# Fast Search in Hamming Space with Multi-Index Hashing

Mohammad Norouzi  Ali Punjani  David J. Fleet
Department of Computer Science
University of Toronto
{norouzi,alipunjani,fleet}@cs.toronto.edu

## Abstract

*There has been growing interest in mapping image data onto compact binary codes for fast near neighbor search in vision applications. Although binary codes are motivated by their use as direct indices (addresses) into a hash table, codes longer than 32 bits are not being used in this way, as it was thought to be ineffective. We introduce a rigorous way to build multiple hash tables on binary code substrings that enables exact $K$-nearest neighbor search in Hamming space. The algorithm is straightforward to implement, storage efficient, and it has sub-linear run-time behavior for uniformly distributed codes. Empirical results show dramatic speed-ups over a linear scan baseline and for datasets with up to one billion items, 64- or 128-bit codes, and search radii up to 25 bits.*

## 1. Introduction

There has been growing interest in mapping image data onto compact binary codes for fast near neighbor search in vision applications (*e.g.,* [20, 21, 23]). Binary codes are storage efficient and comparisons require just a small number of machine instructions; millions of binary codes can be compared to a query in less than a second. But the most compelling reason for binary codes is their use as direct indices (addresses) into a hash table, yielding a dramatic increase in search speed compared to an exhaustive linear scan (*e.g.,* [24, 19, 16]).

The problem is that, in practice, using binary codes as hash indices is not necessarily efficient. To find near neighbors one needs to examine all hash table entries (or *buckets*) within some Hamming ball around the query. And the number of such buckets grows near-exponentially with the search radius (Fig. 2a). Even with a small search radius, the number of buckets to examine may be larger than the number of items in the database, hence slower than linear scan. Recent papers on binary codes mention the use of hash tables, but resort to linear scan when codes are longer than 32 bits (*e.g.,* [23, 19, 12, 16]), although longer codes are often necessary to preserve sufficient similarity (*e.g.,* see Fig. 5).

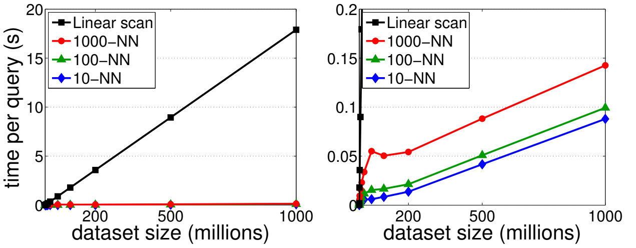
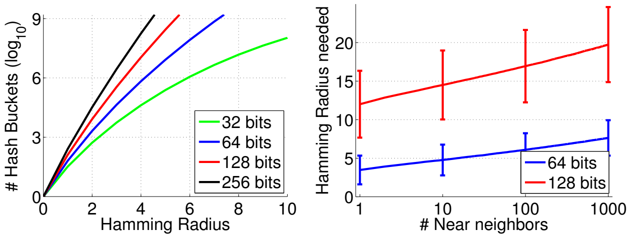


Figure 1. Nearest-neighbor search on a database of 64-bit binary codes learned from SIFT descriptors. Run-times per query are shown for the proposed multi-index hashing algorithm, searching for 10, 100, and 1000 nearest neighbors. They are compared to a linear scan baseline. Left and right plots show different vertical scales; *i.e.,* the vertical axis of the right plot is 100 times smaller than the left, showing query times between 0 and 0.2s.

This paper presents a new algorithm for exact $K$-nearest neighbor search on binary codes that is dramatically faster than linear scan. This has been an open problem since the introduction of hashing techniques with binary codes. Our new multi-index hashing algorithm exhibits sub-linear search times, is storage efficient, and straightforward to implement. As an example, Fig. 1 plots CPU run-times per query as a function of the size of a database comprising 64-bit codes learned from SIFT descriptors with Minimal Loss Hashing [16]. Our current implementation searches a dataset of a billion codes hundreds of times faster than linear scan, on a single computer.

### 1.1. Background: Problem and Related Work

Nearest neighbor (NN) search on binary codes has been used for image search [18, 23, 24], matching local features [9, 21], and parameter estimation [20]. Such techniques begin with a similarity-preserving mapping from high-dimensional data to binary codes. For many problems one wants to preserve Euclidean distance (*e.g.,* [5, 12, 18, 21, 24]), while others focus on semantic similarity (*e.g.,* [16, 20, 19, 23]). Our algorithm does not depend on the specific method for generating the binary codes. Rather, we are primarily concerned with fast search in Hamming



(a) (b)

Figure 2. (a) Curves show the ($\log_{10}$) number of distinct hash table indices (buckets) within a Hamming ball of radius $r$, for different code lengths. With 64-bit codes there are about 1B buckets within a Hamming ball of radius 7 bits. Hence with fewer than 1B database items, and a search radius of 7 or more, a hash table would be less efficient than linear scan. Using hash tables with 128-bit codes is prohibitive for radii larger than 6. (b) This plot shows the expected search radius required for $K$-NN search as a function of $K$, based on a dataset of 1B SIFT descriptors. Binary codes with 64 and 128 bits were obtained by random projections (LSH) from the SIFT descriptors [11]. Standard deviation bars help show the large search radii required in many cases.

space. While there exist several other promising approaches to fast approximate NN search on large real-valued image features (*e.g.,* [1, 10, 15]), we restrict our attention in this paper to the use of compact binary codes.

