Lecture 22:

Additional notes on image matching and retrieval

Visual Computing Systems
CMU 15-869, Fall 2014

Solutions to the descriptor matching problem ("find nearest neighbor") discussed so far

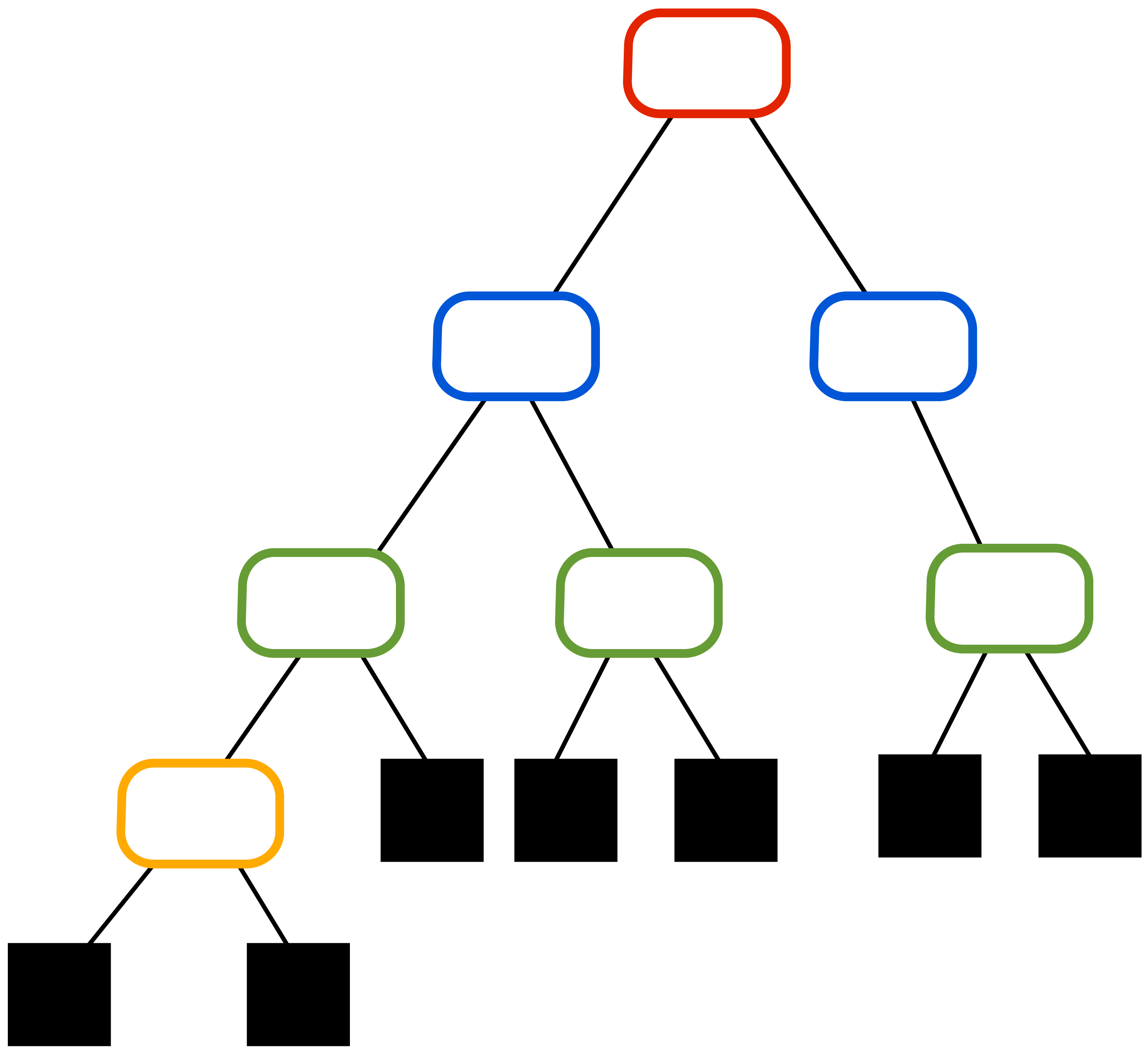
- **Baseline: brute-force linear scan**
  - Work inefficient: O(N) cost
  - Bandwidth intensive: either high memory footprint (if "in core") or must stream off disk
- **Inverted-index based acceleration:**
  - Quickly find candidate images that may contain good matches for the query image
  - Initial filter: reduce problem to finding matches in only these images
  - Good: compact index representation: 8 bytes per visual word occurrence (tf, document id)
  - Bad: loss of information in quantization of descriptor to visual words
- **KD-tree-based methods (single KD-tree or random forest)**
  - Good: like brute force, uses a full database representation (store full descriptors, not visual word vocal index)
  - Bad: high storage cost
    - Example: 1M images, 1K SIFT descriptors per image, 128 floats per descriptor = 128GB
    - Must also store tree node structures (can be expensive if using a forest for ANN)
- **No acceleration structure for DB, but prioritize order of linear scan of database**
  - Assumption: likely to terminate early once a good enough match is found

Common set of trade-offs

- Compute cost
  - Amount of work to perform search
- Storage / memory footprint
  - For database elements
  - For storing index structures (overhead)
- Result quality
  - Quality of results (visual words more compact than descriptors, forest likely to give better ANN results than a single random K-D tree)

Thought experiment

- Consider database of millions of images
- What if a search tree does not fit in memory on a single node? *

