruning and verification simultaneously.

4.1 Reduction of the General Hamming Distance Problem

The prevalent approach to answer Hamming distance query is based on reducing the general problem into several instances of Hamming distance queries with smaller threshold values via partitioning.

First, we introduce a few concepts. We consider a partitioning scheme that divides the N dimensions into κ partitions; each partition, denoted as p_i, is a subset of dimensions $\{D_{i,1}, D_{i,2}, \ldots, D_{i,|p_i|}\}$. Given a vector v, its projection onto a partition, i.e., $v[D_{i,1}, \ldots, D_{i,|p_i|}]$ forms a new *projected* vector, denoted as *projection* v^i.

Definition 1 (Match, Exact-Match and 1-Match). *Given two partitions p_i and p_j, if $H(p_i, p_j) \leq 1$, they are said to* match *each other. In addition, if $H(p_i, p_j) = 0$, they are said to* exact-match *each other; if $H(p_i, p_j) = 1$, they are said to* 1-match *each other.*

Next, we present the following Lemma, which shows the necessary condition for two vectors to be within Hamming distance of k based on partitioning:

Lemma 3. *Given two vectors S and T such that $H(S,T) \leq k$, if we divide the N dimensions (arbitrarily) into κ partitions, there are at least $m = \kappa - \left\lfloor \frac{k}{\lfloor k/\kappa \rfloor + 1} \right\rfloor$ partitions, $\{p_1, p_2, \ldots, p_m\}$, such that $H(S^i, T^i) \leq \lfloor k/\kappa \rfloor$, $\forall 1 \leq i \leq m$.*

Proof. Assume the contrary that at most $m - 1$ partitions have at most $\lfloor k/\kappa \rfloor$ errors. Then all the rest $\kappa - m + 1$ partitions should have at least $\lfloor k/\kappa \rfloor + 1$ errors. Let $\beta = \lfloor k/\kappa \rfloor + 1$. Then the total amount of error is at least

$$\beta \cdot (\kappa - m + 1) = \beta \cdot \left(\left\lfloor \frac{k}{\beta} \right\rfloor + 1 \right) > k$$

which contradicts the condition that $H(S,T) \leq k$. □

This Lemma is a generalization of previous results (such as Theorem 3.1 in [14] and [21]). As such, it has several instantiations and each results in different algorithms. For example, [16] chooses $\kappa = k + 1$, such that there must $m = 1$ *exact-matching* partition, as $\lfloor k/(k+1) \rfloor = 0$. [14] essentially chooses $\kappa = \lfloor k/2 \rfloor + 1$, hence entailing at least $m = 1$ *1-matching* partition. [21] considers the general case of choosing any κ, but fails to capitalize on cases where m could be greater than 1.

In the following, we call these m partitions in Lemma 1 as *matching partitions*. When $\lfloor k/\kappa \rfloor = 1$, we also distinguish *exact-match partitions* and *1-match partitions*, based on whether the Hamming distance is 0 or 1.

Based on Lemma 3, the overall query processing method can be captured in the following general framework:

- In the *indexing phrase*, each vector in the database is partitioned into κ partitions. Each partition is indexed in such a way that it is possible to efficiently answer a Hamming distance query with threshold $\lfloor k/\kappa \rfloor$ for the projection of all vectors on this partition.
- In the *query processing phase* (See Algorithm 1), the query vector is partitioned in the same way into κ partitions. A *special* Hamming distance query with threshold $k' = \lfloor k/\kappa \rfloor$ is issued on each query partition to obtain a list of candidate vectors whose corresponding partitions have at most k' Hamming distance from the query partition Q^i (Line 4). The returned results of these κ queries are added to the *CAND* hash table, which count the number of times a vector has been encountered. We perform the filtering (See Algorithm 2), which essentially check the occurrence number against m. If the vector passes the filtering, it will then be verified (Line 7) against the entire query Q.

Algorithm 1: HammingQuery(Q, k, κ)

```
/* generate candidates                                    */
1 $CAND \leftarrow$ empty hash table that maps vector ID to an integer;
2 partition$(Q, \kappa)$;
3 for each the $i$-th partition $Q^i$ of the query $Q$ do
4     for each vector ID $v \in$ reducedHammingQuery$(Q^i, \lfloor k/\kappa \rfloor)$ do
5         $CAND[v] \leftarrow CAND[v] + 1$;
          // $CAND[v]$ is initialized to 0 upon first visit
/* filtering and then verification                        */
6 for each candidate $v \in CAND$ do
7     if filter(v, $\kappa - \left\lfloor \frac{k}{\lfloor k/\kappa \rfloor + 1} \right\rfloor$) = false then
8         if verify$(Q, v)$ then
9             output $v$;
```

Algorithm 2: filter(v, m)

```
Output : Returns true if $v$ is filtered (i.e., disqualified)
1 if $CAND[v] < m$ then
2     return true;
3 return false;
```

Remark 1. *Note that, almost all existing methods apply the even partition strategy. One way to use this strategy is to partition the vectors into κ partitions evenly, such that each partition has length either $\lfloor N/\kappa \rfloor$ or $\lceil N/\kappa \rceil$, where the last $N - \lfloor N/\kappa \rfloor \cdot \kappa$ partitions having the longer length. However, the evenly partitioning strategy has severe drawbacks in certain conditions, we will discuss uneven partition method in section 5.*

Remark 2. *In addition to the indexing approach described above, the other way is to replicate the vectors and keep the vectors in each copy sorted. Binary search is used on each copy to locate candidates. This method usually incurs much overheads in both space and query time, and is mostly adopted in distributed systems to achieve a high degree of parallelism [16].*

4.2 Partitioning

In our HmSearch method, based on the framework illustrated above, we choose to partition the dimensions into $\kappa = \left\lfloor \frac{k+3}{2} \right\rfloor$ partitions. According to Lemma 3, any query result vector must have at least one matching partitions, i.e., having Hamming distance at most 1. However, we show later that the pruning condition can be strengthen, which will help to keep the candidate size low when k increases.

Our enhanced filtering is based on observing the following artifact of the partitioning scheme. Let $k = 2c$, where c is an integer. Obviously, the partition number $\kappa = \left\lfloor \frac{k+3}{2} \right\rfloor = c+1$. Based on Lemma 3, $m = 1$, which means a query result vector requires only one match. We observe that if first $c + 1$ errors are evenly distributed into $c + 1$ partitions, there are only $c - 1$ errors left to put into $c + 1$ partitions. Hence in this case, two 1-matches exist. By carefully analyzing this condition, we find that if there is no exact-match, there must exist at least two 1-matches. Similar observation can be found when $k = 2c+1$. Therefore, we establish the following Lemma, which gives us a tighter condition for filtering.