In particular, we address two related search problems. Given a dataset of binary codes, $\mathcal{D}$, the first is to find the $K$ codes in $\mathcal{D}$ that are closest in Hamming distance to a given query, *i.e.,* $K$-NN search in Hamming space. The 1-NN problem in Hamming space was called the *Best Match* problem by Minsky and Papert [14]. They observed that there are no obvious approaches significantly better than exhaustive search.

The second problem is to find all codes in a dataset $\mathcal{D}$ that are within a fixed Hamming distance of a query, sometimes called the *Approximate Query* problem [6], or *Point Location in Equal Balls* (PLEB) [8]. A binary code is an $r$*-neighbor* of a query code $\mathbf{q}$ if it differs from $\mathbf{q}$ in $r$ bits or less. One way to find all $r$-neighbors of $\mathbf{q}$ is to use a hash table populated with the binary codes, and examine all hash *buckets* whose indices are within $r$ bits of $\mathbf{q}$ (*e.g.,* [23]). For binary codes of $b$ bits, the number of distinct hash buckets to examine is

$$L(b,r) \;=\; \sum_{k=0}^{r} \binom{b}{k} .$$

As shown in Fig. 2a, $L(b,r)$ grows rapidly with $r$. Thus, this approach is only practical for small radii or short code lengths. Some vision applications restrict search to exact matches (*i.e.,* $r=0$) or a small search radius (*e.g.,* [7]), but in most cases of interest the desired search radius is larger than is currently feasible (*e.g.,* see Fig. 2b).

Our work is inspired in part by the multi-index hashing results of Greene, Parnas, and Yao [6]. Building on the classical Turan problem for hypergraphs, they show that with a suitable choice of a set of over-lapping binary substrings, any two codes that differ by at most $r$ bits are guaranteed to be identical in at least one of the substrings. Accordingly, they propose an exact method for finding all $r$-neighbors of a query using multiple hash tables, one for each substring. At query time, candidate $r$-neighbors are found by using query substrings as indices into their corresponding hash tables. As explained below, the main drawback of their approach is the prohibitive storage required for the requisite number of hash tables. By comparison, the method we propose requires much less storage, and is only marginally slower in search performance.

While we focus on exact search, there also exist algorithms for finding *approximate* $r$-neighbors of queries ($\epsilon$-PLEB), or approximate nearest neighbors ($\epsilon$-NN) in Hamming space. One example is Locality Sensitive Hashing (LSH) for binary codes [8, 2]. LSH also uses multiple substrings as hash indices to formulate a bound on the probability of missing a $(1+\epsilon)r$-neighbor. Such approximate methods are interesting, and indeed, our approach below could be made faster by allowing misses. Nevertheless, this paper is concerned primarily with the *exact* search problem.

This paper proposes a data-structure that applies to both $K$-NN and to finding $r$-neighbors in Hamming space. We prove that for uniformly distributed binary codes of $b$ bits, and a search radius of $r$ bits when $r/b$ is small, our query time is sub-linear in the size of dataset. We also apply the algorithm to real-world datasets, with performance far superior to linear scan. To our knowledge this is the first practical data-structure solving exact $K$-NN in Hamming space.

## 2. Multi-Index Hashing

Our approach is a form of multi-index hashing. Binary codes from the database are indexed $m$ times into $m$ different hash tables, based on $m$ disjoint binary substrings. Given a query code, entries that fall *close* to the query in at least one such substring are considered *neighbor candidates*. Candidates are then checked for validity using the entire binary code, to remove any non-$r$-neighbors. To be practical for large-scale datasets, the substrings must be chosen so that the set of candidates is small, and storage requirements are reasonable. We also require that all true neighbors will be found.

In more detail, each binary code $\mathbf{h}$, comprising $b$ bits, is partitioned into $m$ disjoint substrings, $\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(m)}$, each of length $\lfloor b/m \rfloor$ or $\lceil b/m \rceil$ bits. For convenience in what follows, we assume that $b$ is divisible by $m$, and that the substrings comprise contiguous bits. The key idea rests on the following proposition: When two binary codes $\mathbf{h}$ and $\mathbf{g}$ differ by $r$ bits or less, then, in at least one of their $m$ substrings they must differ by at most $\lfloor r/m \rfloor$ bits. Formally, when $\|\mathbf{h}-\mathbf{g}\|_H \le r$, where $\|.\|_H$ denotes Hamming norm, then there exists a substring $k$, $1 \leq k \leq m$, such that

$$\|\mathbf{h}^{(k)} - \mathbf{g}^{(k)}\|_H \;\leq\; \left\lfloor \frac{r}{m} \right\rfloor . \qquad (1)$$

Proof of (1) follows straightforwardly from the Pigeonhole Principle. That is, if Hamming distance between each of the $m$ substrings is strictly greater than $\lfloor r/m \rfloor$, then $\|\mathbf{h} - \mathbf{g}\|_H$ must be larger than $r$, which is a contradiction.