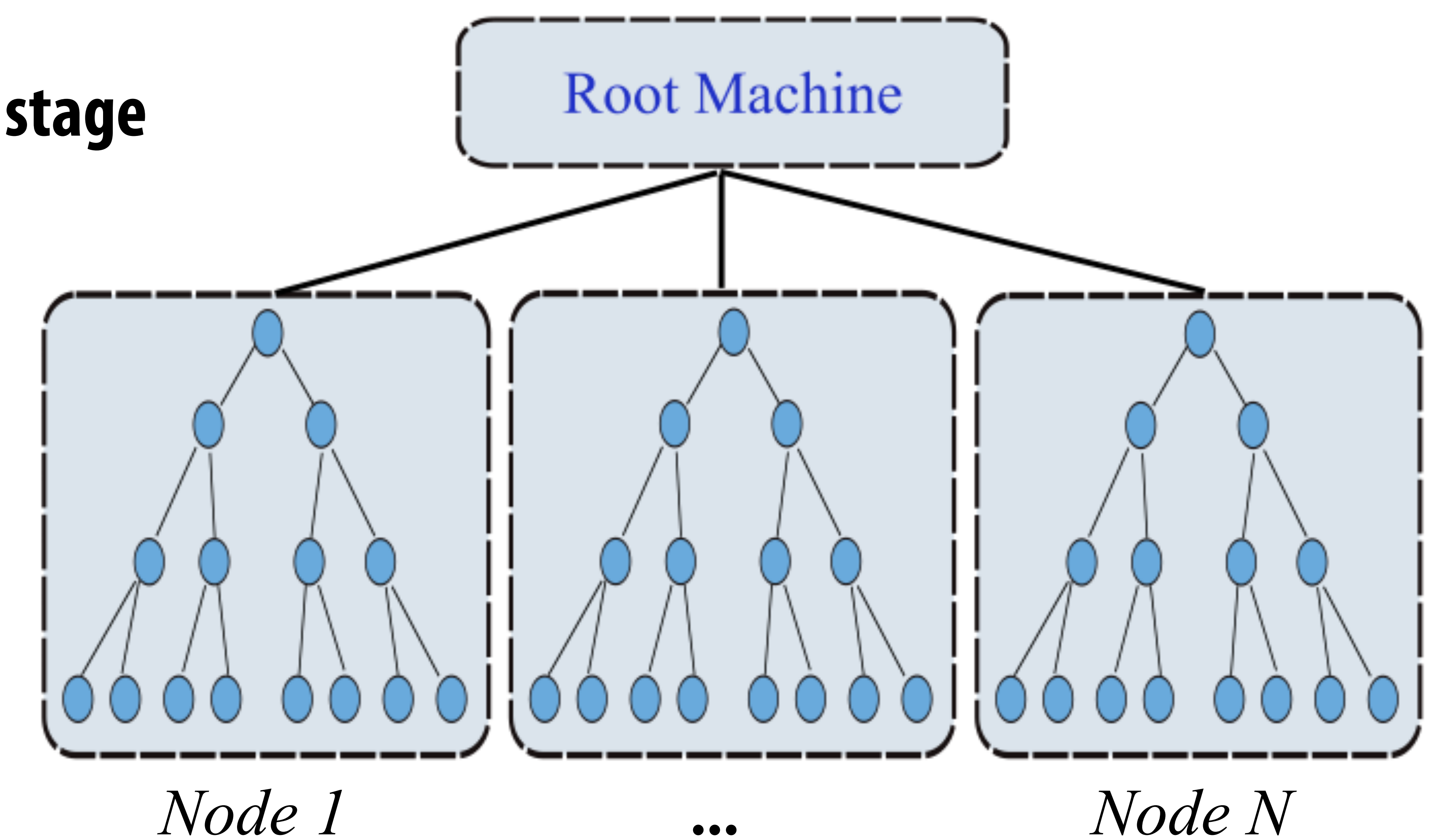


* Yes, ray tracing of complex scenes poses a similar problem

Distributing a search tree

- **Simple solution:**
  - Partition dataset into chunks of data points that fit in memory on a node
  - Build K-D trees independently and in parallel on all nodes
  - For each query:
    - Broadcast query to all N nodes
    - Run N independent k-NN searches in parallel
    - Broadcast results to a master node
    - Master sorts results to produce overall k-NN

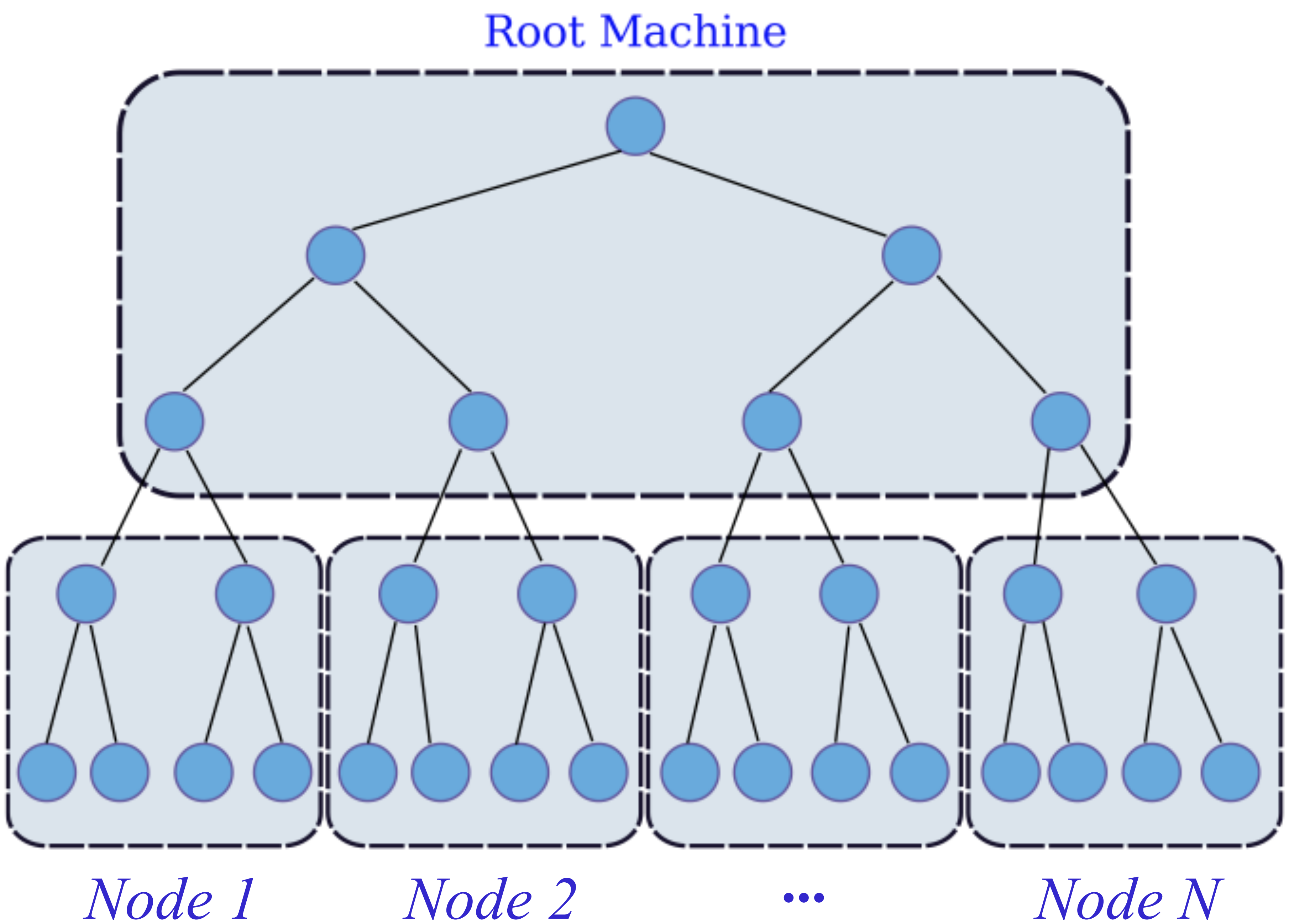
- **Problems:**
  - Lack of parallelism in the combine results stage
  - Less efficient structure
    - N independent K-D tree lookups
    - Search through single, large K-D tree would visit fewer nodes



