Fast Search in Hamming Space with Multi-index Hashing

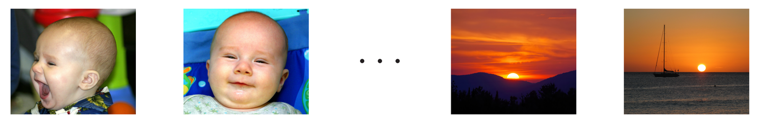
Mohammad Norouzi Ali Punjani David J. Fleet
University of Toronto

Problem Context

Open Problem: *Exact* sub-linear nearest neighbor search in Hamming distance on binary codes.

Context: *Fast* similarity search with large, high-dimensional datasets: images, videos, documents, <*your data here*>.

1. Map data-points onto similarity-preserving binary codes:
   - Similar data items should map to nearby codes
   - Dissimilar data items map to distant codes



… 110010 100010 … 000101 001101 …

2. Perform nearest-neighbor search in the Hamming space.

Why binary codes?

– Binary codes are storage-efficient.
– Hamming distance is inexpensive to compute.

Key Tasks: Given a corpus of b-bit codes, and a query $\mathbf{q}$,

– Find r-*neighbors*: find all codes in the database that differ from $\mathbf{q}$ in r bits or less (*aka.* Point-Location in Equal Balls).
– kNN: find k codes with k smallest Haming distances from $\mathbf{q}$.

Linear Scan *vs.* Hash Indexing

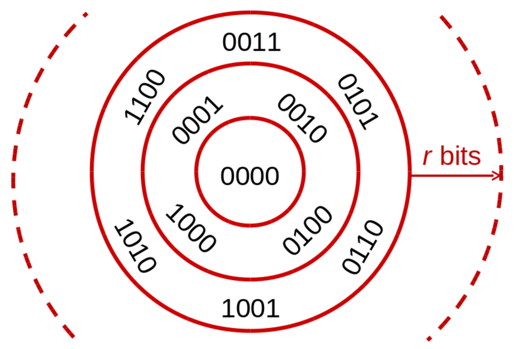
How to structure the database, so that r-neighbors and kNN queries can be answered quickly?

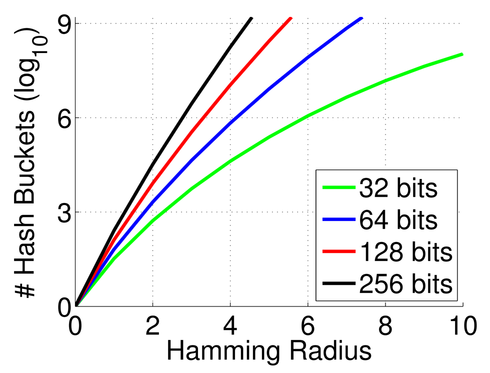
(1) Exhaustive search (*i.e.*, linear scan through the database) ~ 50 million comparisons/second.

(2) Populate a hash table with the database codes. At query time, flip bits of $\mathbf{q}$ and lookup the entries in the vicinity of $\mathbf{q}$.

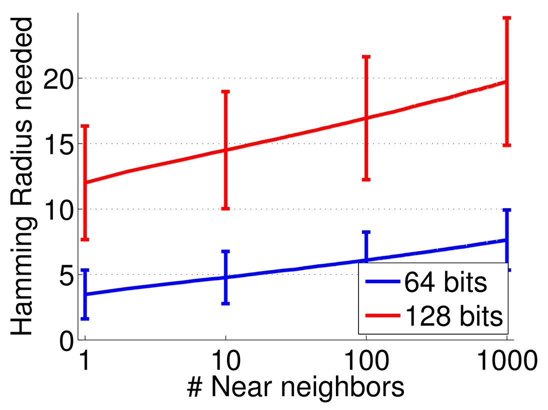
Issues with hash indexing:

– Volume of the Hamming ball grows near-exponentially in r.
$V(b,r) = 1 + \binom{b}{1} + \binom{b}{2} + \ldots + \binom{b}{r}$





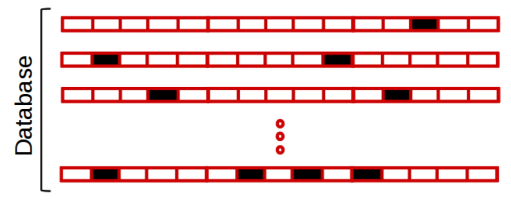
– For typical databases / tasks, a large search radius r is necessary. The following plot is produced from 1B LSH codes on SIFT.