The significance of (1) arises from the fact that the substrings have only $b/m$ bits, and that the required search radius in each substring reduces to $\lfloor r/m \rfloor$. For example, if $\mathbf{h}$ and $\mathbf{g}$ differ by 3 bits or less, and $m = 4$, at least one of the 4 substrings must be identical. If they differ by at most 7 bits, then in at least one substring they differ by no more than 1 bit; *i.e.,* we can search a Hamming radius of 7 bits by searching a radius of 1 bit on each of 4 substrings. More generally, instead of examining $L(b, r)$ hash buckets, it suffices to examine $L(b/m, \lfloor r/m \rfloor)$ buckets in each of $m$ substring hash tables.

Given a dataset, one hash table is built for each of the $m$ substrings of the codes. For a query $\mathbf{q}$ with substrings $\{\mathbf{q}^{(i)}\}_{i=1}^m$, we search the $i^{th}$ substring hash table for entries that are within a Hamming distance $\lfloor r/m \rfloor$ of $\mathbf{q}^{(i)}$, thereby retrieving a set of candidates, denoted $\mathcal{N}_i(\mathbf{q})$. According to the above proposition, the union of the $m$ sets, $\mathcal{N}(\mathbf{q}) = \bigcup_i \mathcal{N}_i(\mathbf{q})$, is necessarily a superset of the $r$-neighbors of $\mathbf{q}$. The last step of the algorithm computes the Hamming distance between $\mathbf{q}$ and each candidate in $\mathcal{N}(\mathbf{q})$, retaining only those codes that are true $r$-neighbors of $\mathbf{q}$.

The key idea here stems from the fact that, with $n$ binary codes of $b$ bits, the vast majority of the $2^b$ possible buckets in a full hash table will be empty, since $2^b \gg n$. It seems expensive to examine all $L(b, r)$ buckets within $r$ bits of a query, since most of them contain no items. Instead, we merge many buckets together (most of which are empty) by marginalizing over different dimensions of the Hamming space. The downside is that these larger buckets are not restricted to the Hamming volume of interest around the query. Hence not all items in the merged buckets are $r$-neighbors of the query, so we need to cull any candidate that is not a true $r$-neighbor.

The search cost depends on the number of *lookups* (*i.e.,* the number of buckets examined), and the number of candidates tested. Not surprisingly there is a natural trade-off between them. With a large number of lookups one can minimize the number of extraneous candidates. By merging many buckets to reduce the number of lookups, one obtains a large number of candidates to test. In the extreme case with $m = b$, substrings are 1 bit long, so we can expect the candidate set to include the entire database.

Note that the idea of building multiple hash tables is not novel in itself (*e.g.,* see [6, 8]). However previous work relied heavily on exact matches in substrings, which we relax in this paper. This leads to a much more practical method.

# 3. Performance Analysis

Next, we develop an analytical model of search performance to answer two questions: (1) How does search cost depend on substring length and hence the number of substrings? (2) How do run-time and storage complexity depend on database size, code length, and search radius? To help answer these questions we exploit a well-known bound on the sum of binomial coefficients; *i.e.,* for any $0 < \epsilon \leq \frac{1}{2}$ and $1 \leq \eta$,

$$\sum_{\kappa=0}^{\lfloor \epsilon\, \eta \rfloor} \binom{\eta}{\kappa} \;\leq\; 2^{H(\epsilon)\, \eta} , \qquad (2)$$

where $H(\epsilon) \equiv -\epsilon \log_2 \epsilon - (1-\epsilon) \log_2 (1-\epsilon)$ is the entropy of a Bernoulli distribution with probability $\epsilon$ [4].

As above, let $n$ denote the number of $b$-bit codes in the database, and let $r$ be the Hamming search radius. Let $m$ denote the number of hash tables (one for each substring), and let the substring length be $s = b/m$. Hence, the substring search radius becomes $\lfloor r/m \rfloor = \lfloor s\, r/b \rfloor$.

The number of lookups in our multi-index hashing algorithm is the product of $m$, the number of substrings, and the number of hash table buckets within a radius of $\lfloor s\, r/b \rfloor$ on substrings of length $s$ bits. Accordingly, using (2), if the search radius is less than half the code length, $r \leq b/2$, then the total number of lookups is given by

$$\mathit{lookups}(s) \;=\; m \sum_{k=0}^{\lfloor s\, r/b \rfloor} \binom{s}{k} \;\leq\; \frac{b}{s}\, 2^{H(r/b) s} . \qquad (3)$$

Clearly, as we decrease the substring length $s$, thereby increasing the number of substrings $m$, exponentially fewer lookups are needed.

To analyze the expected number of candidates per bucket, we consider the case in which the $n$ binary codes are uniformly distributed over the Hamming space. In this case, for a substring of $s$ bits, the substring hash table has $2^s$ buckets, and hence the expected number of items per bucket in a substring hash table is $n/2^s$. The size of the candidate set therefore equals the number of lookups times $n/2^s$.

