An Index Structure for Fast Range Search in Hamming Space

E.M. Reina
Faculty of Science, UOIT
Oshawa Canada
ernesto.reina@uoit.net

F.Z. Qureshi and K.Q. Pu
Faculty of Science, UOIT
Oshawa Canada
{*faisal.qureshi, ken.pu*}*@uoit.ca*

***Abstract*—This paper addresses the problem of indexing and querying very large databases of binary vectors. Such databases of binary vectors are a common occurrence in domains such as information retrieval and computer vision. We propose an indexing structure consisting of a compressed bitwise trie and a hash table for supporting range queries in Hamming space. The index structure, which can be updated incrementally, is able to solve the range queries for any radius. Our approach significantly outperforms state-of-the-art approaches.**

***Keywords*-trie indexing, hashing, binary vectors**

I. Introduction

Increasingly many applications in domains ranging from image matching to information retrieval generate and analyze very large number of multidimensional feature descriptors [16], [15], [27], [26], [4], [14], [28], [20], [23], [10], [29]. These feature descriptors are sometimes encoded as binary vectors, since binary vectors can be efficiently stored, indexed, and searched. Given a set of binary vectors D and a query vector $\mathbf{q}$, finding exact matches is often not sufficient. Rather most applications seek to find the subset of D that are within some distance of the query vector $\mathbf{q}$ [22]. Hamming distance is a widely used distance metric over binary vector spaces. The Hamming distance between two l-bit vectors $\mathbf{x}, \mathbf{y} \in 2^l$ is defined as: $H(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{l-1} (\mathbf{x}[i] \oplus \mathbf{y}[i])$, where $\oplus$ is the XOR operator. 2^l is the set of all l-bit binary vectors. We refer to the problem of finding vectors in D that are within some Hamming distance of $\mathbf{q}$ as the *r-neighbours search problem* (r-NSP).

Definition 1. *The r-neighbours search over a set of binary vectors $D \subseteq 2^l$, given the query binary vector $\mathbf{q} \in 2^l$ and a radius $r \in \mathbb{Z}^+$, is defined as finding all vectors in D that are at most at Hamming distance r of $\mathbf{q}$: $N_r(D, \mathbf{q}) = \{\mathbf{d} \in D : H(\mathbf{d}, \mathbf{q}) \leq r\}$.*

Many schemes for solving r-NSP assume that r is fixed, i.e., all queries use the same value of r that is known beforehand. This is sometimes referred to as the static r-NSP. We are interested in, the so called, dynamic r-NSP. r is not known *a priori*. r in this case is part of the input [17]. It is worth mentioning that schemes developed for static r-NSP perform poorly for dynamic r-NSP. The remainder of this paper, simply refers to the r-NSP, dropping the term "dynamic" in the interest of brevity.

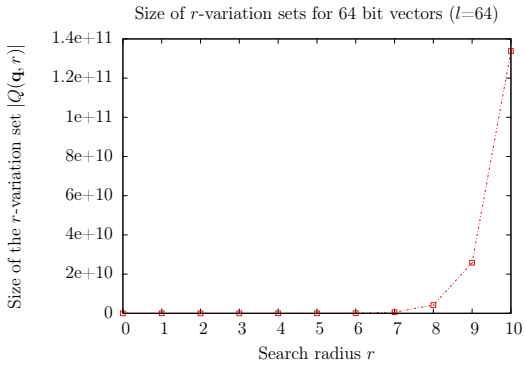

Figure 1. Size of the r-variations set for 64 bits vectors. Notice the size grows exponentially with increasing values for r.

r-NSP, for example, arises naturally in image matching, search, and retrieval [16]. First, each image in the collection is encoded as a set of local binary descriptors [24], [7], [1]. Next, binary descriptors collected over the entire collection are indexed into a data structure that supports fast r-neighbours queries or k-nearest-neighbours queries. An inverted index matches each stored vector (descriptor) to the image that contains it. Binary descriptors computed from the query image are compared against the database to find the set of "closest" vectors in the database. Each of these vectors point to an image in the collection and voting is performed to identify the image (stored in the collection) that best matches the query image. Details can be found in Landré & Truchetet [16]. The key challenge here is one of scale. For example, say we encode each image using 10^3 128-bit binary vectors. This suggests that even a moderate size collection of one million images will have 10^9 binary vectors. At query time, for each image, a naïve approach would require 10^{12} comparisons. Clearly, we need efficient methods to deal with this problem.

Hash tables have been used for r-NSP. The basic idea is as follows. For a given r, all r-variations of the query vector $\mathbf{q}$ are generated and checked against the hash table. $\mathbf{q}' \in 2^l$ is an r-variation of a binary vector $\mathbf{q} \in 2^l$ iff $H(\mathbf{q}, \mathbf{q}') \leq r$. The set of all r-variations of a binary vector $\mathbf{q}$ is $Q(\mathbf{q}, r) =$

$\{\mathbf{q}' \in 2^l : H(\mathbf{q}, \mathbf{q}') \leq r\}$, where $r \in [0, l]$. The cardinality of set $Q(\mathbf{q}, r)$ is

$$|Q(\mathbf{q}, r)| = \sum_{z=0}^{r} \binom{l}{z}, \tag{1}$$

which grows exponentially with l and r (Fig. 1). Consequently, the naïve approach of generating all r-variations and matching them against the hash table only works for small values of l and r, say, for $l < 16$ and $r < 2$.