Lemma 4 (Enhanced Filtering Condition). *Consider processing the Hamming distance query for Q with threshold k, and that the dimensions have been divided into $\kappa = \lfloor (k + 3)/2 \rfloor$ partitions. A query result S must satisfy the following conditions:*

- *If k is an even number, S must have at least* one *exact-matching partition, or* two *1-matching partitions.*
- *If k is an odd number, S must have at least* two *matching partitions, where at least* one *of the matches should be an exact-match, or S must have at least* three *1-matching partitions.*

Proof. When k is even, let $k = 2c$. Then $\kappa = c + 1$. Assume the contrary, i.e., there is at most *one* 1-matching partition. Then one partition has at least 1 error and the rest c partitions have at least 2 errors each. The total number of errors is at least $2c + 1 = k + 1$, which contradicts the fact that it is a query result.

When k is odd, let $k = 2c + 1$. Then $\kappa = c + 2$. Assume the contrary, i.e., there is at most *two* 1-matching partitions or *one* exact-matching partition. Considering the former condition, since both of the matching partitions have at least 1 error each and the rest c partitions have at least 2 errors each. The total number of errors is at least $2k + 2 = k + 1$; Considering the latter condition, since $c + 1$ partitions have at least 2 errors each. The total number of errors is at least $2k + 2 = k + 1$. Both of the cases contradict the fact that it is a query result. □

This Lemma helps to control the growth of candidate size when k increases. As show in experiment Figures 5(l) and 5(m), the reduction of candidate size could reach up to 2 orders of magnitude.

4.2.1 Implementation based on 1-Variants

Consider HmSearch implemented in the general framework of Algorithm 1, where reducedHammingQuery is based on *1-variants* as signatures. Hence, we will use the indexing and query processing methods described in Section 3.2.1 to implement reducedHammingQuery. The only subtlety is that we index each signature enhanced with its partition ID, so that we can index signatures from different partitions in *one* index without them interfering with each other.

Lemma 4 requires the ability to distinguish between the exact-match with 1-match. We achieve this by the following modification to the postings lists of the inverted index. The inverted index maps a signature sig to I_{sig} which is a list of vectors such that sig is one of their 1-variants. Now we propose to divide vectors in the posting lists into two parts: ones that match sig exactly, and the others that have one error. We denote the former set as $I_{sig}[0]$ and the latter $I_{sig}[1]$. This can be implemented by keeping an additional pointer at the beginning of the postings list which points to the starting entry of $I_{sig}[1]$, as shown in Figure 1. Therefore, if a candidate is returned from $I_{sig}[0]$, it is an exact-match; otherwise it is a 1-match.

Finally, we check the number of matching partitions according to Lemma 4 in the function filter.

The complete listings of the algorithms are given in Algorithms 3 to 5.

Algorithm 3: oneHammingQuery1Var(q)

```
$C \leftarrow \emptyset$;
for each vector ID $v$ in $I_q[0]$ do
    $C \leftarrow (v, 0)$;
for each vector ID $v$ in $I_q[1]$ do
    $C \leftarrow (v, 1)$;
return $C$;
```

Algorithm 4: HmSearch − V(Q, k, κ)

```
/* generate candidates */
$CAND \leftarrow$ empty hash table that maps vector ID to a list of integers;
partition($Q, \kappa$);
for each the $i$-th partition $Q^i$ of the query $Q$ do
    for each vector ID $(v, err) \in$ oneHammingQuery1Var($Q^i$) do
        $CAND[v]$.append($err$);
/* filtering and then verification */
for each candidate $v \in CAND$ do
    if enhancedFilter($v, k$) = false then
        /* See Algorithm 7 for HBVerify */
        if HBVerify($Q, v$) then
            output $v$;
```

Algorithm 5: enhancedFilter(v, k)

```
Output : Returns true if $v$ is filtered (i.e., disqualified)
$errors \leftarrow CAND[v]$;    /* the list of errors */;
if $k$ is even then
    if errors has less than two number then
        if $errors[0] = 1$ then
            return true;
else
    if errors has less than three number then
        if errors has only one number or $errors[0] = 1$ and $errors[1] = 1$ then
            return true;
return false;
```

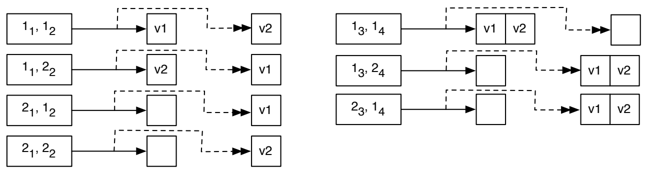


Figure 1: Index for 1-Variants

Example 3. *Consider $N = 4$, $k = 2$, $Q = [1_1, 1_2, 2_3, 2_4]$ and the following data vectors:*

$$v_1 : [1_1, 1_2, 1_3, 1_4]$$
$$v_2 : [1_1, 2_2, 1_3, 1_4]$$

Assume the domain is $\{1, 2\}$. $\kappa = \lceil \frac{2+3}{2} \rceil = 2$, and the first partition is consisted of the first two dimension and the rest two dimensions form the second partition. The variants and the index built for them is shown in Figure 1.

At the beginning, CAND and C are initialized to empty. The query is partitioned into $[1_1, 1_2]$ and $[2_3, 2_4]$. Since $[1_1, 1_2]$ is in the index, all its postings are retrieved. v_1 is in $I[0]$. This means Q and v_1 has an exact-match $[1_1, 1_2]$. We denote this matching as $(v_1, 0)$ and send it to C. Next, v_2 is in $I[1]$, which means Q and v_2 has a 1-match on this partition, so this matching is marked as $(v_2, 1)$ and was sent to C too. Then, in CAND, conditions of matchings are added to each vectors belonging to C (e.g., $error[0]$ denotes the number of errors the first matching incurs). In this case, we have $v_1.error[0] = 0$ and $v_2.error[0] = 1$. Then for v_1, $v_1.errors$ array has length 1, which is smaller than 2. This means it has one match with the query. In addition, $v_1.error[0] = 0$, which means the match is an exact-match, so it cannot be filtered and will be further verified. Next, v_2 is processed. Because $v_2.errors$ array has length 1, which is smaller than 2, it also has one match with the query. However, as $v_2.error[0] = 1$, which means this is not an exact-match, so it is pruned. Next, as $[2_3, 2_4]$ has no match in the index, the query processing finishes.