[Figure credit: Aly et al. VISAPP 2011]

Distributing a search tree

- Idea: store top part of tree in master, bottom parts of tree are distributed across nodes
- Tree construction:
  - Build top subtree using sparse sampling of entire dataset that fits in memory
  - Top subtree height must be at least lg(N) (to generate N leaf trees for N machines)
  - For each remaining datapoint:
    - Determine which subtree data belongs inside
    - Build leaf trees in parallel on respective nodes



[Figure credit: Aly et al. VISAPP 2011]

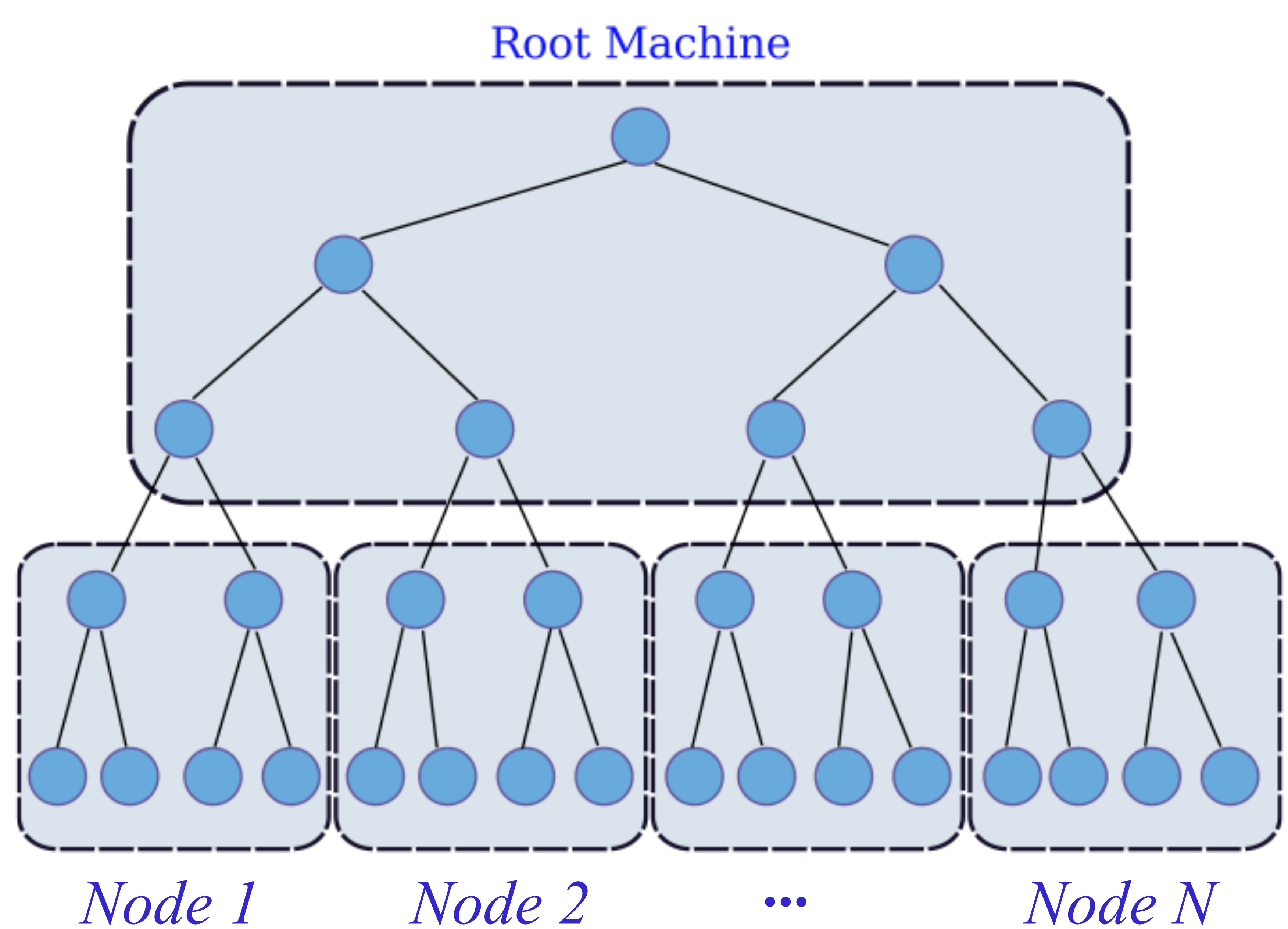
Distributing a search tree

- For each query image:
  - Compute features, for each feature:
    - Search top of tree, find all leaf nodes within distance *d* to query
    - Send query to these leaf nodes
    - All leaf nodes carry out search in parallel
  - Send k-NN results back to master for combination