Current best scheme for dealing with r-neighbours search—and its close variant, k-nearest-neighbours search—is *Multi-Index Hashing* (MIH) [22], which deals with the growth in r-variations through partitioning. MIH defines a partition that divides the binary vector (originally of length l) into sub-vectors (each of length less than l, obviously). A hash table is constructed for each partition, storing sub-vectors from that partition. The query vector is also partitioned and r-variations of sub-vectors are matched against individual hash tables. MIH works well because it generates far fewer r-variations, since the lengths involved are smaller.

Hash based schemes, including MIH, however, do not eliminate the problem entirely and cannot easily deal with high-dimensional binary vectors (say $l > 128$). This is due to the fact that these approaches end up doing a lot of wasted work. Hash based schemes generate r-variations of the query vector (or sub-vectors constructed from the query vector) and check these variations against the hash table. The size of r-variations grows much faster than the size of database, suggesting that a majority of hash table lookups will end up at an empty bucket. It is easy to confirm it by noticing that, in general, $2^l \gg |D|$, where $|D|$ is the total number of l-bit vectors stored in the database. For example, the number of r-variations for $l = 64$ and $r = 10$ is more than 133 trillions (1.33×10^{11}). If there are 1 million different vectors in the hash table, the maximum number of non-empty buckets is 1 million (assuming no collisions). That implies that the great majority of lookups performed using the näive approach will go to empty buckets. We need a way to avoid (or at least reduce) generating r-variations that do not exist in the hash table.

Hash tables alone cannot deal with the problem identified above. Hash tables can efficiently answer membership queries (i.e., these exhibit *lookup efficiency*). Hash tables, however, do not support *local searchibility*, i.e., given a vector, there is no easy way to check if its neighbours in Hamming space exist in the hash table. We propose to use a trie data structure, which organizes the vectors according to their prefixes. Trie exhibits good *local searchibility* properties, as vectors that share a prefix are easily discovered from each other. We develop a hybrid data structure that indexes binary vectors using both a trie and a hash table, and we propose a query processing algorithm that switches back and forth between the two indexing structures, generating far fewer r-variations that need to be checked against the hash table. Similar to MIH, the proposed method also benefits from partitioning. Experimental evaluation demonstrates that our method achieves state-of-the-art results.

II. Related Work

Several approaches have attempted to use trie data structure for r-NSP in Hamming space. Brodal and Gasieniec, for example, use trie to solve static r-NSP for $r = 1$ [6]. Arslan and Egecioglu proposed to use trie to solve dynamic r-NSP (for values of $r \in [0, 2]$) [3], [2]. MaaB and Nowak also propose a similar scheme for dynamic r-NSP for $r \in [0, 2]$ [18]. These schemes suffer from two shortcomings: 1) they can only deal with small values of r and 2) they require that the binary vectors fit within a machine word. Clearly, we need methods that can deal with binary vectors of arbitrary lengths.

The second class of methods for r-NSP leverages hierarchical decomposition of search space. Yao and Yao, for example, propose a binary search tree inspired data structure for static r-NSP [30]. Their method, however, only allows for $r = 1$. Geometric Near-Neighbour Access Tree, or GNAT, was proposed by Brin in 1995 [5]. GNAT has been applied to r-NSP in many metric spaces, including Hamming space. Brin's method, however, only solves approximate r-NSP, and is suitable only for small l and r. Muja and Lowe proposed an algorithm based on k-medoids decomposition of the search space [21]. This algorithm does not attempt to give an exact answer and some r-neighbours are missed. Liu et al. present a method that extends the idea of Muja and Lowe. Their method searches not only the query but a small set of its r-variations (say p perturbations of the query vector $\mathbf{q}$, where $p \approx 10 \ll |Q(\mathbf{q}, r)|$) [17]. These methods either only solve static r-NSP [30] or solve approximate dynamic r-NSP and that for small values of r [5], [21], [17].

Locality Sensitive Hashing (LSH) schemes use hash tables for r-NSP problems. The key operation here is to generate all r-variations of the query vector and intersect this set with the set of vectors D using hash lookups. [13] is a widely used hashing technique that uses a projection operation to project input vectors into a lower dimensional space. LSH methods builds upon the idea that if two vectors are close in Hamming space then their projections will be close in Hamming space as well. Projection defines a partition over the input vector, which can be used to break the vector into sub-vectors. An index is setup for each partition, and all sub-vectors belonging to a partition are indexed in its corresponding index. At search time, the query vector is divided into sub-vectors using the same partition and its sub-vectors are searched in the corresponding indexes. The results obtained from different indexes are scanned linearly to answer the original query.

A simple scheme is to use a random projection operator [25], [12], [11]; however, this reduces accuracy. Actu-

ally, accuracy decreases with increasing values of r. LSH techniques that construct non-overlapping set of sub-vectors from the input vectors have also been explored. Notice that it is possible to divide the vector in m non-overlapping sub-vectors of $\lfloor \frac{l}{m} \rfloor$ or $\lceil \frac{l}{m} \rceil$ bits. Then, if two binary vectors differ in r bits, there are at least $p = m - \left\lfloor \frac{r}{\lfloor \frac{r}{m} \rfloor + 1} \right\rfloor$ sub-vectors that differ in at most $r' = \lfloor \frac{r}{m} \rfloor$ bits [31]. [19], [17], [31] approaches are based on this idea; however, these approaches only solve the static r-NSP problem.