The total search cost per query is the cost for lookups plus the cost for candidate tests. While these costs will vary with the code length $b$ and the way the hash tables are implemented, we find that, to a reasonable approximation, the costs of a lookup and a candidate test are similar. Accordingly, we model the total search cost per query, for retrieving all $r$-neighbors, in units of the time required for a single lookup, as

$$\mathit{cost}(s) \;=\; \left(1 + \frac{n}{2^s}\right) \frac{b}{s} \sum_{k=0}^{\lfloor sr/b \rfloor} \binom{s}{k} , \qquad (4)$$

$$\;\leq\; \left(1 + \frac{n}{2^s}\right) \frac{b}{s}\, 2^{H(r/b)s} . \qquad (5)$$

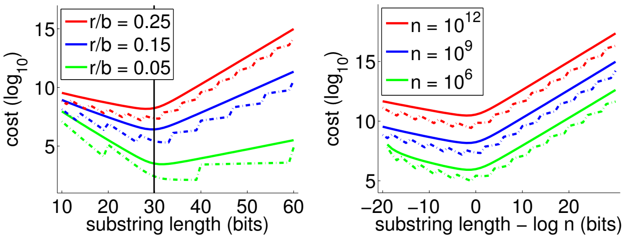


Figure 3. The cost (4) and its upper bound (5) are shown as functions of substring length. The code length in all cases is $b = 128$ bits. The left panel shows different search radii, all for a database with $n = 10^9$ codes. The right panel shows three database sizes, all for a search radius $r = 0.25\,b$. Here, each curve has been displaced horizontally by $-\log_2 n$. The minima are aligned.

### 3.1. Optimal Substring Length

In order to optimize $cost(s)$ with respect to $s$, note that dividing (4) by $b$ has no effect on the optimal $s$, denoted $s^*$. So $s^*$ can be viewed as a function of $n$ and the ratio $r/b$.

Figure 3 plots cost as a function of substring length $s$, for 128-bit codes, different database sizes $n$, and different search radii (expressed as a fraction of the code length $b$). Dashed curves depict $cost(s)$ in (4) while solid curves of the same color depict the upper bound in (5). The tightness of the bound is evident in the plots, as are the quantization effects of the upper range of the sum in (4).

Fig. 3 (left) shows cost for search radii equal to $5\%$, $15\%$, and $25\%$ of the code length, with $n = 10^9$ for all curves. One striking property of these curves is that the cost is persistently minimal in the vicinity of $s = \log_2 n$, indicated by the vertical line close to 30 bits. This behavior remains consistent over a wide range of database sizes.

Fig. 3 (right) shows the dependence of cost on $s$ for databases with $n = 10^6$, $10^9$, and $10^{12}$, all with $r/b = 0.25$ and $b = 128$ bits. In this case we have laterally displaced each curve by $-\log_2 n$; notice how this aligns the minima close to $0$. These curves suggest that, over a wide range of conditions, cost is minimal for $s = \log_2 n$. Interestingly, for this choice of the substring length, the expected number of items per substring bucket, *i.e.,* $n/2^s$, reduces to 1. As a consequence, the number of lookups is expected to be equal to the number of candidates.

A somewhat involved theoretical analysis, based on Stirling's approximation, also suggests that deviating from $s = \log_2 n$ will increase the order of retrieval time. We omit this analysis here due to lack of space.

### 3.2. Run-Time Complexity

Choosing $s$ in the vicinity of $\log_2 n$ provides a characterization of retrieval run-time complexity. When $s = \log_2 n$, the upper bound on the number of lookups (3) also becomes a bound on the number candidates. In particular, if we substitute $\log_2 n$ for $s$ in (5), then we find the following upper bound on the cost, now as a function of the database size, the code length, and the search radius:

$$query\ cost \ \leq\ 2\,\frac{b}{\log_2 n}\ n^{H(r/b)}\,. \tag{6}$$

Thus, for a uniform distribution over binary codes, if we choose $m$ such that $s \approx \log_2 n$, the expected query time complexity is $O(b\,n^{H(r/b)}/\log_2 n)$. For a small ratio of $r/b$ this is sub-linear in $n$. For example, if $r/b \leq .11$, then $H(.11) < .5$, and the run-time complexity becomes $O(b\sqrt{n}/\log_2 n)$. That is, the search time increases with the square root of the database size. For $r/b \leq .06$, this becomes $O(b\sqrt[3]{n}/\log_2 n)$. The time complexity with respect to $b$ is not as important as that with respect to $n$ since $b$ is not expected to vary significantly in most applications.

### 3.3. Storage Complexity

The storage complexity of our multi-index hashing algorithm is also appealing. To store the full database of binary codes requires $O(nb)$ bits. For each of $m$ hash tables, we also need to store $n$ unique identifiers to the database items. This allows one to identify the retrieved items and fetch their full codes; this requires an additional $O(mn\log_2 n)$ bits. In sum, the storage required is $O(nb + mn\log_2 n)$. When $m \approx b/\log_2 n$, as is suggested above, this storage cost reduces to $O(nb + n\log_2 n)$. (Here, the $n\log_2 n$ term always appears because $m \geq 1$.)

While the storage cost of our multi-indexing algorithm is close to linear in $n$, the multi-indexing algorithm of Greene *et al.* [6] entails storage complexity that is super-linear in $n$. To find all $r$-neighbors, for a given search radius $r$, they construct $m = O(r2^{sr/b})$ substrings of length $s$ bits per binary code. Their suggested substring length is also $s = \log_2 n$, so the number of substring hash tables becomes $m = O(rn^{r/b})$. And of course each such hash table requires $O(n\log_2 n)$ amount of storage. As a consequence for large values of $n$, even with small $r$, this technique requires a prohibitive amount of memory to store all of the hash tables.