- Good:
  - Efficacy similar to single big tree (each node contains an actual subtree, not a subsampling of data points)

- Bad: serialization of work at root

- Optimizations:
  - Replicate root tree to increase over system throughput (but not individual query latency)

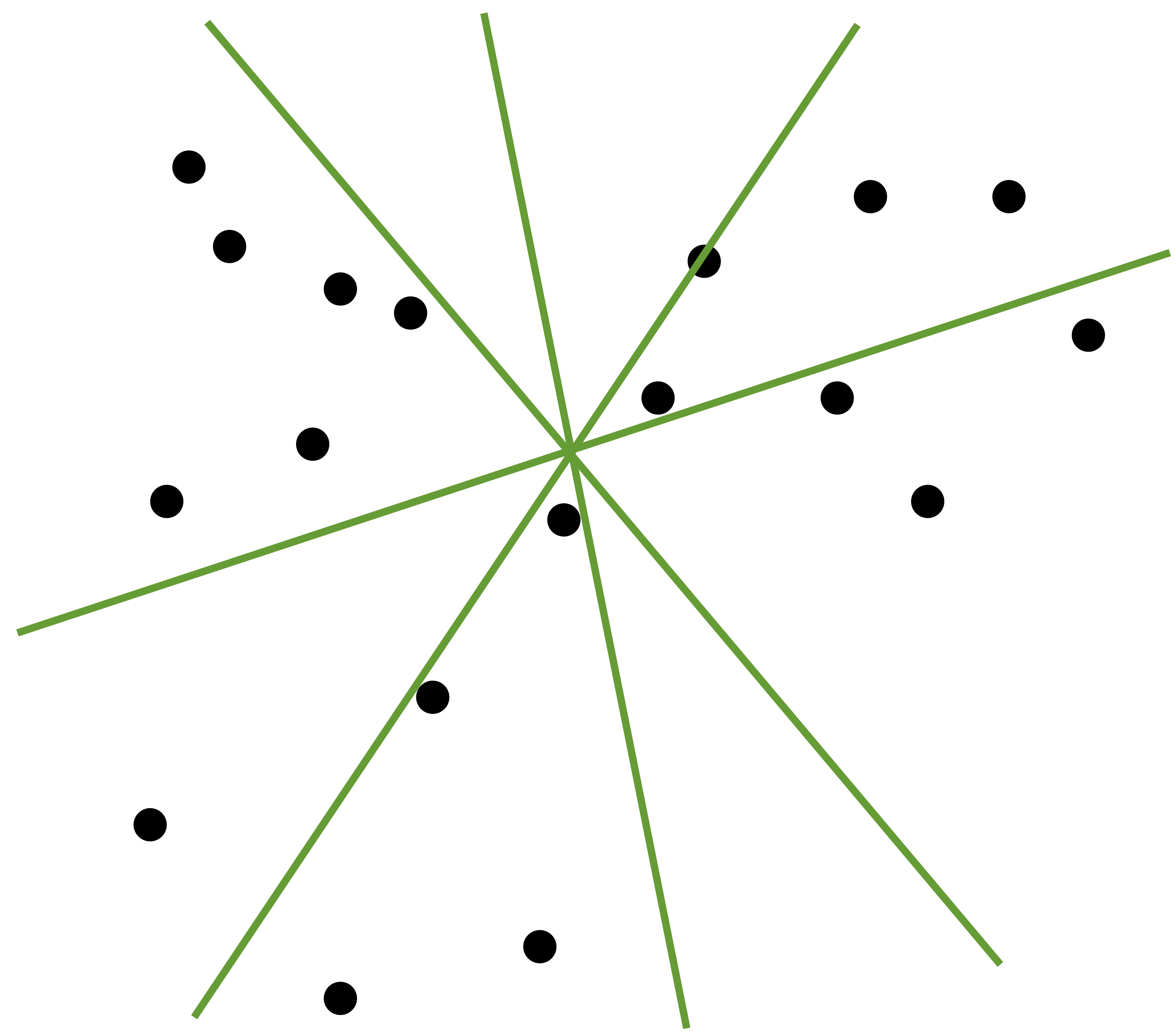


Locality sensitive hashing

- Accelerate search with a hash table lookup
- Challenge: want similar points to end up in similar hash bins
- Basic intuition:
    - Hash points into buckets, such that points nearby in space are likely to fall into the same (or nearby) buckets
- Given x1 and x2 and distance *r* in feature space
    - If $d(x1,x2) < r$, then $P(h(x1) = h(x2))$ is high
    - If $d(x1,x2) > \alpha r$, then $P(h(x1) = h(x2)$ is low

Locality sensitive hashing

- Example hash function: pick *m* random projections in N-D space
  - For each input query, hash query into *m* different hash keys (associated with *m* different hash tables)
  - Union of data points from matching bins is candidate nearest neighbor set
    - Compute full distance function on these points



Locality sensitive hashing (as an embedding)

- Example: pick *m* random projections (hyperplanes in N-D space)
  - For each input query, compute 1 bit per projection
  - Query descriptor is now represented as an m-bit string
  - 1 hash table containing (m-bit keys)
  - Check all hash bins with hamming distance similar to query!