Conclusion: For binary codes longer than 32 bits, linear scan is more effective than vanilla hash indexing.

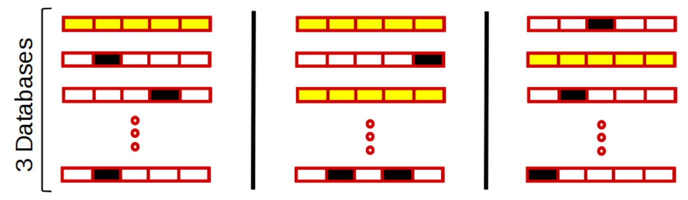
Multi-Index Hashing – Idea

Imagine a dataset of 15-bit codes, and a search radius of $r = 2$. Black marks depict bits that differ from a given query.



(Note: the first 3 codes are the 2-neighbors of the query.)

Key Idea: Partition the codes into 3 substrings. Then, instead of searching $r=2$ in the full codes, search $r=0$ in the substrings.



In general, partition codes into m substrings $\mathbf{h} \equiv (\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(m)})$. Instead of exploring a Hamming ball of radius r in the full codes, search a radius of $\lfloor r/m \rfloor$ in the substrings. This works because:

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, *i.e.*,

$$\|\mathbf{h} - \mathbf{g}\|_H \leq r \implies \exists k \quad \|\mathbf{h}^{(k)} - \mathbf{g}^{(k)}\|_H \leq \left\lfloor \frac{r}{m} \right\rfloor ,$$

where $\|\,.\,\|_H$ is the Hamming norm.

– Resembles the pigeonhole principle.
– This condition is necessary but not sufficient. Thus, we retrieve a superset of r-neighbors, and then cull the non-r-neighbors.

Key benefits of Multi-Index Hashing:

– Search occurs on much smaller binary code lengths
– Search radius is much smaller

Multi-Index Hashing – Algorithm

Data structure:

– Given m, partition each database code into m disjoint pieces.
– Generate m hash tables with the m substrings of each code.

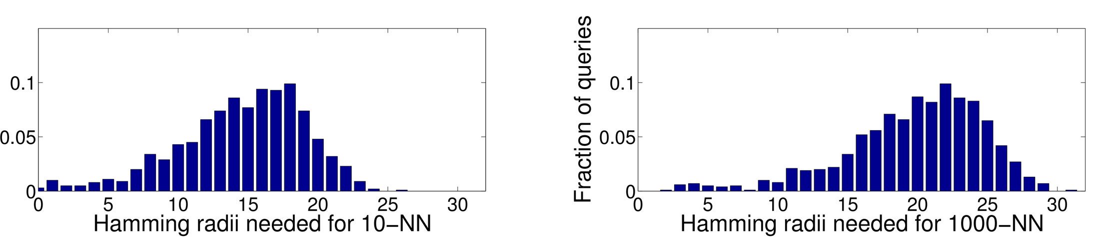
Finding r-neighbors:

Given a query $\mathbf{q}$ with substrings $\{\mathbf{q}^{(i)}\}_{i=1}^m$,

1. **Lookups:** 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 $\mathcal{N}_i(\mathbf{q})$.
2. **Candidates:** Take the union of the m sets, $\mathcal{N}(\mathbf{q}) = \bigcup_i \mathcal{N}_i(\mathbf{q})$, and prune the duplicates. The set $\mathcal{N}(\mathbf{q})$ is necessarily a superset of the r-neighbors of $\mathbf{q}$.
3. **Evaluation:** Compute the Hamming distance between $\mathbf{q}$ and each candidate in $\mathcal{N}(\mathbf{q})$, retaining only the true r-neighbors.

Finding kNNs:

Find r-neighbors with progressively increasing values of r until k items are found.



1B 128-bit LSH codes

Time and Space Complexity

Notation:

n: number of binary codes
b: bit length
r: radius of Hamming search
m: number of substrings
s: substring length ($s = b/m$)

Assume: $\begin{cases} \text{substring length } s = \log_2 n \\ \text{uniformly distributed codes (for run-time)} \end{cases}$

Run-time:

$$\text{query cost} \;\leq\; 2\,\frac{b}{\log_2 n}\, n^{H(r/b)},$$

where $H(\epsilon) \equiv -\epsilon \log_2 \epsilon - (1-\epsilon)\log_2(1-\epsilon)$.