4.2.2 Implementation based on 1-deletion-Variants

The only major difference to the previous section is the method to distinguish between exact-match and 1-match. For the former case, we know the number of common 1-deletion-variants in a partition p is exactly $|p|$, i.e., the number of dimensions in p. For the latter case, we know the number is exactly 1. So we only need to test if the number is greater than 1 to tell these two cases apart (See Algorithm 6). Hence we can replace oneHammingQuery1Var by oneHammingQuery1DelVar at line 4 of Algorithm 4 to implement HmSearch based on *1-deletion-variants*.

Algorithm 6: oneHammingQuery1DelVar(q)

```
1  $C \leftarrow$ empty hash table that maps vector ID to an integer;
2  for each $1$-deletion-variant $\delta(q)$ of $q$ do
3      for each vector ID $v$ in $I_{\delta(q)}$ do
4          $C[v] \leftarrow C[v] + 1$;
5  $C' \leftarrow$ empty list;
6  for each key $v$ in $C$ do
7      if $C[v] > 2$ then
8          $C' \leftarrow C' \cup (v, 0)$;
9      else
10         $C' \leftarrow C' \cup (v, 1)$;
11 return $C'$;
```

4.3 Hierarchical Binary Filtering and Verification

Another improvement in our HmSearch method is a new algorithm, HBVerify, that can perform additional filtering and verification simultaneously; it is also highly optimized by exploiting vertical layout and bit-parallelism.

Let $d = \lceil \log_2 |\Sigma| \rceil$. We can represent each dimension value of a vector using d bits. For a vector v, we store its dimension values in binary in a *vertical* format, i.e., using N bits to store the most significant bits of all the N dimension values, and another N bits for the second most significant bits, and so on so forth. We use the notation $v_{(i)}$ to denote the array of bits consist of the i-th most significant bits of the dimension values of vector v.

We can derive a filtering condition as follows:

Lemma 5. *If $H(Q, v) \leq k$, then $H(Q_{(i)}, v_{(i)}) \leq k$, $\forall i$.*

Proof. Let $H(v, Q) \leq k$. Assume the contrary, i.e., $H(Q_{(i)}, v_{(i)}) \geq k$, which means $v_{(i)}$ has at least $k + 1$ different bits with $Q_{(i)}$. Since each bit in $v_{(i)}$ belongs to a unique dimension value in v, v has at least $k + 1$ different dimension values with Q, which contradicts the fact that $H(Q, v) \leq k$. □

This filter can be implemented efficiently using bit-level operations exploiting the bit-parallelism offered by CPUs.

- **XOR**: Perform bitwise-XOR between $v_{(i)}$ and $Q_{(i)}$ and obtain a bitmap A. This only requires $\lceil N/w \rceil$ instructions.
- **BitCount**: Count the number of set bits (i.e., `1`s) in the A. This can be done using $\lceil 12N/w \rceil$ machine instructions based on the trick at `http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel`.

Therefore, the filtering can be performed efficiently, exploiting the bit-parallelism.

We can further strengthen the above filter by invoking it in an accumulative fashion over our binary representation (See Algorithm 7). In each iteration, we reuse the XOR'ed bitmap A obtained in the last step (stored in B). We can perform the XOR operations for the current level of bits, and then mask off the bits that are already different in early iterations using bitwise OR. The resulting bitmap will be bit-counted. Therefore, the number of different bits is actually the total number of dimensions where the two vectors are different so far. Obviously, the filtering power is much better than applying the filter alone for the current iteration. We choose to iterate from the least significant bit to the most significant bits, to maximize the probability of filtering (Line 3).

Another benefit of this filtering is that after we iterate over all the d levels, the final bit count number is *exactly* the Hamming distance between the two vectors. So we do *not* need a separate verification stage.

Algorithm 7: HBVerify(Q, S)

```
1  $maxlevel \leftarrow \log_2(|\Sigma|)$;
2  $cumdiff \leftarrow \lceil N/w \rceil$ machine words filled
   with 0x0;  /* $w$ is the size of a machine word in bits */;
3  for $i = maxlevel$ downto 1 do
4      $errs \leftarrow 0$;
5      for $j = 0$ to $\lceil N/w \rceil$ do
6          $diff \leftarrow Q_{(i)}[j] \oplus v_{(i)}[j]$;       /* XOR for diffs */;
7          $cumdiff[j] \leftarrow cumdiff[j] \vee diff$;        /* OR */;
8          $errs \leftarrow errs +$ popcount($cumdiff[j]$);
           /* count set bits */;
9      if $errs > k$ then
10         return false;
11 Output $(v, errs)$;
12 return true;
```

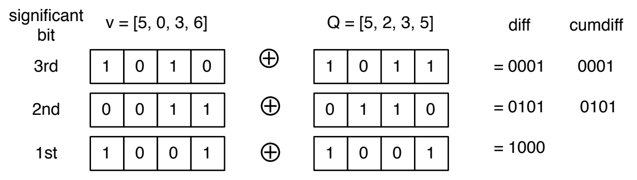


Figure 2: Example of Hierarchical Binary Filtering and Verification

Example 4. *Consider the vector $v = \{5, 0, 3, 6\}$ and the query $Q = \{5, 2, 3, 5\}$ in vertical binary representation in Figure 2. Let $N = 4$, $k = 1$, $|\Sigma| = 8$, and $w = 4$. We first filter-and-verify the 3rd most significant bits of Q and v. There are 1 mismatch between* `1010` *and* `1011`*. The cumulative difference bitmap cumdiff in Algorithm 7 is* `0001`*. After bit counting, the total number of errors is 1, which is no larger than $k = 1$. So we move on to the 2nd most significant bits of Q and v. $diff = 0011 \oplus 0110 = 0101$. The cumdiff is then OR'ed with the diff and produce curdiff $= 0101$, which has 2 bits set. This means $H(Q, v) \geq 2$ and hence we can prune v immediately.*

4.4 Complexity Analysis

We list the time and space complexities of previous and our methods in Table 2.

5. REARRANGE DIMENSIONS

In this section, we present our dimension rearrangement technique to handle skewed datasets.