Norouzi et al. [22] method can deal with dynamic r-NSP. They also partition the vectors into m sub-vectors. All sub-vectors in a partition are indexed using hash tables. Their method can handle arbitrary values of r. Query vector is divided into m sub-vectors as before. Each sub-vector is searched within the hash table for its partition using value $r = \lfloor \frac{r}{m} \rfloor$. The number of lookups performed for each hash table is computed on-the-fly using l, m, and r. Here l is the dimension of query vector, m is the number of partitions, and r is the search radius for the original query. Results obtained from m partitions are combined and checked for correctness to answer the original query. Finding a good value for m is central to the efficiency of multi-index hashing with non-overlapping subvectors. When the value of m is too large or too small the approach will not be effective. For [22] $m = \log_2(|D|)$ yields a near-optimal search cost. Here $|D|$ is the size of the vector. [22] is the current best scheme for dynamic r-NSP (and k-nearest-neighbours), and we show the our method outperforms [22] on dynamic r-NSP.

III. r-NSP USING TRIE AND HASH TABLE

In the interest of keeping our discussions on point, we assume that the reader is familiar with 1) binary vectors and 2) common operations, such as prefix, suffix, etc., on these vectors. We begin our discussions by formalizing null r-variations and the trie data structure.

Definition 2. *Given a set of binary vectors $D \subseteq 2^l$, a binary vector $q \in 2^l$ and a value $r \in \mathbb{Z}^+$, all vectors $x \in Q(q,r) \setminus N_r(D,q)$ are called null variations. l is the length of binary vector.*

A. Bitwise Trie

A trie is a tree data structure where all the descendants of a node have a common prefix stored at that node. The root node stores the empty string) [9]. So nodes at level i represent the set of all vectors that begin with the same prefix of length i, and branching at node i is determined by $i+1$ character of the string. Trie data structure storing binary vectors is commonly referred to *bitwise* trie (see Fig. 2 (left)). For bitwise trie, each node has two children, since $i+1$ character of a binary string can only take one of two possible values.

Definition 3. *Given $D \subset 2^l$. A* bitwise trie *of D denoted as T_D is defined as follow:*

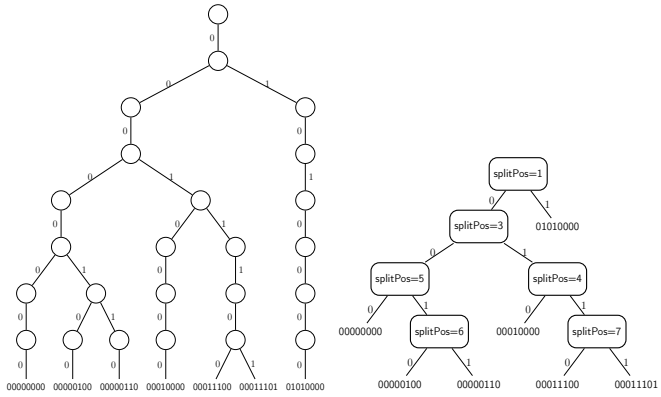

Figure 2. Bitwise trie (left) and compressed bitwise trie (right) for D={01010000, 00010000, 00011100, 00011101, 00000110, 00000100, 00000000}.

i) All nodes $n \in T_D$ represent a subset of vectors of D denoted as $V(n)$ that share a common prefix $p(V(n))$.

ii) All node $n \in T_D$ is divided in two subtree named $left(n)$ and $right(n)$ (see Fig. 2 (left)):

- *The left subtree represents a subset of the vectors associated with n that has a bit equal zero in the position after the common prefix. Formally $V(left(n)) = \{\mathbf{x} \in V(n) : \mathbf{x}[|p(V(n))|] = 0\}$.*
- *The right subtree represents a subset of the vectors associated with n that has a bit equal one in the position after the common prefix. Formally $V(right(n)) = \{\mathbf{x} \in V(n) : \mathbf{x}[|p(V(n))|] = 1\}$.*

iii) For all nodes n, the length of its prefix is one bit less than the length of the prefix of each subtree. Formally $|p(V(n))| = |p(V(left(n)))|+1 = |p(V(right(n)))|+1$.

Nodes without any children are called leaf *nodes. Nodes with at least one child are called* tree *nodes.*

The *root* node corresponds to the whole set D. Each tree node corresponds to the common prefix of its leaf nodes. Furthermore, the common prefix of a child is strictly longer than that of its parent. Thus, the length of the trie built from $D \subset 2^l$ must be at most (exactly) l.

B. r-NSP Queries Using Trie

Our method is motivated by the observation that, given a query vector, it is possible to identify its r-neighbours in the set of vectors stored in a bitwise trie. We can traverse the trie searching for r-variations as follows. At every node we are going to visit both children keeping track of the Hamming distance of the maximum common prefix of the visited node with the query vector. However, since one of the two children of an internal node can be empty, no non-null r-variation can be generated from that child. Any leaf that is reached during this traversal is an r-variation (of the query vector) that also exists in the set D. Note that this process effectively prunes