Note: in practice, techniques seek to learn good hash functions from the dataset (rather than use random projections)

Benefits of NN search in hamming space

1. **Efficient distance computation:**
   - **Hamming distance: number of bits that differ between two *b*-bit codes**

```
int hamming_distance(bitstring x, bitstring y) {
    return count_bits( xor(x, y) );
}
```

2. **Compact database representation:**
   - ***bn* bits to store bitcodes for *n* images in database**
   - **Recall SIFT descriptor: 512 bits per keypoint, hundreds/ thousands of keypoints per image!**

Example: K-NN search (K=5) in hamming space:

[Torralba et al. 2008]

- 12.9M elements in database
  - Each element corresponds to full-image descriptor (384-element vector)
  - Full database size = 19.8 GB

- Brute-force search for top 5 nearest neighbors:
  - 30-bit codes: 400 MB of memory, 74 ms
  - 256-bit codes: 3.2 GB of memory, 0.23 sec

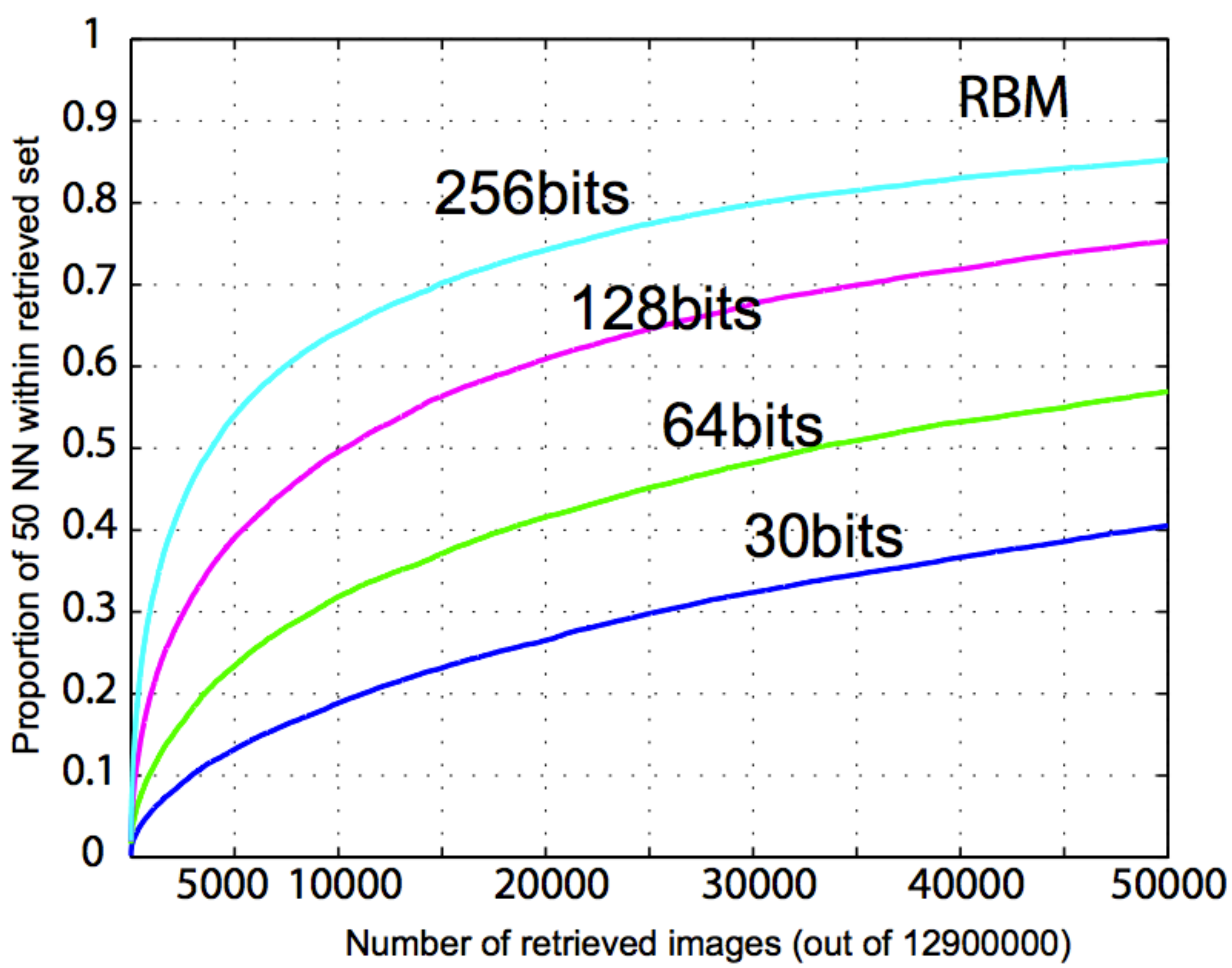
- Two orders of magnitude faster than brute force search (or K-D tree search) on database containing full-representation GIST (384-float-element) descriptors *