Algorithm	Query Time	Index Size
[16] (1 level part.)	$(k+1) \cdot f(\frac{N}{k+1}) + vc_1$	$(k+1) \cdot n$
[16] (2 level part.)	$(k+1)^2 \cdot f(\frac{(2k+1)N}{(k+1)^2}) + vc_2$	$(k+1)^2 \cdot n$
[14]	$N \cdot \lvert\Sigma\rvert \cdot g_1(\frac{2N}{k}) + vc_3$	$N \cdot n$
HmSearch 1-var	$\frac{k}{2} \cdot g_1(\frac{2N}{k}) + vc_4$	$N \cdot \lvert\Sigma\rvert \cdot n$
HmSearch 1-del-var	$N \cdot g_2(\frac{2N}{k} - 1) + vc_5$	$N \cdot n$

where $f(x) = \max\left(1, \frac{n}{|\Sigma|^{N-x}}\right)$, $g_1(x) = \max\left(1, \frac{n \cdot |\Sigma| \cdot N}{|\Sigma|^{N-x}}\right)$, and $g_2(x) = \max\left(1, \frac{n \cdot N}{|\Sigma|^{N-x}}\right)$, under the uniform assumption. vc_i stands for the total time used for pruning and verifying the candidates in each algorithm.

Table 2: Complexities of Empirical Hamming Distance Query Methods

5.1 Impact of Data Skewness

Consider a partition of l dimensions with n vectors. A data skew exists if one of the $|\Sigma|^l$ values occurs very frequently (e.g., close to n times). If the corresponding partition value of the query is exactly this frequently-occurring value, then the majority of the vectors will become candidates. Such a large amount of candidate will make the algorithm degenerate to a brute-force linear scan algorithm.

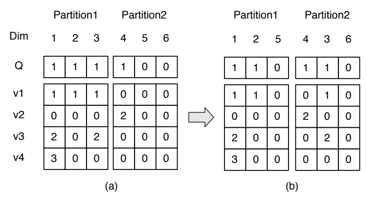


Figure 3: Impact of Data Skew and Benefit of Dimension Rearrangement

EXAMPLE 5. *Consider the example dataset in Figure 3(a). $N = 6$ and $k = 1$, so $\kappa = 2$. Since all the vectors' second partitions are within Hamming distance of 1 with the query's second partition, all of them will become candidates. However, if we permute the dimensions before the partitioning as in Figure 3(b), the only candidate is v_1.*

5.2 A Greedy Dimension Rearrangement Algorithm

As the problem to obtain the optimal dimension rearrangement is likely to be a hard problem, instead, we resort to a bottom-up, greedy algorithm to find a reasonably good rearrangement for a specific κ. Assuming that we have a way to measure the quality of a partition, the general idea of the algorithm is:

- Initially we form κ partitions, each consisting of the "worst" single dimensions (in terms of quality) among the remaining dimensions.
- In each of the $N - \kappa$ rounds, we choose a worst partition, and add one of the remaining dimensions to this partition such that the resulting new partition has the best possible quality.

Now we consider how to define the "quality" of a partition. Consider a partition $\mathcal{D}$ consisting of l dimensions. Since we do not know the query's value on this partition *a priori*, we choose to minimize the maximum frequency of any values occurring in these dimensions, i.e., $\mathit{MaxFreq}(\mathcal{D}) \overset{\text{def}}{=} \max_{x \in \Sigma^l} |\{ v_i \in DB \mid v_i[\mathcal{D}] = x \}|$. Minimizing *MaxFreq* also contributes to minimizing the candidate size for a query in the worst case. Let $\mathcal{D} \circ \{ D_j \}$ denotes the partition formed by adding dimension D_j to $\mathcal{D}$. We can choose the dimension D_j such that it results in the smallest $\mathit{MaxFreq}(\mathcal{D} \circ \{ D_j \})$.

The complexity of this greedy algorithm is $O(N^3 \cdot n)$. While it could have a long running time when N is large, it only needs to be run once for a fixed dataset. Also to reduce its running time for large N, we run it on a sample of the dataset. Finally, the efforts in dimension rearrangement are worthwhile as it is shown in our experiment (See Section 6.6).

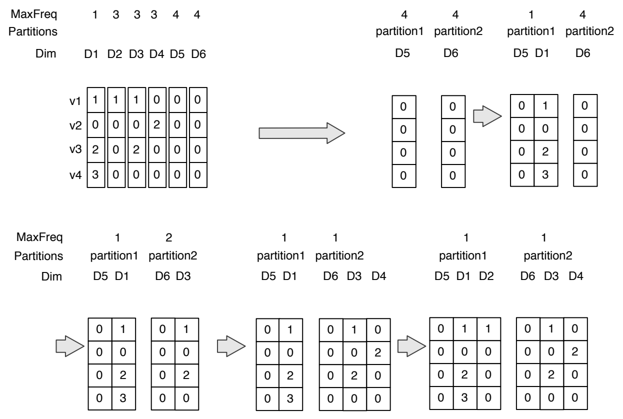

MaxFreq 1 3 3 3 4 4
Partitions
Dim D1 D2 D3 D4 D5 D6
v1 v2 v3 v4
4 4 partition1 partition2 D5 D6
1 4 partition1 partition2 D5 D1 D6
MaxFreq 1 2 Partitions partition1 partition2 Dim D5 D1 D6 D3
1 1 partition1 partition2 D5 D1 D6 D3 D4
1 1 partition1 partition2 D5 D1 D2 D6 D3 D4

Figure 4: Dimension Rearrangement Example

EXAMPLE 6. *We illustrate the process of running the dimension rearrangement algorithm on the dataset in Example 5 for $\kappa = 2$ in Figure 4. Initially, the MaxFreqs of the single dimensions are first computed. We pick the worst two to start the partitions. Then we consider the best dimension to add to partition 1 (currently only D_5) such that the resulting MaxFreq is minimized; we found D_1, and this results in the MaxFreq of 1 for the new partition $\{ D_5, D_1 \}$. The process runs until all remaining dimensions have been distributed to one of the partitions.*

6. EXPERIMENTS

In this section, we report findings in our extensive experimental study. We first compare the performance of our proposed algorithms with three state-of-the-art methods for Hamming distance queries. Then we evaluate our dimensions rearrangement method to show its resulting performance improvement. Finally, we analyze the scalability and index size of our methods.

6.1 Experiment Setup

The following algorithms are used in the experiment.