Algorithm 1 r-neighbours search using a trie

```
1:  procedure RANGEQUERY(T_D, q, r)
2:      if T_D = ∅ then
3:          return ∅
4:      return RQA(root(T_D), 0, q, r)
5:
6:  procedure RQA(node, i, q, r)
7:      if node = NIL or r < 0 then
8:          return ∅
9:      if isLeaf(node) then
10:         return {q}
11:     else
12:         if q[i] = 0 then
13:             vLeft = RQA(left(node), i + 1, q, r)
14:             vRight = RQA(right(node), i + 1, q, r − 1)
15:         else
16:             vRight = RQA(right(node), i + 1, q, r)
17:             vLeft = RQA(left(node), i + 1, q, r − 1)
18:         return vLeft ∪ vRight
```

all null variations. Algorithm 1 shows the pseudo-code for generating all non-null r-variations of $\mathbf{q}$ given T_D:

Lemma 1. *Algorithm 1 does not generate null variations.*

Proof: Since Algorithm 1 only returns vectors when a leaf node is reached (line 10), all r-variations returned by this algorithm are in D. ■

1) Compressed Bitwise Trie: It is possible to store set D in a trie more efficiently by using compressed bitwise trie that was first proposed by Coffman and Eve in [8]. Here internal nodes that only have one child are merged with their parents (Fig. 2 (right)).

Definition 4. *Given $D \subset 2^l$. A* compressed bitwise trie *of D denoted as CT_D is defined as follow:*

i All nodes $n \in CT_D$ represent a subset of vectors of D denoted as $V(n)$ that share a common prefix $p(V(n))$. The value $|p(V(n))|$ is called split position *($splitPos$).*

ii When the split position of a node is l, it is a leaf node, otherwise it is an internal node.

iii All internal nodes $N \in CT_D$ have two subtree named $left(N)$ and $right(N)$.

- *The left subtree represents some subset of the vectors associated with n that has a bit equal to zero in the position after the common prefix. Formally $V(left(n)) = \{\mathbf{x} \in V(n) : \mathbf{x}[|p(V(n))|] = 0\}$.*
- *The right subtree represents a subset of the vectors associated with n that has a bit equal to one in the position after the common prefix. Formally $V(right(n)) = \{\mathbf{x} \in V(n) : \mathbf{x}[|p(V(n))|] = 1\}$.*

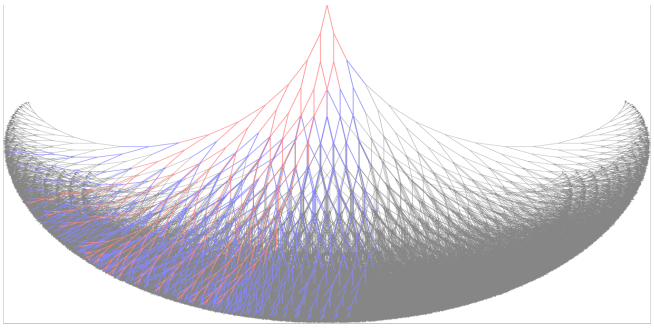

Figure 3. This image shows the nodes traversed when $r > 0$ (in red) and when $r = 0$ in blue on a trie of 1 million 64-bit vectors searched with $r = 2$. This suggests that a mechanism to avoid traversal when $r = 0$ will lead to significant performance gain by avoiding traversing all blue paths.

Algorithm 2 r-neighbours search using a compressed bitwise trie

```
1:  procedure RANGEQUERY(CT_D, q, r)
2:      if T_D = ∅ then
3:          return ∅
4:      return RQA(root(CT_D), q, r)
5:  procedure RQA(node, i, q, r)
6:      if node = NIL or r < 0 then
7:          return ∅
8:      if isLeaf(node) then
9:          return {q}
10:     r = r − H(q[: splitPos(node)], p(V(node)))
11:     q = p(V(node))‖q[splitPos(node) :]
12:
13:     if q[splitPos(node)] = 0 then
14:         vLeft = RQA(left(node), q, r)
15:         q[splitPos(node)] = 1
16:         vRight = RQA(right(node), q, r − 1)
17:     else
18:         vRight = RQA(right(node), q, r)
19:         q[splitPos(node)] = 0
20:         vLeft = RQA(left(node), q, r − 1)
21:     return vLeft ∪ vRight
```

C. r-NSP Using Hybrid Index

While bitwise trie offers superior pruning of null variations, we noticed that using trie alone for r-NSP does not achieve acceptable performance. This is due to the processing overhead of traversing a trie to generate non-null r-variations as compared to, say MIH [22], which generates *all* r-variations and checks them against the hash table. The good news is that it is possible to combine trie with hash table to improve null variations pruning, such that the r-NSP query performance of combined hybrid (trie + hash table) index is better than methods using only trie or hash table. We observe that it is possible to stop trie traversal early (i.e., before reaching a leaf node) by matching the current node

with the hash table. Fig. 3 highlights this observation. Paths shown in red refer to trie traversals when $r > 0$; where as, paths shown in blue refer to trie traversals after r has reached 0. Notice that there are far fewer red paths than there are blue paths. We can get large savings by avoiding going down the blue paths.

We have developed a novel strategy that uses a hash table to avoid subsequent traversals when $r = 0$ is reached. This results in a substantial reduction in the number of nodes explored during candidate r-variations generation. The proposed scheme stores the collection of binary vectors in both a trie and a hash table. Given a query vector, trie data structure is used to generate the candidate r-variations. This is accomplished by traversing the trie. During traversal, when r reaches 0, the current candidate is looked up in the hash table; we refer to it as the *membership check*. This determines in constant time if the current r-variation candidate is in the collection. No further traversals are needed below this node. Choosing hash table lookup at $r = 0$ is simply a design decision. It is indeed possible to extend the idea of using hash table for membership checks to $r > 0$; however, we would like to remind the reader the exponential growth of membership checks for values of r greater than 0 (see Eq. 1). Thus choosing $r > 0$ may result in little or no performance gain. We leave an in depth study of this phenomenon for a later date.