$r/b \leq 0.06$	$r/b \leq 0.11$	$r/b \leq 0.17$
$O\left(\frac{b\, n^{1/3}}{\log_2 n}\right)$	$O\left(\frac{b\sqrt{n}}{\log_2 n}\right)$	$O\left(\frac{b\, n^{2/3}}{\log_2 n}\right)$

Storage: Multi-index hashing requires $m = n/\log_2 n$ hash tables. Each hash bucket stores identifiers for its codes. We also store n codes of length b bits. Thus, it can be shown that:

$$\text{space complexity is } O\left(n\,b + n \log_2 n\right)$$

Cost Model

Run-time per query depends on the #lookups and the #candidates. In general,

$$\#\text{lookups} \;=\; m\; V(s, r/m)$$

For n uniformly distributed codes we expect $n/2^s$ codes per hash bucket, so we expect

$$\#\text{candidates} \;=\; m\, \frac{n}{2^s}\, V(s, r/m)$$

Assuming the cost of 1 lookup equals the cost of 1 candidate test:

$$\text{cost}(s) \;=\; m\left(1 + \frac{n}{2^s}\right) V(s, r/m) \;\leq\; \frac{b}{s}\left(1 + \frac{n}{2^s}\right) 2^{s\,H(r/b)}$$

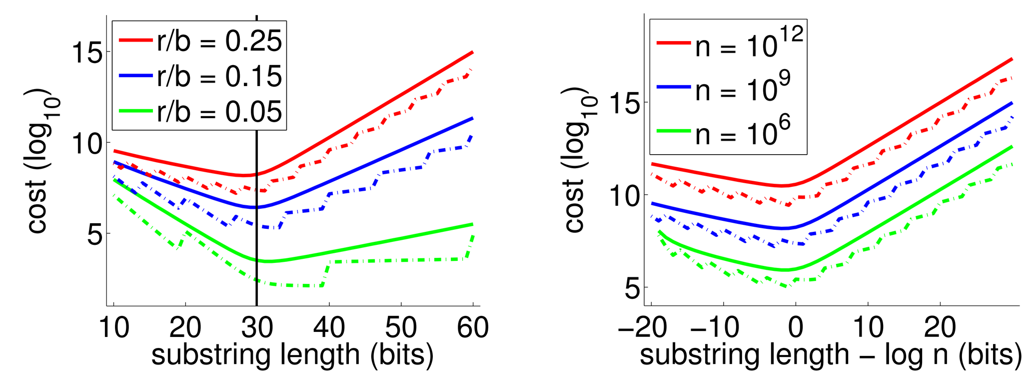
Optimal Substring Length

At the extremes:

– when $s = b$, then #lookups $= V(b,r)$, which grows too quickly.
– when $s = 1$, then #candidates $= n$, i.e., the entire dataset.

Analysis based on Stirling's approximation shows that the optimal substring length puts approximately one database entry in each substring hash bucket on average: $s^* \approx \log_2(n)$.

Plots show cost and its upper bound versus substring length, here with $b = 128$ bits. Note how minima are aligned at $s^* \approx log_2(n)$.



Left: for different search radii, all with $n = 10^9$ codes.

Right: for 3 database sizes, all for search radii $r = 0.25\,b$. (curves are displaced horizontally by $-\log_2(n)$).

Experiments

Hash Functions:

– LSH: Locality-sensitive Hashing [1]
– MLH: Minimal loss hashing [2]