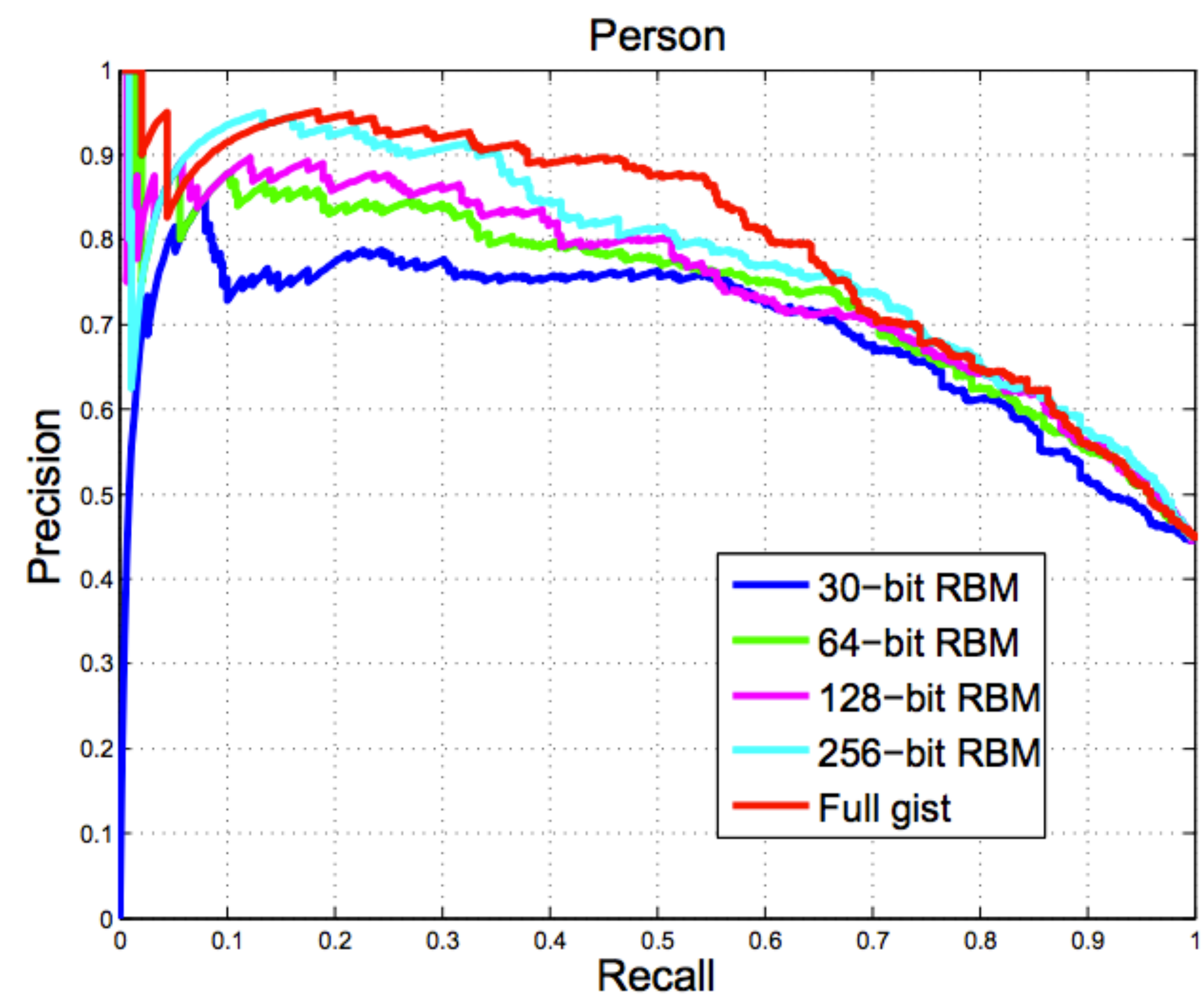
* Unfair comparison: should have compared to approximate k-NN implementation to be more fair since bitcode search results are not the same (see next slide)

Bitcode search "performance"

[Torralba et al. 2008]

- Baseline: GIST full image descriptor (384 floats)
- Experiment (left): compute top 50 NN in GIST-space, then measure how many of these NN appeared in the NN results in hamming space
- Experiment (right): object detection by transferring class label (person) from NN's to query image (does query picture contain a person?)





Benefits of NN search in hamming space

1. **Efficient distance metric computation:**
   - **Hamming distance: number of bits that differ between two *b*-bit codes**

2. **Compact database representation:**
   - ***bn* bits to store bitcodes for *n* images in database**

3. **Potential for using binary code directly as hash table index for O(1) search**

Simple problem formulation

- **Find all images within hamming distance *r* from query**
- **Search process: (assume 2^b indices in hash table)**

```
Compute b-bit key for query
For all indices within distance r from query:
    Add images in hashtable[index] to result set
```