The hybrid index structure consists of a hash table HT and a compressed bitwise trie CT. All elements of D will be inserted in both structures. Algorithm 3 shows the pseudo-code to query the hybrid index. We direct readers attention to the fact that the only difference between Algorithm 2 and Algorithm 3 is that the latter terminates recursion early (lines 13 and 14), thereby avoids exploring the blue paths (Fig. 3).

D. Multi (Hybrid) Index

The hybrid index structure is proposed as a replacement of a pure hash table index. However, for high-dimensional vectors (i.e., large l) and large radius r, the current best approach is MIH proposed by Norouzi et al. [22]. We appropriate MIH method for our purposes by replacing hash table indexes with our hybrid indexes. Everything else stays the same.

The multi hybrid index approach that uses our proposed hybrid index structure is as follows. Given $D \in 2^l$ and $m \in [1, l)$:

1) vectors $\mathbf{v}$ of D are partitioned into m non-overlapping sub-vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_m$ of size $\lfloor \frac{l}{m} \rfloor$ or $\lceil \frac{l}{m} \rceil$ such as $\sum_{i=1}^{m} |\mathbf{v}_i| = l$.
2) m hybrid indexes, denoted as $HI_1, HI_2, \ldots HI_m$, are created. Each hybrid index will store sub-vectors from the corresponding partition, i.e., all $\mathbf{v}_1$'s are indexed in HI_1, all $\mathbf{v}_2$'s are indexed in HI_2, and so on.

Algorithm 3 r-neighbours search using hybrid index

```
1:  procedure RANGEQUERY(HT_D, CT_D, q, r)
2:      if T_D = ∅ then
3:          return ∅
4:      v = RQA(root(CT_D), q, r)
5:      Result = ∅
6:      for all v ∈ v do
7:          if v ∈ HT_D then
8:              Result = Result ∪ {v}
9:      return Result
10: procedure RQA(node, i, q, r)
11:     if node = NIL or r < 0 then
12:         return ∅
13:     if isLeaf(node) or r = 0 then
14:         return {q}
15:     vAtNode = ∅
16:     if H(q[: splitPos(node)], p(V(node))) > 0 then
17:         r = r − H(q[: splitPos(node)], p(V(node)))
18:         q = p(V(node))‖q[splitPos(node) :])
19:     if q[splitPos(node)] = 0 then
20:         vLeft = RQA(left(node), q, r)
21:         q[splitPos(node)] = 1
22:         vRight = RQA(right(node), q, r − 1)
23:     else
24:         vRight = RQA(right(node), q, r)
25:         q[splitPos(node)] = 0
26:         vLeft = RQA(left(node), q, r − 1)
27:     return vLeft ∪ vRight
```

During search, the query vector $\mathbf{q}$ is also partitioned into m sub-vectors as before, i.e., $\mathbf{q}$ will be divided into $\mathbf{q}_1, \mathbf{q}_2, \ldots \mathbf{q}_m$ sub-vectors. Next, each query sub-vector is matched in the corresponding hybrid index using radius r', where $r' = \lfloor \frac{r}{m} \rfloor$. In other words, $\mathbf{q}_i$ is searched in HI_i using r'. The results obtained from each index are then combined to see if the corresponding vector in D is at most at Hamming distance r of the the query $\mathbf{q}$. We refer to reader to [22] for more details about how this is accomplished.

IV. Experimental Evaluation

As stated previously, MIH is the current best scheme for r-NSP [22]. Therefore, we compare our method with MIH. MIH implementation is available to the research community, and we have used their implementation to generate performance results for their approach. We also added our compressed trie into their system to implement our MI-Trie and MI-Hybrid methods. MI-Trie and MI-Hybrid replace hash table index used in MIH scheme with our MI-Trie and hybrid (compressed trie + hash table) index, respectively. We will compare four methods: 1) linear scan; 2) MIH [22]; 3) MI-Trie; and 4) MI-Hybrid. Similar to MIH, MI-Trie and MI-Hybrid methods partition the vectors into m sub-vectors.

Experiments are performed on a workstation with a 2.9 GHz quad-core Intel Xeon processor, 20 MB of L2 cache, and 64 GB of RAM. It is worth noting that large L2 cache significantly improves the performance of linear scan [22]. For our experiments, we only used a single core to simplify runtimes measurements. The runtimes reported in this work are the result of five runs of the algorithm in exactly the same conditions. The datasets used for evaluation are uniformly-distributed randomly-generated vectors.

We realize that MIH technique developed in [22] provides a mechanism for selecting an "optimal" value for m given the dataset. Our method also benefits from this value of m. The results shown here suggest that our method MI-Hybrid outperforms MIH even for optimal values of m. The results also suggest that our method performs slightly worse than MIH when trie is full. We discuss below that trie is almost never full when dealing with high dimensional vectors, suggesting that our method will outperform MIH when dealing with such vectors.