Our approach is more memory-efficient than that of [6] because we do not enforce exact equality in substring matching. In essence, instead of creating all of the hash tables off-line, and then having to store them, we flip bits of each substring at run-time and implicitly create some of the substring hash tables on-line. This increases run-time slightly, but greatly reduces storage costs.

## 4. $K$-Nearest Neighbors Search

To use multi-index hashing in practice, one must specify a Hamming search radius $r$. For many tasks, the value of $r$ is chosen such that queries will, on average, retrieve $K$ near neighbors. Nevertheless, we find that, for many hashing

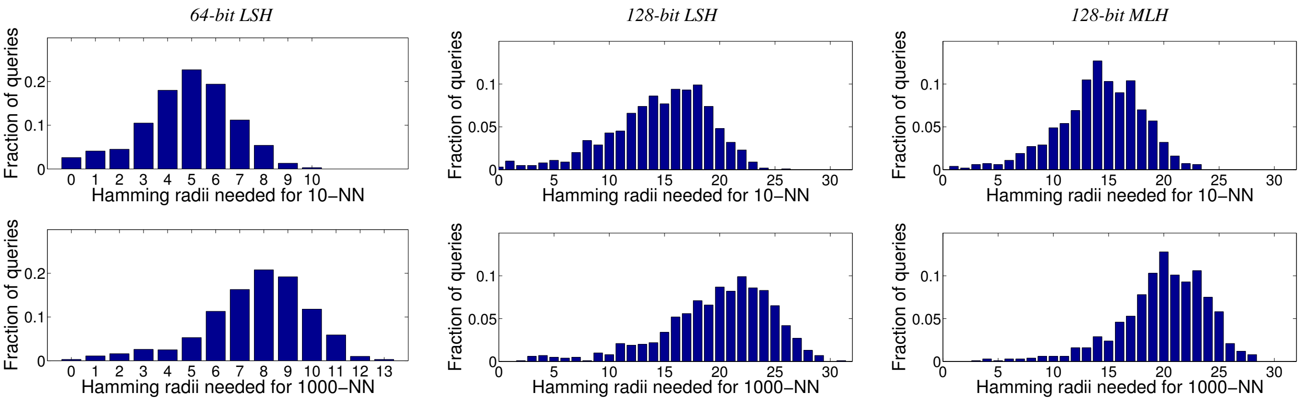


Figure 4. Shown are histograms of the search radii that are required to find 10-NN and 1000-NN, for 64 and 128-bit code from LSH [2], and 128-bit codes from MLH [16], based on 1B SIFT descriptors [11]. Clearly shown are the relatively large search radii required for both the 10-NN and the 1000-NN tasks, as well as the increase in the radii required when using 128 bits versus 64 bits.

techniques and different sources of visual data, the distribution of binary codes is such that a single search radius for all queries will not produce similar numbers of neighbors.

Figure 4 depicts empirical distributions of search radii required for 10-NN and 1000-NN on three corpora of binary codes obtained from 1 billion SIFT descriptors [11, 13]. In all cases, for 64 and 128-bit codes, and for hash functions based on LSH [2] and MLH [16], there is a substantial variance in the search radius. This suggests that binary codes are not uniformly distributed over the Hamming space. As an example, for 1000-NN in 64-bit LSH codes, more than $10\%$ of the queries require a search radius of 10 bits or larger, while for about $10\%$ of the queries it can be 5 or smaller. Also evident from Fig. 4 is the growth in the required search radius as one moves from 64-bit codes to 128 bits, and from 10-NN to 1000-NN.

A fixed radius for all queries would produce too many neighbors for some queries, and too few for others. It is therefore more natural for many tasks to fix the number of required neighbors, *i.e.,* $K$, and let the search radius depend on the query. Fortunately, our multi-index hashing algorithm is easily adapted to accommodate query-dependent search radii. Given a query, one can progressively increase the Hamming search radius per substring, until a specified number of neighbors is found. For example, if one examines all $r'$-neighbors of a query's substrings, from which more than $K$ candidates are found to be within a Hamming distance of $(r'+1)\,m-1$ bits (using the full codes for validation), then it is guaranteed that $K$-nearest neighbors have been found. Indeed, based on our proposition, if all $K$-NN of a query $\mathbf{q}$, differ from $\mathbf{q}$ in $(r'+1)\,m-1$ bits or less, then they will be found in $\{r'\text{-neighbors of } \mathbf{q}^{(k)}\}_{k=1}^{m}$ given by the substring hash tables.

In our experiments, we follow this progressive increment of the search radius until we can find $K$-NN in the guaranteed neighborhood of a query. This approach is helpful because it uses a specific search radius for each query depending on the distribution of codes in that neighborhood.

## 5. Experiments

All experiments are run on a single core 2.0 GHz CPU with 256 GB of memory. Both linear scan and multi-index hashing were implemented in C++ and compiled with the same flags. The memory requirements for multi-index hashing are described, along with other implementation details, in the Appendix. We currently require approximately 86 GB for multi-index hashing with one billion 64-bit codes, and approximately twice that for 128-bit codes. As discussed in the Appendix, these numbers could be reduced significantly in a more memory-efficient implementation. Further, a distributed implementation of multi-index hashing is straightforward, in which each substring hash table is stored on a separate machine with much less memory.