- **HSD, HSV** are our proposed algorithms. HSD generates *1-deletion-variants* as signatures. HSV generates *1-variants* as signatures. Both algorithms employ all three techniques we proposed, including Enhanced Filter (EF), Hierarchical Binary Filter (HB) and Rearranging Dimensions (RD). HSD-nEB and HSV-nEB are two variations that remove EF and HB techniques from HSD and HSV, HSD-nB and HSV-nB are another two variations that only remove HB from HSD and HSV. HSD-nR and HSV-nR only remove RD from HSD and HSV.

- **ScanCount** [13] is an index merge method that scans through the posting list of each element of query and count the occurrences of data IDs. We use it as a baseline method. Note that when dealing with Hamming distance constraint using ScanCount, 0s in each vector must also be indexed and processed to guarantee the correctness.
- **Google** [16] is one of the state-of-the-art Hamming distance query algorithms and is specifically designed for detecting near duplicate documents at Web scale. This method is based on partitioning and exact-matching. We also implemented a variation of Google, Google-R, which integrates the Rearranging Dimensions (RD) technique we proposed.
- **HEngine** [14] is a recently proposed Hamming distance query processing method. It is based on partitioning and reducing the k-query to 1-queries.

In our experiments, we select four publicly available real datasets. They cover a wide range of data distributions and application domains.

- **Audio** is extracted from the DARPA TIMIT collection[1]. It contains 54,387 192-dimensional feature vectors. We use p-stable LSH [9] to convert each feature vector into a 64 dimension integer vector.
- **TREC** is extracted from the TREC-9 Filtering Track Collections[2]. Each string is a reference from the MEDLINE database with author, title, and abstract information. We apply the SimHash [16] to convert each string into a 64-dimension binary vector.
- **ENRON** is extracted from the Enron email collection[3]. We extract and concatenate the email title and body. We employ MinHash [5] to convert each string a 64-dimension integer vector. As MinHash selects a token in the string as its signature, the $|\sigma|$ of ENRON is large.
- **PubChem** is a database of chemical molecules[4]. We sample 1 million entries. Each entry contains a fingerprint, which is a 881-dimension binary vector.

Statistics about the datasets are listed in Table 3.

The experiments for Audio, TREC, ENRON data were carried out on a PC with Intel(R) Xeon(R) X3330 2.66GHZ CPU and 4GB RAM. The operation system is Debian 5.0.6. The experiments for PubChem data were carried out on a PC with Quad-Core AMD Opteron(tm) Processor 8378 2.4GHZ CPU and 96GB RAM and the operation system is Ubuntu/Linaro 4.6.3-1ubuntu5. All algorithms were implemented in C/C++ and compiled using GCC 4.4.5 with -O3 flag. All algorithms run in in-memory mode.

[1]http://www.cs.princeton.edu/cass/audio.tar.gz

[2]http://trec.nist.gov/data/t9_filtering.html

[3]http://www.cs.cmu.edu/~enron/

[4]http://pubchem.ncbi.nlm.nih.gov/

We measured the query time and candidate size in the experiments. By query time, we mean the average elapsed time (measured in millisecond) for a query. Due to the wide range of values, the y-axes of most figures on running time are plotted in logarithmic scale. The candidate size we measure is the average number of data vectors that are sent to the final verification.

Table 3: Statistics of Datasets

Data	n	N	**Generation Function**	$\lvert\Sigma\rvert$
Audio	$54,387$	64	2-stable LSH	16
TREC	$239,580$	64	SimHash	2
ENRON	$95,997$	64	MinHash	172
PubChem	$1,000,000$	881	chemical fingerprinting	2

6.2 Hamming Similarity Query Performance

To test the query processing time of all algorithms on four datasets, we randomly sample 1,000 vectors from each dataset as queries. We measure the query time and show the results of five algorithms in Figures 5(a)–5(d). For Audio, TREC and ENRON datasets, the Hamming distance threshold varies from 1 to 31 (nearly 50% error rate). For the PubChem dataset, the Hamming distance threshold varies from 1 to 81 (nearly 10% error rate).

We observe that

- The query performance on Audio, TREC and PubChem exhibits following patterns.
  - The fastest algorithm is HSV for all Hamming distance thresholds.
  - For small threshold (less than 7), Google is better than HSD. On the other hand, when the threshold gets larger (than 7), HSD outperforms Google by up to 2 orders of magnitude.
  - When Hamming distance threshold is 1, HSV and Google have the similar performance, as both methods use highly selective signatures. When Hamming distance threshold increases, the performance of Google deteriorates faster than HSV. The reason is that HSV's partition length is nearly twice as long as that of Google's partition. Hence HSV generates much more selective signatures and this results in HSV's better performance.
  - The slowest algorithm is always ScanCount and it is insensitive to the Hamming distance threshold. This is because ScanCount always naïvely goes through all the posting lists for each dimension values of the query and collects the number of occurrence of each vector encountered.
- ENRON has a large alphabet size, hence HSV becomes inapplicable. We compare HSD with other algorithms in Figure 5(c). The trend is,
  - HSD has a competative performance from middle (10) to large (31) Hamming distance thresholds.
  - When the threshold is low, for instance, up to 7, Google outperforms HSD in most experiment cases. This is because when the threshold is low, both Google and HSD generate highly selective signatures. Therefore, the advantage of longer signatures of HSD is not obvious. In the meanwhile, the overhead of HSD enumerating 1-deletion variants on the query contributes to the slowing down the query time.

  - HEngine has substantially worse performance on ENRON, because it needs to generate a large amount of the query's 1-variants and probing them against the index. This cost is proportional to the alphabet size $|\sigma|$.

- The overall trend for HSV, HSD and Google is that the query time increases with the increase of the Hamming distance threshold. This is reasonable, as larger threshold leads to more candidates and eventually more results which increase the computation time.

6.3 Candidate Size Analysis

We measure the candidate sizes of four algorithms on the four datasets and show the results in Figures 5(e)–5(h).

We observe that

- Except for ScanCount, the candidate sizes of other algorithms increase with the increase of the Hamming distance threshold. Google has a larger candidate size than HSV and HEngine. The reason is that Google's partition length is about half of that of HSV, HSD and HEngine.
- When the Hamming distance threshold increases, the candidate sizes of HSV and HSD grow much slower than other algorithms thanks to the enhanced filtering.
- The candidate sizes of all partitioning-based methods will reach n (i.e., all data vectors become the candidates) when the threshold is sufficiently large. When an algorithm's candidate size reaches n, it is better off to use a brute-force verification-only method, and hence this is the maximum error threshold the algorithm could be useful for. For Google, this happens when the threshold is around 25 on Audio and ENRON, and when thresholds are 10 and 17 on TREC and PubChem, respectively. This phenomenon occurs much later for both HSV and HSD than Google and HEngine.

