Large-scale visual recognition
Efficient matching

Florent Perronnin, XRCE
Hervé Jégou, INRIA

CVPR tutorial
June 16, 2012

Outline

- **Preliminary**

- Locality Sensitive Hashing: the two modes
  - Hashing
  - Embedding

- Searching with Product Quantization

Finding neighbors

- Nearest neighbor search is a critical step in object recognition
  - To compute the image descriptor itself
    E.g., assignment with k-means to a large vocabulary
  - To find the most similar images/patches in a database
  - For instance, the closest one w.r.t to Euclidean distance:

$$\mathrm{NN}(x) = \arg\min_{y \in \mathcal{Y}} ||x - y||^2$$

- Problems:
  - costly operation of exact exhaustive search: O(n*d)
  - High-dimensional vectors: for exact search the best approach is the naïve exhaustive comparison

The cost of (efficient) exact matching

- But what about the actual timings ? With an efficient implementation!

- Finding the 10-NN of 1000 distinct queries in 1 million vectors
  - Assuming 128-D Euclidean descriptors
  - i.e., 1 billion distances, computed on a 8-core machine

Poll: How much time?

The cost of (efficient) exact matching

- But what about the actual timings ? With an efficient implementation!

- Finding the 10-NN of 1000 distinct queries in 1 million vectors
  - Assuming 128-D Euclidean descriptors
  - i.e., 1 billion distances, computed on a 8-core machine

  5.5 seconds

- Assigning 2000 SIFTs to a visual vocabulary of size k=100,000
  - 1.2 second

- Hamming distance: 1000 queries, 1M database vectors
  - Computing the 1 billion distances: 0.6 second

Need for approximate nearest neighbors

- 1 million images, 1000 descriptors per image
  - 1 billion distances per local descriptor
  - 10^{12} distances in total
  - 1 hour 30 minutes to perform the query for Euclidean vectors

- To improve the scalability:
  - We allow to find the nearest neighbors in probability only:
    Approximate nearest neighbor (ANN) search

- Three (contradictory) performance criteria for ANN schemes
  - search quality (retrieved vectors are actual nearest neighbors)
  - speed
  - memory usage

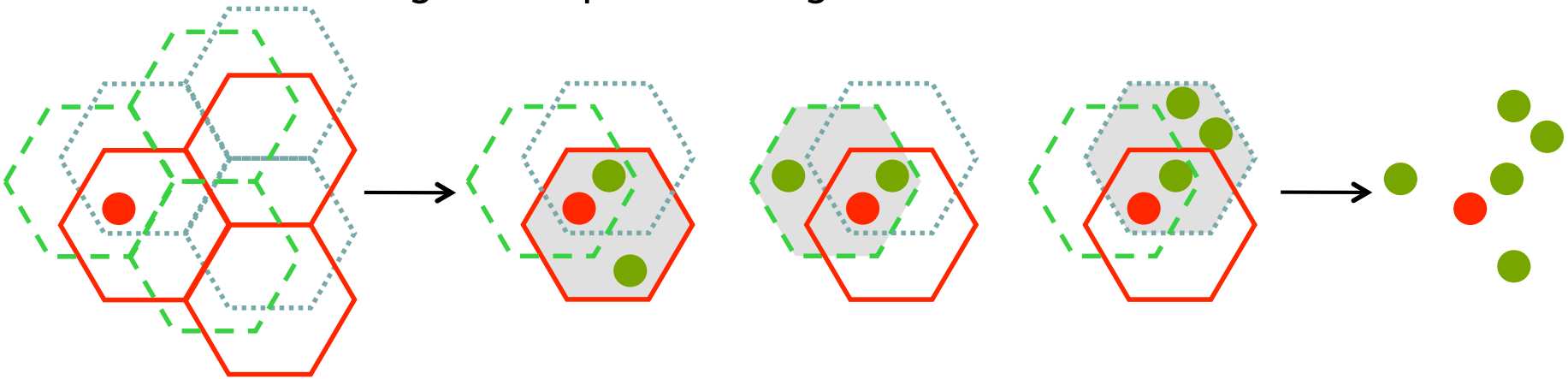
Outline

- Preliminary
- **Locality Sensitive Hashing: the two modes**
  - Hashing
  - Embedding
- Searching with Product Quantization