Our implementation of the linear scan baseline searches almost 55 million 64-bit codes in just under a second. This is remarkably fast compared to Euclidean NN search with 128D SIFT vectors. The speed of linear scan is in part due to memory caching, without which it would be about 10 times slower. Without optimizing cache usage, our multi-index hashing method solves exact 1000-NN for a dataset of one billion 64-bit codes in less than 0.2 seconds, over 100 times faster than linear scan (see Table 1). Performance on 1-NN and 10-NN are even more impressive.

### 5.1. Datasets

Experiments are conducted on two well-known vision corpora: 80M Gist descriptors from 80 million tiny images [22] and 1B SIFT features from the BIGANN dataset [11]. SIFT vectors [13] are 128D descriptors of local image structure in the vicinity of feature points. Gist [17] features extracted from $32\times32$ images capture global image

|  |  |  | **speed-up factors for $K$-NN *vs.* linear scan** |  |  |  |  |
|---|---|---|---|---|---|---|---|
| dataset | nbits | mapping | 1-NN | 10-NN | 100-NN | 1000-NN | linear scan |
| SIFT 1B | 64 | MLH | 213 | 205 | 182 | 126 | 18.03s |
|  |  | LSH | 229 | 213 | 175 | 107 |  |
|  | 128 | MLH | 272 | 170 | 87 | 37 | 35.33s |
|  |  | LSH | 204 | 114 | 56 | 25 |  |
| Gist 79M | 64 | MLH | 161 | 128 | 78 | 33 | 1.41s |
|  |  | LSH | 169 | 80 | 31 | 8 |  |
|  | 128 | MLH | 58 | 21 | 11 | 6 | 2.74s |
|  |  | LSH | 28 | 12 | 6 | 3 |  |

Table 1. Summery of results for 8 datasets of binary codes. The first four rows correspond to 1 billion binary codes, while the last four rows show the results for 79 million codes. Codes are either 64 or 128 bits long, obtained by LSH or MLH. The run-time of linear scan is reported along with the speed-up factors for $K$-NN with multi-index hashing.

structure in 384D vectors. These two feature types cover a wide spectrum of NN search problems in computer vision from feature to image indexing.

We use two similarity-preserving mappings to create datasets of binary codes, namely, binary Locality Sensitive Hashing (LSH) [3], and Minimal Loss Hashing (MLH) [16]. LSH is considered a baseline random projection method, closely related to cosine similarity. MLH is a state-of-the-art learning algorithm that, given a set of similarity labels, finds an optimal mapping by minimizing a loss function over pairs of binary codes.

Both the 80M Gist and 1B SIFT corpora comprise a training set, a base set, and a test query set. Gist descriptors are randomly divided into a base set of 79 million items, 1000 items for the query set, and 300$K$ for the training set. The SIFT corpus is already divided into three sets, 100M for training, $10^9$ in the base set, and $10^4$ for the test queries. The training sets are used to adjust the hash function parameters. For LSH we subtract the mean, and pick a set of coefficients from a normal density for a linear transform, followed by quantization. For MLH the training set is used to optimize several hash function parameters [16]. After learning is complete, we remove the training data and use the resulting hash function with the base set to create the database of binary codes. With two image corpora (SIFT and Gist), two code lengths (64 and 128 bits), and two hashing methods (LSH and MLH), we obtain 8 datasets of binary codes with which to evaluate our multi-index hashing algorithm.

Figure 5 shows Euclidean NN recall rates based on $K$-NN search on binary mappings of 1M and 1B SIFT descriptors. In particular, we plot the fraction of Euclidean $1^{st}$ nearest neighbors found by $K$-NN in 64-bit and 128-bit LSH [3] and MLH [16] binary codes. As expected 128-bit codes are more accurate, and MLH outperforms LSH. Since multi-index hashing solves exact $K$-NN in Hamming distance, the approximations are only due to the binary quantization. To preserve enough of the similarity structure in the original SIFT descriptors, it seems necessary to use 64 or 128-bit codes, and exploit data-dependant hash functions such as MLH. Interestingly, as described below, the speedup factors of multi-index hashing on MLH codes are better than those for LSH.

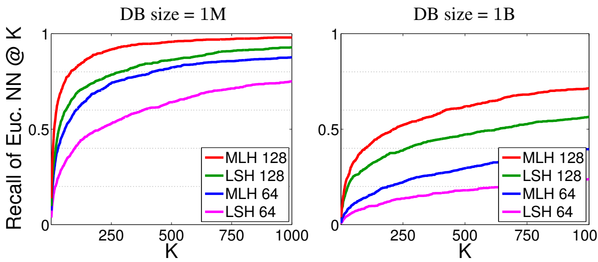


Figure 5. Recall rates for BIGANN dataset [11] (1M and 1B subsets) obtained by $K$-NN on 64- and 128-bit MLH and LSH codes.

## 5.2. Results

Each experiment involves 1000 queries for which we report the average run-time. Table 1 shows run-time per query for the linear scan baseline, along with speed-up factors of multi-index hashing for different $K$-NN problems and the 8 datasets. The running time of linear scan does not depend on the number of neighbors, nor on the underlying distribution of binary codes. The running time for multi-index hashing, however, depends on both factors.