A. Hash Table Lookups

Since the goal of our hybrid approach is to improve runtime by reducing the number of hash table lookups that need to be performed, we compare both methods on the number of hash table lookups performed. MIH uses the naïve approach to compute all r-variations, and the total number of lookups that it will perform can be computed using Eq. 1 for every block (parameter m). MI-Hybrid, however, skips null r-variations, reducing the number of lookups to perform.

Fig. 4 (left) plots the number of hash table lookups for MIH and MI-Hybrid methods for a dataset of one million 128-bits vectors for $m = 4$. Each index, therefore, stores 32-bit sub-vectors. Notice that hash table lookups for our method (MI-Hybrid) is significantly lower than those for MIH. Furthermore, the difference in the number of lookups increases sharply as r increases. Clearly, MI-Hybrid method is effectively pruning null null r-variations.

Fig. 4 (middle) repeats the experiment with 6 indexes. $m = 6$ is the theoretical "best" value for this scenario as determined by MIH [22]. Again, observe the hash table lookups savings obtained by our method (MI-Hybrid) over MIH. This time, however, the savings obtained are not as good as those obtained in the previous case ($m = 4$). For $m = 6$, each index is storing 21-bit sub-vectors. The total number of 21-bit vectors is 2^{21}, which is $\approx 10^6$. We are storing 1 million vectors, so trie corresponding to each index is nearly full. When trie is full, it looses its ability to prune null r-variations. Still even in this case, our method (MI-Hybrid) performs far fewer lookups than MIH.

Fig. 4 (right) is designed to showcase what happens when trie is "full." Here we index 1 million 64-bit vectors using 4 indexes, i.e., $m = 4$. Each index here stores 16-bit vectors. The total number of 64-bit vectors is approximately 64000 ($\ll$ 1 million). So the trie is full. In this case the number of lookups for both methods—ours and MIH—is exactly the same. The trie offers no advantage. This suggests that it is important to pick m carefully. Large values for m will result in a dense trie. The performance for MIH also depends upon m. It turns out that performance of both MI-Hybrid and MIH decreases if m is too small or too large.

B. Runtime Comparison

The above results compare the number of hash table lookups for our method MI-Hybrid and MIH. We observe that when trie is not full, our method generates far fewer hash table lookups. Here, we explore if this reduction in lookups results in improved runtimes.

Fig. 5 (left) shows a comparison of the runtimes on 1000 queries for the linear scan method (that serves as baseline), the MIH scheme [22], and our methods MI-Trie and MI-Hybrid. The linear scan method does not depend on the radius r, while MIH, MI-Trie, and MI-Hybrid methods depend upon r. Specifically, the processing times for all three increase sharply for increasing r. This behaviour is to be expected. As can be seen, MI-Hybrid outperforms other methods (Linear scan, MIH, and MI-Trie) by a significant margin for large values of r. Note also that MIH performs worse than linear scan for $r > 20$; granted that $m = 4$ used here is not the optimal value of m for MIH.

Fig. 5 (middle) shows a comparison of the runtimes on 1000 queries for MIH, MI-Trie, and MI-Trie+MIH methods. Here we use $m = 6$, which is the recommended value of m as determined by MIH. MIH in this case performs better than linear scan (not shown here, but indicated by the red line in the left-hand plot) and MI-Trie. MI-Trie+MIH, however, again outperforms MIH by a significant margin for large values of r. For $r = 20$, MI-Hybrid is twice as fast as MIH.

Fig. 5 (right) shows the case when trie is full. In this case 64-bit vectors are partitioned into 4 16-bit sub-vectors. Here MIH outperforms MI-Trie and MI-Trie+MIH methods. This is to be expected. We designed this test to get a full trie. We know that our method performs worse than MIH (the current best method) when trie is full. The good news is that this is a somewhat contrived example, since trie is almost always highly sparse for high dimensional vectors (see discussion below).

C. How Likely is Sparse Trie

These results demonstrate two things: 1) the number of lookups for our method MI-Hybrid when trie is sparse is significantly less than MIH and 2) the far fewer number of lookups is closely tied to the overall runtime performance. Specifically, when trie is sparse our method MI-Hybrid significantly outperforms MIH, which is the current best method. This confirms our hypothesis that it is possible to decrease runtimes by decreasing the number of lookups. The question then is, how likely is it to get a sparse trie for real world dataset. Since the savings in the number of lookups

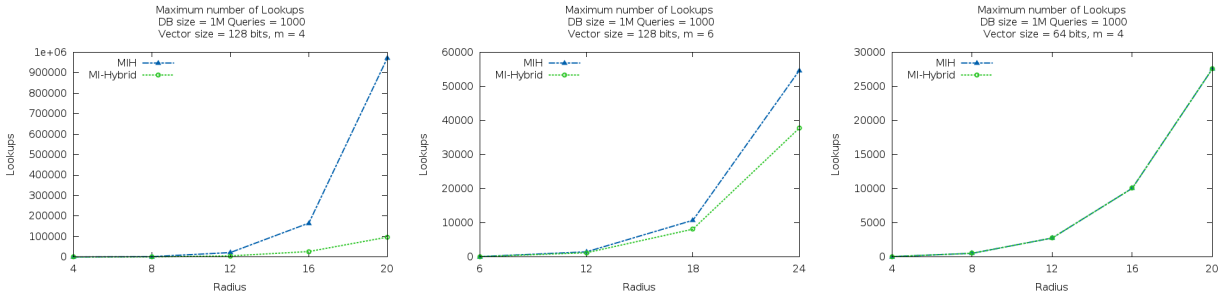

Figure 4. Number of hash table lookups for MIH and our hybrid approach (MI-Hybrid) for a database of 1 million 128-bit vectors (left and middle) and 64-bit vectors (right). The number of lookups is an average over 1000 queries. (Left) $m = 4$ and each index stores 32-bit vectors. (Middle) $m = 6$ and each index stores 21-bit vectors. (Right) $m = 4$ and each index stores 64-bit vectors.