- **Simple example: $r=0$, just check one bucket**

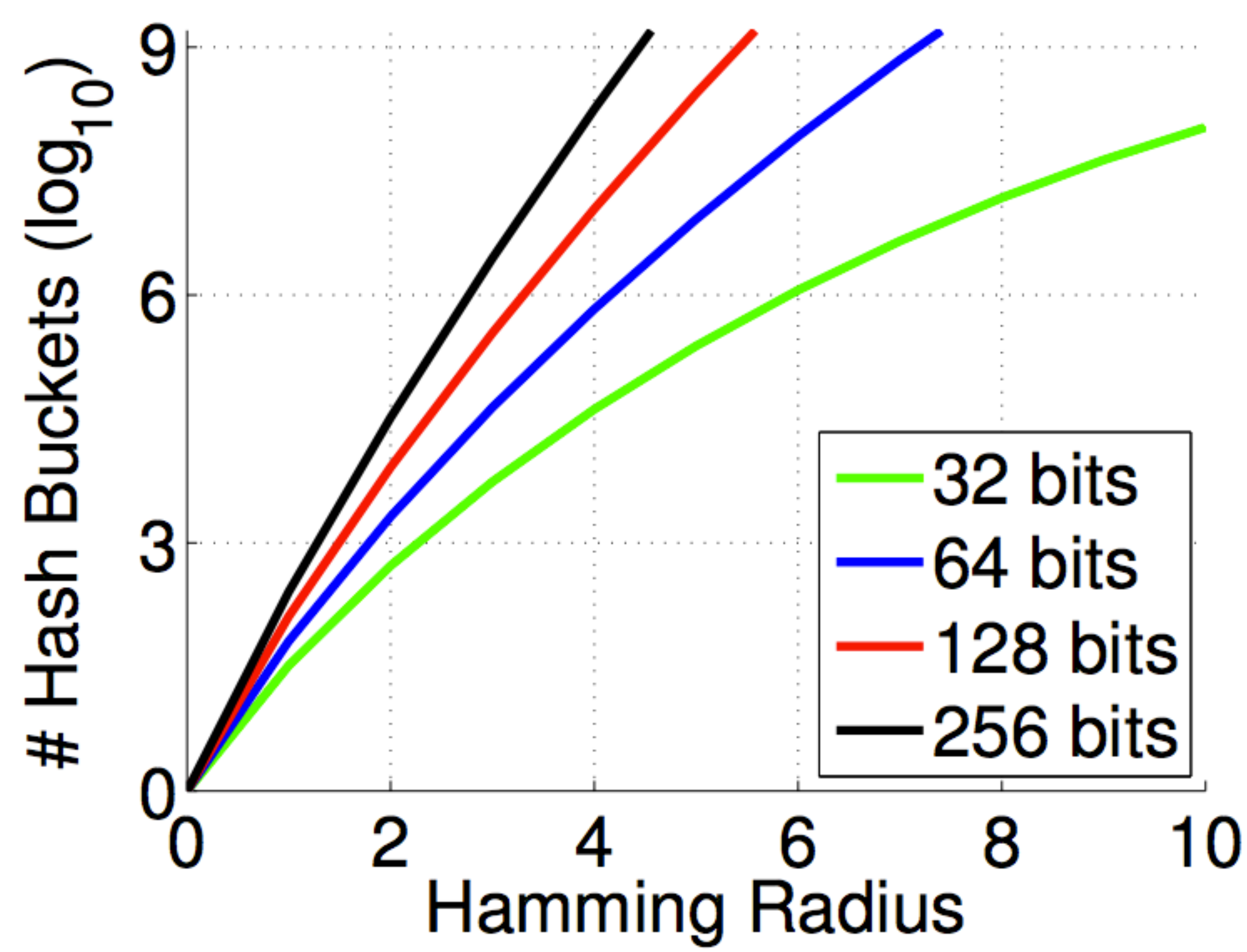
Problem

- **Number of buckets to check increases rapidly with *r***
  - Volume of the "hamming ball" of radius *r*
- **Number of candidate buckets:**

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

- **Example: *b*=64, then about 1B buckets for *r*=7**
  - If database is smaller than 1B elements, most of these indices will be empty
  - Consider database of millions of elements: faster to just run brute-force linear search through database!

[Norouzi et al. 2012]



Multi-index hashing: to improve k-NN search in hamming space

[Norouzi et al. 2012]

- **Basic intuition:**
  - Divide query bit string into *m* disjoint *b*/*m*-bit substrings
  - Bit strings that are close in one of the substrings might be close overall
- **Key idea:**
  - If binary codes *x* and *y* differ by at most *r* bits, then in one of their *m* substrings they must differ by at most floor(*r*/*m*) bits.
  - Proof by pigeon-hole principle (if they differed by more than *r*/*m* bits in each substring, then overall *x* and *y* must differ by more than *r* bits

Efficient k-NN using multi-index hashing

- **For each set of length-*m* substrings, find substrings of within Hamming radius of floor(*r*/*m*)**
  - **These are candidate strings within Hamming radius *r* of full query string**

- **Finding candidates is a much easier problem!**
  - **Previously: search needed to examine $L(b, r)$ hash buckets**
  - **Now need to examine only $L(b/m, \lfloor r/m \rfloor)$ buckets in *m* different hash tables (one table for each substring)**
  - **E.g., *r*=7, *m*=4, then only need to search with radius 1 in each of the substrings**

Full algorithm

- **Construct *m* hashtables using the length *b*/*m* substrings of elements in the original database (hashtable *i* contains all *i* th all substrings)**

- **Given *b*-bit query:**
  - **For each of the *m* substrings of the query:**
    - **Find radius floor(*r*/*m*) neighbors (in hashtable corresponding to current substring) and add them to candidate set**
    - **The candidate set is a superset of the true set of elements within hamming distance *r*, so compute actual set by performing full Hamming distance computation for all elements in candidate set (brute force linear scan)**