As the desired number of NNs increases, the Hamming radius of the search must also increase (*e.g.,* see Fig. 4); this implies longer run-times for multi-index hashing. Indeed, notice that going from 1-NN to 1000-NN on each row of Table 1 shows decreasing speed-up factors.

The multi-index hashing running time also depends on the distribution of binary codes. Indeed, one can see from Table 1 that MLH code databases yield faster run times than the LSH codes; e.g., for 1000-NN in 1B 128-bit codes the speed-up for MLH is 37× *vs* 25× for LSH. Fig. 4 depicts the histograms of search radii needed for 1000-NN with 1B 128-bit MLH and LSH codes. Interestingly, the mean of the search radii for MLH codes is 19.9 bits, while it is 19.8 for LSH. This suggests that LSH might be marginally faster to

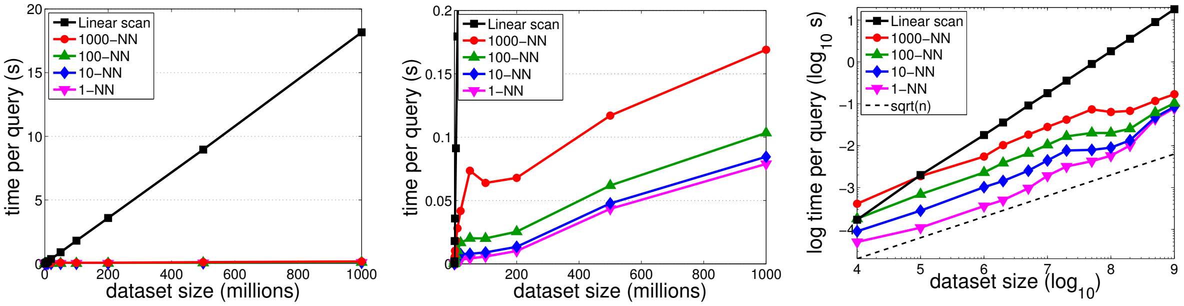


Figure 6. Run-times per query for multi-index hashing with 1, 10, 100, and 1000 nearest neighbors, and a linear scan baseline on 1B 64-bit binary codes given by LSH from SIFT.

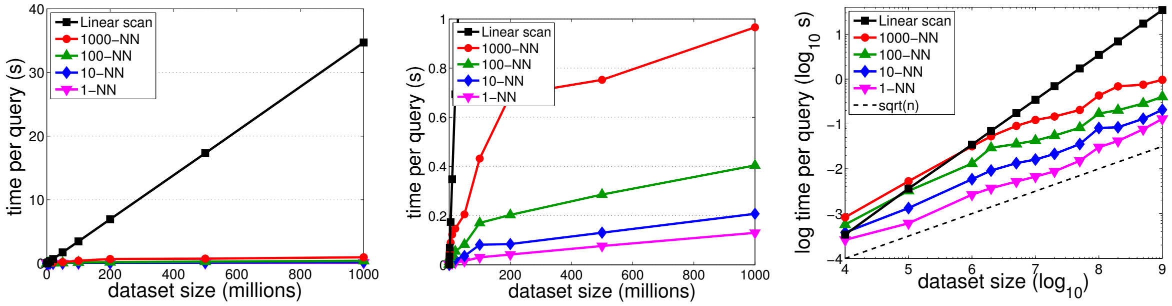


Figure 7. Run-times per query for multi-index hashing with 1, 10, 100, and 1000 nearest neighbors, and a linear scan baseline on 1B 128-bit binary codes given by MLH from SIFT.

search. On the other hand, the standard deviations of the search radii for MLH and LSH are 4.0 and 5.0 respectively. The longer tail of the distribution of search radii for LSH plays an important role in the expected running time. In fact, queries that require relatively large search radii tend to dominate the average query cost. From these observations, multi-index hashing performs best when the mean of required radii is small (*e.g.,* in case of 1-NN *vs.* 1000-NN) and when the standard deviation is small too (*e.g.,* in case of 1000-NN MLH *vs.* LSH).

It is also interesting to look at the multi-index hashing run-times as a function of the number of binary codes in the database *i.e.,* $n$. To that end, Fig. 6 and 7 depict run-times for linear scan, and multi-index $K$-NN search. The left two figures in each show different vertical scales (since the behavior of multi-index $K$-NN and linear scan are hard to see at the same scale). The right-most panels show the same data on log-log axes. First, it is clear from these plots that multi-index hashing is much faster than linear scan for a wide range of dataset sizes and $K$. Just as importantly, it is evident from the log-log plots that as we increase the database size, the speedup factors improve. The dashed lines on the log-log plots depict $\sqrt{n}$ as function of $n$ (up to a scalar constant). The similar slope of multi-index hashing curves with the square root curves suggests that multi-index hashing has sub-linear query time, even for the empirical, non-uniform distributions of codes.

## 6. Conclusion

This paper describes a new algorithm for exact nearest neighbor search on large-scale datasets of binary codes. The algorithm is a form of multi-index hashing that has provably sub-linear run-time behavior for uniformly distributed codes. It is storage efficient and easy to implement. We show empirical performance on datasets of binary codes obtained from 1 billion SIFT, and 80 million Gist features. With these datasets we find that, for 64-bit and 128-bit codes, our new multi-index hashing implementation is often more than 100 times faster than a linear scan baseline.