6.4 Query Time Fluctuation

The overall trend of the query time is that it increases when the Hamming distance threshold increases. However, as shown in Figure 5(i), on the micro-scale, the query time of our HSV may fluctuate. For example, query time at $k = 26$ is slightly more than that at $k = 27$. This phenomenon is caused by the enhanced filtering due to Lemma 4. When the Hamming distance constraint is even, in certain conditions, two variant matches are required to pass the filtering condition. However, when the threshold increase by one, the filtering condition may be strengthened to requiring three matches. Although the increase of the threshold shortens the partition length and accordingly the selectivities, the stronger pruning condition may eventually reduces the candidate size substantially and hence improve the overall performance.

6.5 Effect of Enhanced Filter and Hierarchical Binary Verification

We present the query time and candidate size of our algorithms to exhibit the effects of Enhanced Filter and Hierarchical Binary Verification. HSV denotes the algorithm that contains both Enhanced Filter and Hierarchical Binary Verification. HSV-nB denotes the algorithm with only Enhanced Filter and HSV-nEB denotes the algorithm without either technique. The query times are shown in Figures 5(j) and 5(k), and the corresponding candidate sizes are shown in Figures 5(l) and 5(m).

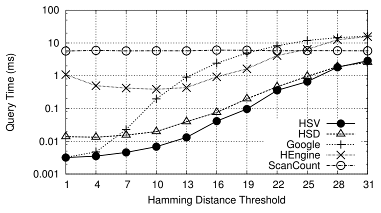
For HSV-nEB and HSV-nB, Enhanced Filter contributes significantly to performance improvement when the threshold is in the middle range. For example, when threshold is between 13 and 28, HSV-nB has almost one order of magnitude faster than HSV-nEB. The reason is that at these thresholds, the selectivity of variants is low, thus requiring two or three matching partitions helps improve the performance dramatically. The same trend also appears in Figures 5(l) and 5(m). We can notice that the average candidate size reduction is more significant than the reduction of running time. For example, when the Hamming distance threshold equals to 16 on Audio data, there is a nearly 90% reduction of candidate size with a nearly 70% reduction of running time. This is mainly due to the overhead of performing the filtering. Another observation is that when the threshold is very small, the improvement due to the Enhanced Filter is insignificant. For example, for Enron, when the Hamming distance threshold is 1, both HSV-nEB and HSV-nB have almost the same performance. The reason is that since the threshold is very small, the variants of HSV-nEB are long enough to have a very high selectivity.

By comparing HSV-nB (HSD-nB) and HSV (HSD), we can evaluate the effectiveness of Hierarchical Binary Verification. The improvement of applying Hierarchical Binary Verification is noticeable in both datasets, especially when the Hamming distance threshold is not small. Generally speaking, the performance gap between the traditional verification and the Hierarchical Binary Verification enlarges with the increase of the Hamming distance threshold. For example, when the Hamming distance threshold is 22 for Audio data, the performance improvement over traditional verification is over 4 times.

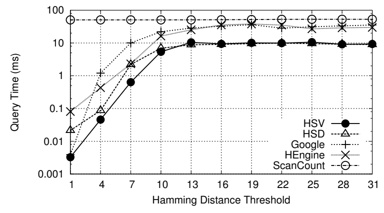
6.6 Effect of Rearranging Dimensions

We study the effectiveness of rearranging dimensions in Figures 5(n)–5(q). Note that Google-R is the Google with the dimension rearrangement technique, and HSV (HSV-nR) and HSD (HSD-nR) are our algorithms with and without the dimension rearrangement technique, respectively. The following observations can be made:

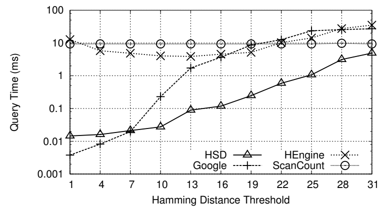
- The effect of dimension rearrangement can boost the performance in most cases, especially for the PubChem data (up to two orders of magnitude). The reason is that since each dimension corresponds to a manually defined feature of chemical molecules, there are plenty of skews in the PubChem dataset. In addition, it is not uncommon that several of these skewed dimensions are consecutive and may reside in the same partition by existing methods. This may lead to the fact that the majority of the dataset will be retrieved as candidates.
- For the Audio and TREC datasets, the effect of dimension rearrangement is noticeable but not significant (See Figures 5(n) and 5(o)). The reason is that the dimensions of these datasets are generated by various independent LSH functions, therefore, there is much less data skew in the Audio and TREC datasets. Hence, the improvement of dimension rearrangement is not as remarkable as PubChem.



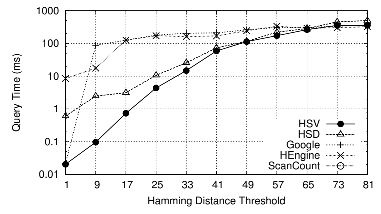
(a) Audio, Query Time ($N = 64$)



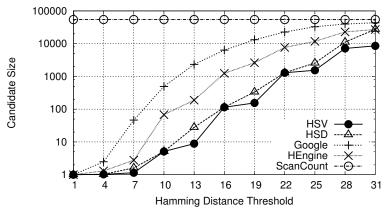
(b) TREC, Query Time ($N = 64$)



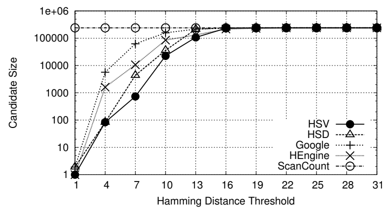
(c) ENRON, Query Time ($N = 64$)



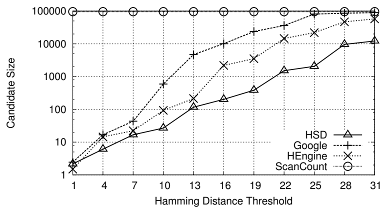
(d) PubChem, Query Time ($N = 881$)