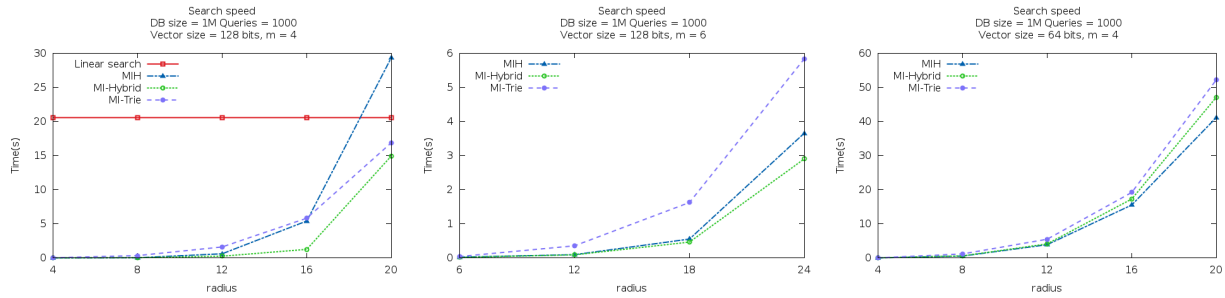

Figure 5. Runtimes comparison of Linear scan, MIH, MI-Trie, and our hybrid approach (MI-Hybrid) for a database of 1 million 128-bit (left and middle) or 64-bit (right) vectors using 1000 queries. (Left) $m = 4$ and every index manages 32-bits. (Middle) $m = 6$ and every index manages 21-bit vectors. $m = 6$ is the recommended value of m for MIH. (Right) $m = 4$ and this plot shows the performance when trie is full. Linear scan runtimes shown on the left (red) apply to all three plots; linear scan runtimes depend only upon the size of the database.

are tied to the degree of sparseness of the trie. Generally speaking, a trie that stores high-dimensional vectors is much more likely to be sparse. Lets do a thought experiment. Say, we need to store 512-bit vectors using a multi-index method. For 512-bit vectors, according to MIH, the optimal value for m is 9. This suggests that we should use 9 indexes, each of which will be storing 56-bit sub-vectors constructed from the original 512-bit sub-vectors. If using trie, we would need more than 10^{16} vectors to get a full trie. We conclude that for high dimensional vectors, getting a full trie is extremely unlikely. Consequently, our method (MI-Hybrid) will outperform MIH for high-dimensional vectors.

V. Conclusion

We propose a novel solution to dynamic r-NSP. We have developed a hybrid data structure for indexing binary vectors, plus the associated query processing machinery. The data structure combines a compressed bitwise trie and a hash table to index the dataset and the query algorithm seamlessly uses both—hash table and trie—during r-neighbours searches. The proposed data structure exhibits both *lookup efficiency* and *local searchibility*, and it achieves better performance than if using hash table or trie alone. Our method is able to efficiently prune null r-variations of the query vector, which results in far fewer hash table lookups, translating in better performance and reduced runtimes.

MIH scheme that appeared in [22] is the current best approach for dynamic r-NSP, and its close variant the k-nearest-neighbours search problem. We observe that MIH cannot effectively deal with high-dimensional vectors due to exponential growth in r-variations for large l and r, where l is the length of the vector and r is the radius used for r-NSP. This is true even though MIH partitions incoming l-bit vectors into $\log_2 l$ sub-vectors. MIH uses hash table as index, and it is unable to exploit *local searchibility* to prune null r-variations. We too partition the incoming query vector into sub-vectors; however, we store these sub-vectors into our hybrid (hash table + compressed bitwise trie) index. Our hybrid index outperforms hash table when trie is sparse. Furthermore, the difference in performance of our hybrid index and the hash table increases dramatically with increasing l, since the degree of sparseness of trie increases with l.

We devised 6 tests to study the performance of our method and show that for sparse trie, our method can result in substantial performance increase for large r. We also show that our method achieves similar performance to MIH when trie is full, albeit on a somewhat contrived example. This is so because trie is almost never full. Actually, for high-dimensional vectors, a trie is much more likely to be highly

sparse. Consequently, we safely conclude that our approach (MI-Hybrid) achieves state-of-the-art performance for real world datasets.

It is obvious that high-dimensional vectors result in a sparse trie. However, it is possible that the distribution of the data also effects the performance of our method. We suspect that the effect of data distribution on the performance of ours (MI-Hybrid), and on MIH, will be minimal. We plan to investigate it in detail in the future.

This work addresses dynamic r-NSP; where as, MIH also works for k-nearest-neighbours problem. We can as easily adapt our index to solve for k-nearest-neighbours problem through successive expansion on r. This is, however, a trivial solution. We want to explore if *local searcibility* provided by our hybrid index can be exploited to answer k-nearest-neighbours queries more efficiently. Currently all index structures exist in main memory, thus the dataset is limited by the size of the physical memory available to the algorithm. We would like to investigate the issues and solutions of designing disk based index structures. We foresee that there are some interesting issues associated with merging our Hybrid index with well-known disk based hashing using B+ tree.