## References

[1] M. Aly, M. Munich, and P. Perona. Distributed kd-trees for retrieval from very large image collections. *BMVC*, 2011.

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Comm. ACM*, 51:117–122, 2008.

[3] M. Charikar. Similarity estimation techniques from rounding algorithms. *ACM STOC*. 2002.

[4] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[5] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. *IEEE CVPR*, 2011.

[6] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. *IEEE FOCS*, pp. 722–731, 1994.

[7] J. He, R. Radhakrishnan, S.-F. Chang, and C. Bauer. Compact hashing with joint optimization of search accuracy and time. *IEEE CVPR*, 2011.

[8] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. *ACM STOC*, pp. 604–613, 1998.

[9] H. Jégou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. *ECCV*, v. I, pp. 304–317, 2008.

[10] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE PAMI*, 33:117–128, 2011.

[11] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. *IEEE ASSP*, pp. 861–864. 2011.

[12] B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. *NIPS*, 2009.

[13] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60:91–110, 2004.

[14] M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1969.

[15] M. Muja and D. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISSAPP*, 2009.

[16] M. Norouzi and D. J. Fleet. Minimal Loss Hashing for Compact Binary Codes. *ICML*, 2011.

[17] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *IJCV*, 42:145-175, 2001.

[18] M. Raginsky and S. Lazebnik. Locality-sensitive binary codes from shift-invariant kernels. *NIPS*, 2009.

[19] R. Salakhutdinov and G. Hinton. Semantic hashing. *Int. J. Approx. Reasoning*, 2009.

[20] G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. *IEEE ICCV*, 2003.

[21] C. Strecha, A. Bronstein, M. Bronstein, and P. Fua. Ldahash: improved matching with smaller descriptors. *IEEE PAMI*, 34:66–78, 2012.

[22] A. Torralba, R. Fergus, and W. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE PAMI*, 30:1958–1970, 2008.

[23] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. *IEEE CVPR*, 2008.

[24] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. *NIPS*, 2008.

## A. Implementation Details

Our algorithm is essentially a set of exact-match queries in disjoint $s$-bit substrings of the codes. When substring length is small (*e.g.,* $s \leq 32$) one can explicitly allocate memory for $2^s$ buckets and store all the data points associated with each substring in its correspoding bucket. Then, given a query one fetches all the relevant data points by $m$ address lookups with the $m$ substrings of the query.

**Folded Hash Tables:** When $s$ is large, the $2^s$ addresses required for explicit memory allocation becomes infeasible. Instead one could use folded hash tables, for which the binary substring indices are hashed into smaller hash tables *e.g.,* by taking them modulo a prime number. This approach is storage efficient, but slower because it requires a method to handle hash collisions. We avoided using folded hash tables since the longest substring we need is 32 bits.

**Memory Requirements:** A hash table in our implementation is an array of pointers to resizable arrays. We store one 64-bit pointer for each hash table bucket; this entails 32 GB for an empty 32-bit hash table. There are simple ways to store empty buckets much more efficiently, but our current implementation is not optimized for memory.

With $m$ (unfolded) substring hash tables of length $s$ bits, and a 64-bit address per bucket, the empty hash tables requires $m2^s8$ bytes. For each non-empty bucket a resizable array is allocated to store the associated data points. Resizable arrays are preferred over linked lists since they are more cache friendy. To store the size of the resizable arrays, at most $4m\min(n, 2^s)$ bytes are needed as the number of non-empty buckets is bounded by $m\min(n, 2^s)$. For each data point per hash table we store an ID to reference the full binary code; each ID is 4 bytes as the size of datasets $n \leq 2^{32}$; this yields a total of $4mn$ bytes. Lastly, storing the full binary codes requires $nms/8$ bytes, because $b = ms$.

In total, the memory cost is $4m(2^{s+1} + \min(n, 2^s) + n + ns/32)$ bytes. For one billion 64-bit codes, and two chunks (32 bits each), this cost is 86 GB. Note that the last two terms (for the IDs and binary codes) are irreducible, but the first term can be reduced in a memory efficient implementation at least by a factor of two. The first term heavily dominates the storage cost. If we search 60-bit binary codes instead of 64-bit ones, then $s = 30$ and the storage cost drops to 41 GB. For 128-bit codes our implementation requires 186 GB of storage, and for 120-bit codes, 82 GB.

**Duplicate Candidates:** When retrieving candidates from the $m$ substring hash tables, inevitably we will find some codes multiple times. In order to catch such *duplicates*, and discard them, we allocate a bit-string with $n$ bits. Every time a candidate is found, we check the corresponding bit. If it is 1, we discard the candidate as a duplicate. Otherwise it is set to 1 and we retain the candidate. Before each query we clear the bit array, which in theory requires $O(n)$ but negligible in practice.

**Hamming Distance:** To compare a query and a candidate, used in both multi-index search and linear scan, we compute the Hamming distance on the full $b$-bit codes, with one `xor` operation for every 64 bits followed by a pop count to tally the ones. We used the built-in GCC function `__builtin_popcount`.

**Number of Substrings:** The number of substrings we use is determined with a hold-out validation set of database entries. From that set we estimate the running time of the algorithm for different choices of $m$, and select the $m$ that yields the best results. Our experiments show that this empirical value for $m$ is typically very close to $b\,/\log_2 n$.