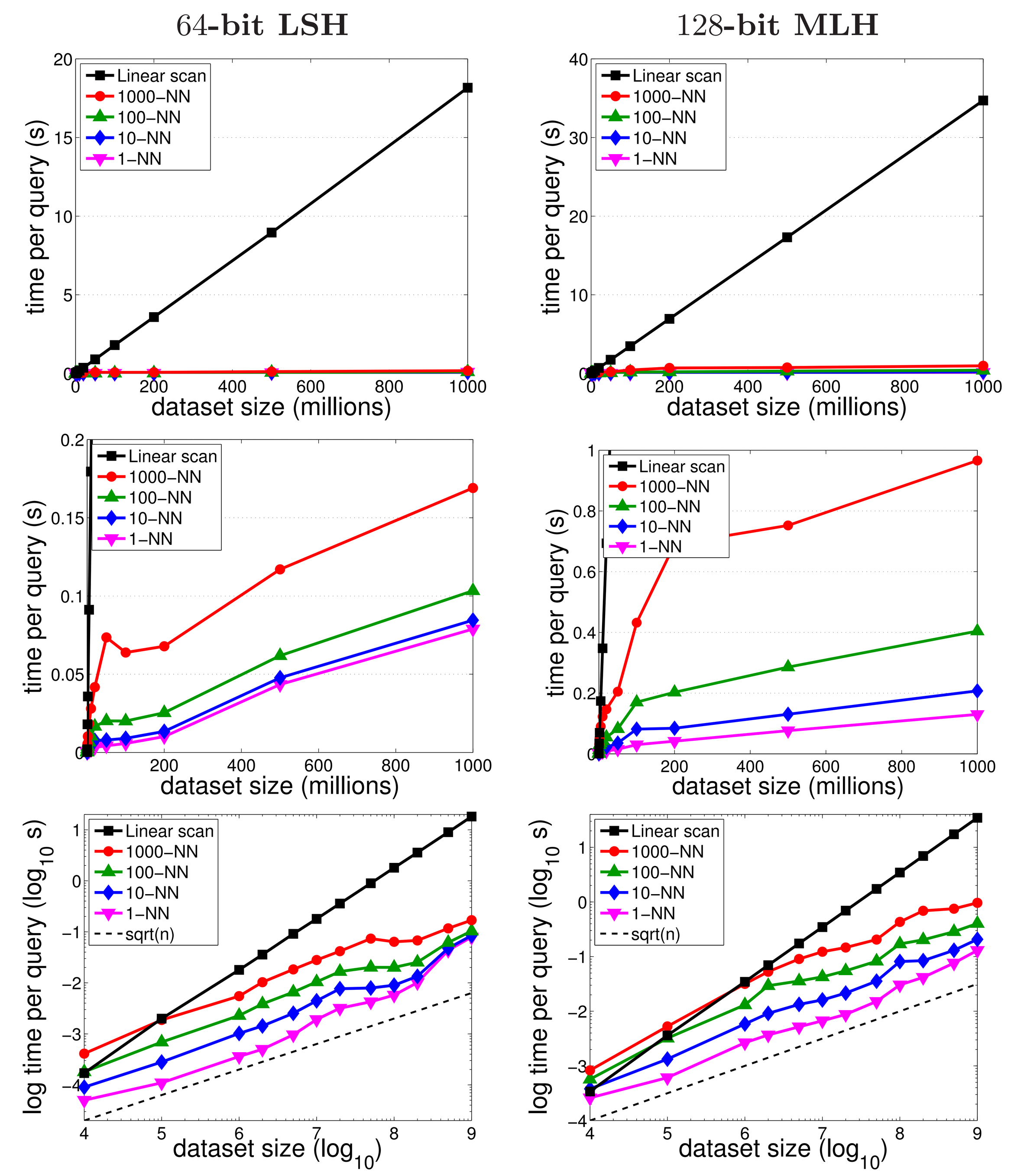
Datasets:

– 1 Billion SIFT descriptors [3]
– 80 Million tiny images (GIST) [4]

Retrieval Speed:

			speed-up factors for kNN				
dataset	nbits	map	1-NN	10-NN	100-NN	1000-NN	lin. 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	

Run-times per query for multi-index hashing with 1, 10, 100, and 1000 nearest neighbors, and a linear scan baseline on 1B codes from 128D SIFT descriptors:



Conclusions / References

Algorithm for exact nearest neighbor search in Hamming distance with theoretical guarantees and strong empirical results.

[1] Charikar (2002) Similarity estimation techniques from rounding algorithms. STOC.
[2] Norouzi & Fleet (2011) Minimal Loss Hashing for compact binary codes. ICML.
[3] Jegou, Tavenard, Douze, Amsaleg (2011) Searching in one billion vectors: re-rank with source coding. ASSP.
[4] Torralba, Fergus, Freeman (2008) 80 million tiny images: A large data set for nonparametric object and scene recognition. PAMI.