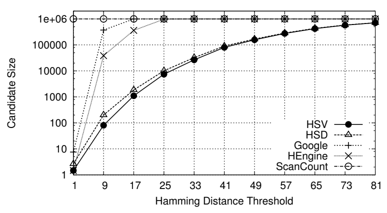
(e) Audio, Candidate Size ($N = 64$)



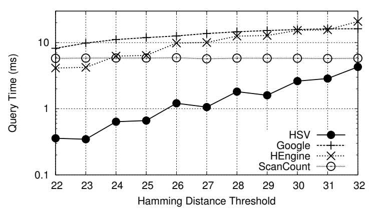
(f) TREC, Candidate Size ($N = 64$)



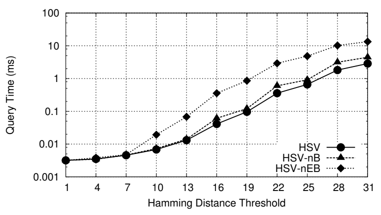
(g) ENRON, Candidate Size ($N = 64$)



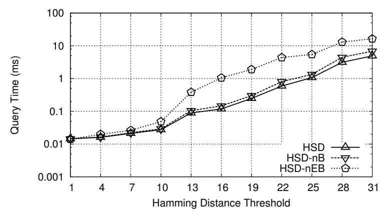
(h) PubChem, Candidate Size ($N = 881$)



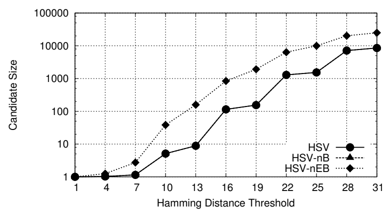
(i) Audio, Query Time Fluctuation



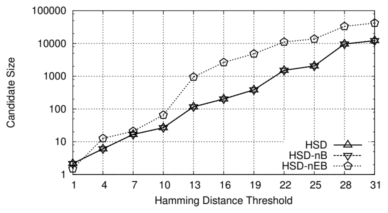
(j) Effect of EF and HB, Audio, Query Time



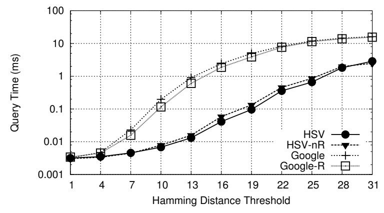
(k) Effect of EF and HB, ENRON, Query Time



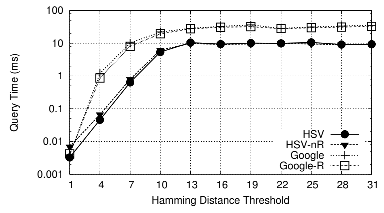
(l) Effect of EF and HB, Audio, Candidate Size



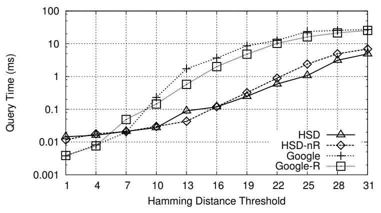
(m) Effect of EF and HB, ENRON, Candidate Size



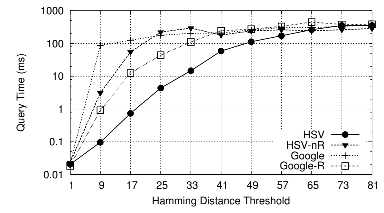
(n) Effect of RD, Audio, Query Time



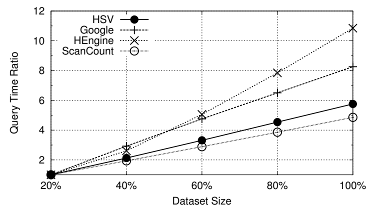
(o) Effect of RD, TREC, Query Time



(p) Effect of RD, ENRON, Query Time



(q) Effect of RD, PubChem, Query Time



(r) Scalability, TREC, Query Time Ratio

Figure 5: Experiment Results - I

- Although our dimension rearrangement method works in most cases, it does not always deliver better performance than without. For example, for the ENRON dataset, when the threshold is between 5 and 8, Google has a better performance than Google-R (see Figure 5(p)). The reason is that under such cases, the variants already have very good selectivites and our greedy algorithm cannot guarantee the global optimality of the dimension rearrangement.

6.7 Scalability

We study the scalability of the algorithms by varying the dataset size. We randomly sample 20% to 100% of the data vectors from TREC as the datasets for this experiment. The Hamming distance threshold is fixed to 7. Figure 5(r) shows the *query time ratio*, which is defined as the query time of the current dataset over the query time of the 20% sampled dataset.

The general trend is that the query time of the four algorithms all grows with the increase of the dataset size. ScanCount exhibits the slowest growth rate, followed by HSV. When the dataset size increases from 20% to 100%, the query time increases by 4.5 times for ScanCount, 5.9 times for HSV, 8.2 times for Google and 11 times for HEngine. This showcases the better scalability of HSV and ScanCount to dataset size than that of Google and HEngine.

6.8 Index Size

Figures 6(a)–6(c) show the index sizes of the algorithms on the three datasets[5] with different Hamming distance thresholds. In general, our HSV has a large index size. For TREC data, when the threshold is small ($k = 1$), the index size is even larger than the case where the threshold is large ($k > 25$). The reason is that, for small k, a large amount of unique signatures are generated by HSV, each of which requires two pointers. Hence the total size of pointers is huge and contributes to the large index size. The HSD and ScanCount have a competitive space usage on ENRON dataset, yet for the TREC and PubChem datasets, they consume a relatively larger index space. Note that, in some cases (e.g., $k \geq 25$ on ENRON), HSD has the smallest index size among all the methods. Generally speaking, Google has a relative small index size for most cases. The HEngine's index size increases linearly with the increasing threshold, and is usually larger than that of HSD.

7. RELATED WORK

Theoretical Studies. The Hamming distance query with threshold k was originally known as the k-query problem. [17] first proposed the d-query problem, which asks if there exists a string in a dictionary (consisting of n strings) within Hamming distance d of a given (binary) query string Q of length m.

Different solutions are required for small d and large d. For the special case when $d = 1$, there exist many efficient solutions [26, 3, 4]. Among them, [4] constructs a data structure that answers 1-query in time $O(1)$ using space $O(n \log m)$ in a cell probe model with word size m [4]. d-query for large d is much harder, with few results beating the naive solution with $O(m^d)$ query time. The state-of-the-art result is obtained by [8], which answers a d-query where $d = O(1)$ in time $O(m + \log^d(nm) + occ)$ and using space $O(n \log^d(nm))$, where occ is the number of query result.