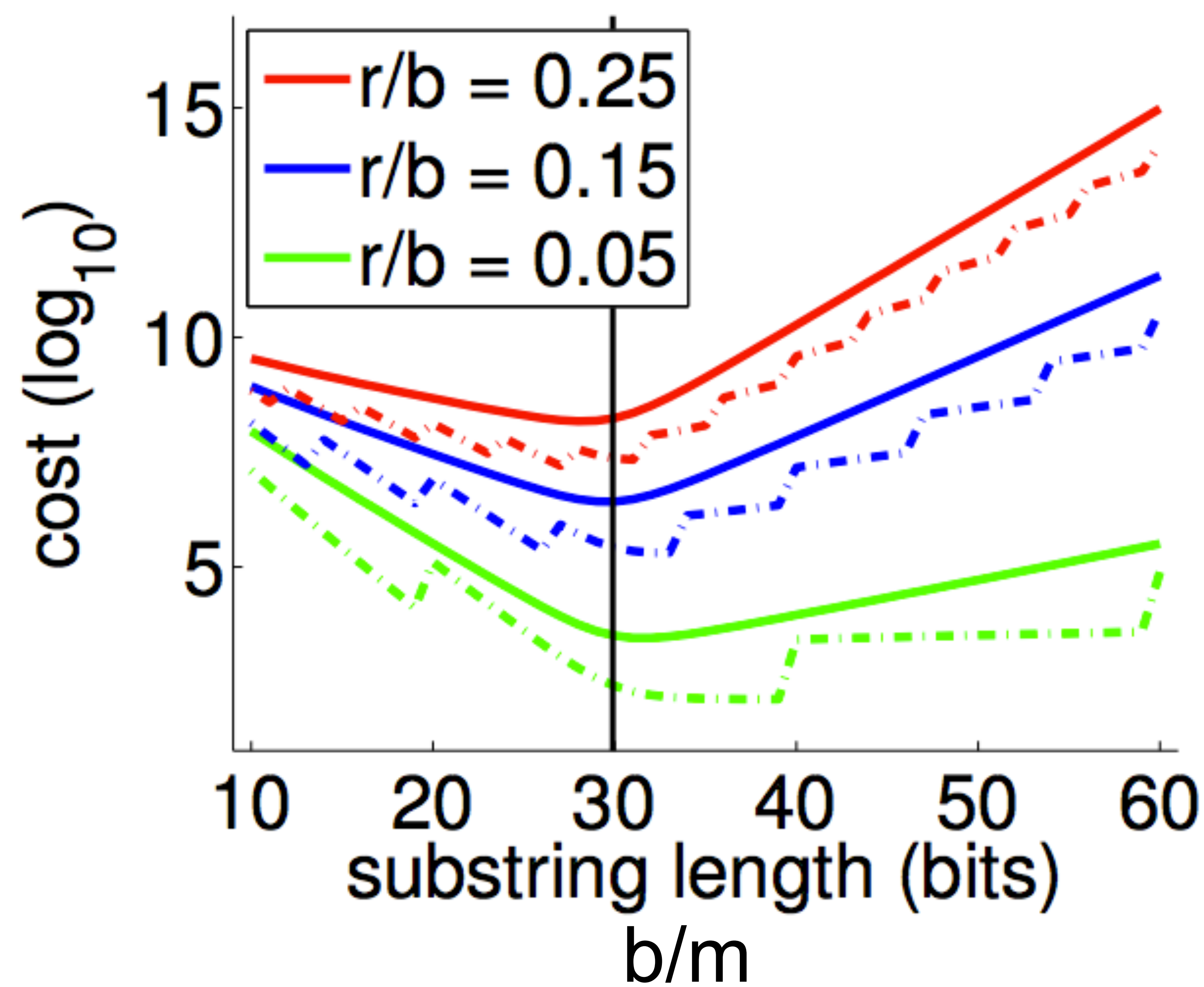
- **Storage cost:**
  - ***bn* bits to represent all descriptors in hash table**
  - ***m* hash tables referring to these descriptors ($mn\lg_2 n$)**
  - **In practice, optimal $m=b/\lg_2 n$ so overall storage cost near linear in *n* (see next slide)**

How to choose *m*?

- Trade-off between having large substrings (tight candidate set, but many bucket lookups in substring searches) and having small substrings (cheap substring search, but very loose candidate set)
  - Consider $m=b$, substrings are of length 1, but all descriptors are in candidate set!

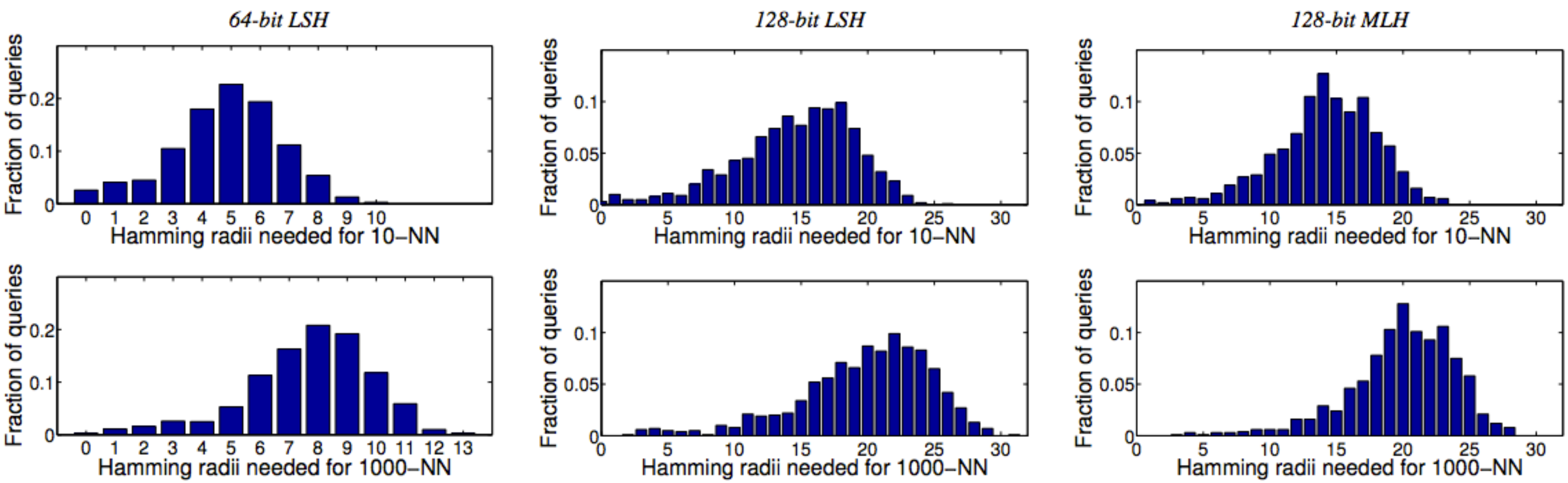
Figure at right:

- Database size: 1B descriptors
- 128-bit codes (b=128)



How to determine *r* from *k*?

- **Algorithm finds all database elements within Hamming distance *r*, but we often want *k* nearest neighbors to a query (not all elements within a fixed distance)**
- **Problem: binary codes not uniformly distributed across Hamming space, so cannot just pick an *r* corresponding to *k* (*r* required to contain k-nn depends on query)**
- **Solution: progressively increase *r* until k-NN are found.**



Summary

- Finding matches (between descriptors, between images) involves high-dimensional nearest neighbor search

- Wide range of techniques, approximations used heavily
  - Approximate k-NN retrieval
  - Approximate to descriptor values (compact features, visual words, bitcodes, etc.)

- Note: search was a common problem in rendering in the find half of the course as well
  - Rasterization: find the samples covered by a triangle
  - Ray casting: find [closest] triangle hit by ray
  - Lower dimensional problems lead to different acceleration structures and techniques