REFERENCES

[1] A. Alahi, R. Ortiz, and P. Vandergheynst. Freak: Fast retina keypoint. In *Proc. of CVPR*, pages 510–517, Providence, June 2012. 1

[2] A. N. Arslan. Efficient approximate dictionary look-up over small alphabets. Technical report, University of Vermont, 2005. 2

[3] A. N. Arslan and O. Egecioglu. Dictionary look-up within small edit distance. In *Proc. COCOON*, pages 127–136, Singapore, August 2002. 2

[4] A. Bergamo, L. Torresani, and A. W. Fitzgibbon. Picodes: Learning a compact code for novel-category recognition. In *Proc. NIPS*, pages 2088–2096, December 2011. 1

[5] S. Brin. Near neighbor search in large metric spaces. In *Proc. VLDB*, Berlin, September 1995. 2

[6] G. Brodal and L. Gasieniec. Approximate dictionary queries. In *Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 65–74. Springer Berlin Heidelberg, 1996. 2

[7] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. Brief: Binary robust independent elementary features. In K. Daniilidis, P. Maragos, and N. Paragios, editors, *European Conference on Computer Vision (ECCV)*, volume 6314 of *Lecture Notes in Computer Science*, pages 778–792. Springer Berlin Heidelberg, September 2010. 1

[8] E. G. Coffman, Jr. and J. Eve. File structures using hashing functions. *Communications of the ACM*, 13(7):427–432, July 1970. 4

[9] R. De La Briandais. File searching using variable length keys. In *Proc. IRE-AIEE-ACM Western Joint Computer Conference*, pages 295–298, San Francisco, March 1959. 3

[10] M. Esmaeili, M. Fatourechi, and R. Ward. A robust and fast video copy detection system using content-based fingerprinting. *IEEE Trans. on Information Forensics and Security*, 6(1):213–226, 2011. 1

[11] M. Esmaeili, R. Ward, and M. Fatourechi. A fast approximate nearest neighbor search algorithm in the hamming space. *TPAMI*, 34(12):2481–2488, 2012. 2

[12] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proc. SIGMOD*, pages 541–552, Scottsdale, May 2012. 2

[13] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. STOC*, pages 604–613, Dallas, May 1998. 2

[14] D. Kuettel, M. Guillaumin, and V. Ferrari. Segmentation propagation in imagenet. In *European Conference of Computer Vision 2012 (EECV)*, volume 7578 of *Lecture Notes in Computer Science*, pages 459–473. Springer Berlin Heidelberg, October 2012. 1

[15] J. Landré and F. Truchetet. Fast image retrieval using hierarchical binary signatures. In *Proc. ISSPA*, pages 1–4, Sharjah, February 2007. 1

[16] J. Landré and F. Truchetet. Image retrieval with binary hamming distance. In *Proc. VISAPP*, Barcelona, March 2007. 1

[17] A. Liu, K. Shen, and E. Torng. Large scale hamming distance query processing. In *Proc. ICDE*, pages 553–564, Hannover, April 2011. 1, 2, 3

[18] M. G. MaaB and J. Nowak. Text indexing with errors. Technical report, Institut für Informatik, Technische Universit ät München, 2005. 2

[19] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proc. WWW*, pages 141–150, Banff, May 2007. 3

[20] M. Miller, M. Rodriguez, and I. J. Cox. Audio fingerprinting: nearest neighbor search in high dimensional binary spaces. In *IEEE Workshop on Multimedia Signal Processing (MMSP)*, pages 182–185, St. Thomas, December 2002. 1

[21] M. Muja and D. G. Lowe. Fast matching of binary features. In *Proc. CRV*, pages 404–410, Toronto, May 2012. 2

[22] M. Norouzi, A. Punjani, and D. Fleet. Fast search in hamming space with multi-index hashing. *TPAMI*, (6), 2014. 1, 2, 3, 4, 5, 6, 7

[23] J. Oostveen, T. Kalker, and J. Haitsma. Feature extraction and a database strategy for video fingerprinting. In *Proc. VISUAL*, pages 117–128, Hsin Chu, March 2002. 1

[24] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *Proc. ICCV*, pages 2564–2571, Barcelona, November 2011. 1

[25] G. Shakhnarovich. *Learning Task-Specific Similarity*. PhD thesis, MIT, 2006. 2

[26] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *Proc. CVPR*, pages 1–8, Anchorage, June 2008. 1

[27] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Proc. NIPS*, pages 1753–1760, Vancouver, December 2008. 1

[28] Q. Xiao, M. Suzuki, and K. Kita. Fast hamming space search for audio fingerprinting systems. In *Proc. ISMIR*, pages 133–138, Miami, October 2011. 1

[29] H. Y. and Y. W. A lbp-based face recognition method with hamming distance constraint. In *Proc. ICIG*, pages 645–649, Chengdu, August 2007. 1

[30] A. C. Yao and F. F. Yao. Dictionary look-up with one error. *Journal of Algorithms*, 25(1):194 – 202, 1997. 2

[31] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu. Hmsearch: An efficient hamming distance query processing algorithm. In *Proc. SSDBM*, pages 19:1–19:12, Baltimore, July 2013. 3