[5]The results on the Audio dataset is similar to that of ENRON.

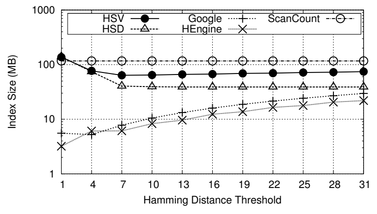
Solutions in Chemical Informatics. Similarity queries with a Tanimoto threshold on binary fingerprints of chemicals are widely used in chemoinformatics applications [10, 7, 6, 19]. There exist many specialized solutions [23, 2, 18, 22, 20]. Most of the solutions are based on bounding the number of 1-bits in the fingerprints or their partitions. [23] develops the bound on the number of 1-bits given a query fingerprint; and [18] further applies this idea to partitioned fingerprints. Another 1-bit bound is developed in [2] where fingerprints are "folded" down to shorter fingerprints via the XOR operation, and a bound of the similarity can be established on the short fingerprints. [22] builds a method named MultiBit Tree, which is a binary tree recursively built by choosing a certain dimension to split the remaining fingerprints. At query time, a depth first traversal on the tree is performed together with pruning based on the number of 1-bits. One of the latest methods is [20], where each fingerprint is transformed into a set (as in Section 2) and inverted index is built on the set elements. This essentially reduces the original problem into a *set overlap search* problem, where the DivideSkip method proposed in their earlier work [13] is employed for query processing.

Practical Solutions. Due to the wide range of applications of Hamming distance queries (e.g., those mentioned in Section 1), many practical solutions have been proposed, and they are all based on reducing the k-query problem into several k'-query sub-problems, where $k' < k$. [15] essentially indexes all the 1-variants of strings in the dictionary to answer 1-query efficiently. To handle small k, [16, 24] divide the string into $k + 1$ partitions such that query results must have at least one exact match with the query in one of the partitions. [16] also proposes methods to recursively apply the same idea again, and this two-level partitioning idea also appears in the PartEnum method [1]. The above methods can only deal with very small value of k, as the number of dimensions in each partition will be small and this results in poor selectivities. Recent work addresses this limitation by reducing the general problem into several 1-query sub-problems [14, 21]. The number of partitions κ is chosen to be $\lfloor k/2 \rfloor + 1$[6]. The difference between the two proposals is that [14] replicates the data while [21] resorts to indexes, which was implemented and compared with in our experiments. We note that the approaches that reduce to 1-queries (including ours) are better than those reducing to 0-query using the two-level partitioning methods [16, 1] as they have the similar signature length ($\frac{2}{k+2}N$ vs. $\frac{2k+1}{(k+1)^2}N$), but the latter generates much more signatures per vector ($k/2$ vs. $(k + 1)^2$); therefore, we do not compare with the latter in our experiment.

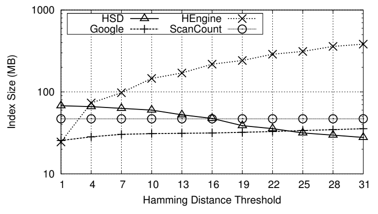
8. CONCLUSIONS

In this paper, we propose HmSearch, an efficient Hamming distance query algorithm that works well for a large spectrum of error threshold, do not have any limitation on the

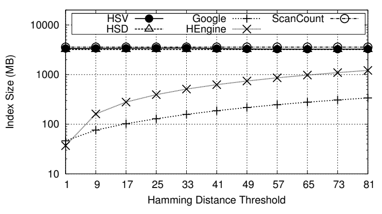
[6]Although both papers mentioned the possibility of $k' > 1$, this will result in very large index as it is superlinear in the number of dimensions.



(a) TREC, Index Size



(b) ENRON, Index Size



(c) PubChem, Index Size

Figure 6: Experiment Results - II

domain size, and is robust against data skew. Our method is based on a different partitioning scheme, with tightened count filtering and a filter-and-verification technique based on hierarchical binary representation. A greedy algorithm to rearrange the dimensions before partitioning is also developed. We demonstrate the superior performance of our proposed method against the previous state-of-the-art methods, using LSH and Chemical datasets under a wide range of parameter settings.

Acknowledgement

Jianbin Qin and Wei Wang are partially supported by ARC DP Projects DP130103401 and DP130103405. Jiaheng Lu is partially supported by NSF China (No. 60903056), Beijing Natural Science Foundation (No. 4112030), and New Century Excellent Talents in University.

9. REFERENCES

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

[2] P. Baldi, D. S. Hirschberg, R. J. Nasr, P. Baldi, D. S. Hirschberg, and R. J. Nasr. Speeding up chemical database searches using a proximity filter based on the logical exclusive-or. *J. Chem. Inf. Model*, pages 1367–1378, 2008.

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

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

[5] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[6] B. Chen, D. Wild, and R. Guha. Pubchem as a source of polypharmacology. *Journal of Chemical Information and Modeling*, 49(9):2044–2055, 2009.

[7] J. Chen, S. J. Swamidass, Y. Dou, and P. Baldi. Chemdb: a public database of small molecules and related chemoinformatics resources. *Bioinformatics*, 21:4133–4139, 2005.

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

[9] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.

[10] 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.

[11] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *SIGMOD Conference*, pages 541–552, 2012.

[12] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

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

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

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

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

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

[18] R. Nasr, D. Hirschberg, and P. Baldi. Hashing algorithms and data structures for rapid searches of fingerprint vectors. *J. Chem. Inf. Model*, 50(8):1358–68, 2010.

[19] R. Nasr, S. J. Swamidass, and P. Baldi. Large scale study of multiple-molecule queries. *J. Cheminformatics*, 1:7, 2009.

[20] 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.

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

[22] P. B. R. Nasr, T. Kristensen. Tree and hashing data structures to speedup chemical searches: Analysis and experiments. *Molecular Informatics*, 30(9):791–800, 2011. Special Issue on Machine Learning Methods in Chemoinformatics/NIPS.

[23] S. Swamidass and P. Baldi. Bounds and algorithms for fast exact searches of chemical fingerprints in linear and sublinear time. *J Chem Inf Model*, 47(2):302–17, 2007.

[24] 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.

[25] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR*, pages 563–570, 2008.

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