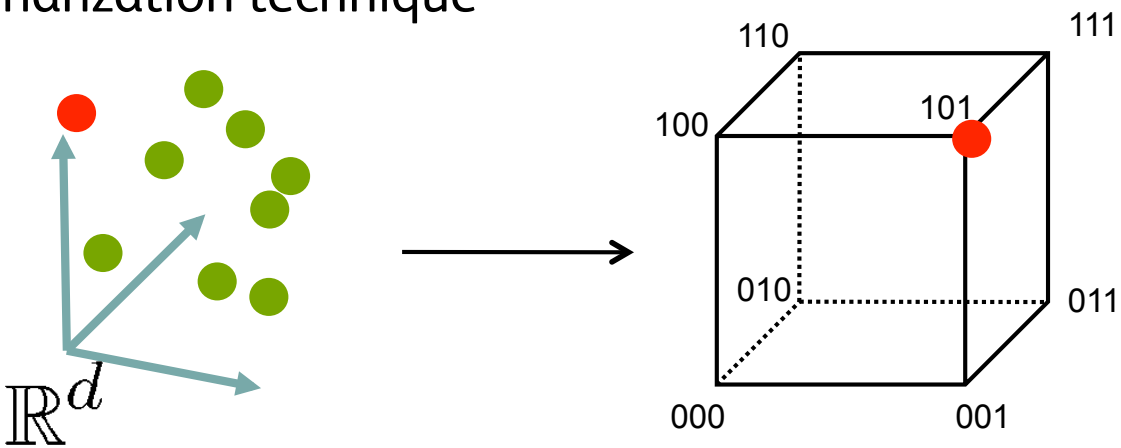
Locality Sensitive Hashing (LSH)

- Most known ANN technique [Charikar 98, Gionis 99, Datar 04,…]

- But "LSH" is associated with two distinct search algorithms
  - As an indexing technique involving several hash functions



  - As a binarization technique

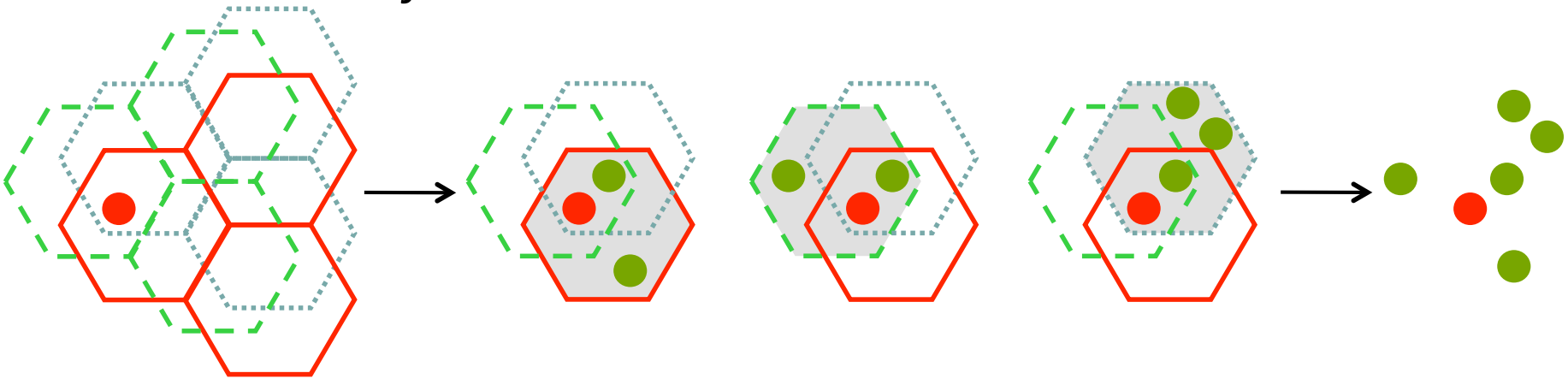


Outline

- Preliminary
- Locality Sensitive Hashing: the two modes
  - **Hashing**
  - Embedding
- Searching with Product Quantization

LSH – partitioning technique

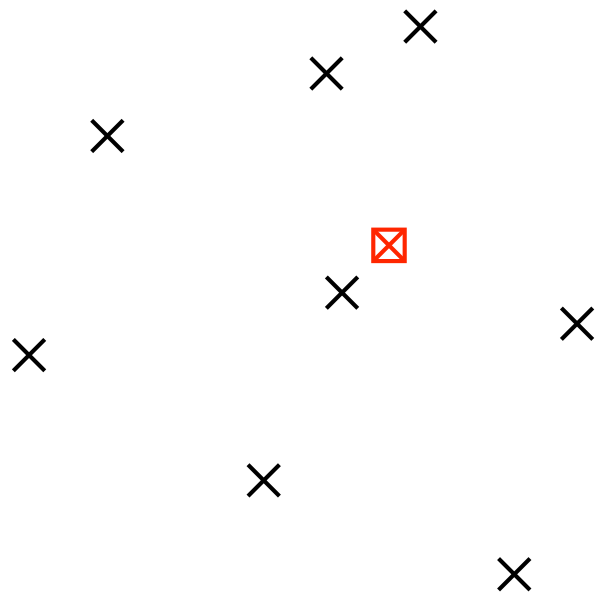
- General idea:
  - Define m hash functions in parallel
  - Each vector: associated with m distinct hash keys
  - Each hash key is associated with a hash table



- At query time:
  - Compute the hash keys associated with the query
  - For each hash function, retrieve all the database vectors assigned to the same key (for this hash function)
  - Compute the exact distance on this short-list

E2LSH: hash function for Euclidean vectors

[Datar 04]



1) Projection on i=1…m random directions

$$h_i(x) = \left\lfloor \frac{a_i^\top x - b_i}{w} \right\rfloor$$

2) Construction of l hash functions: concatenate m indexes h_i per hash function

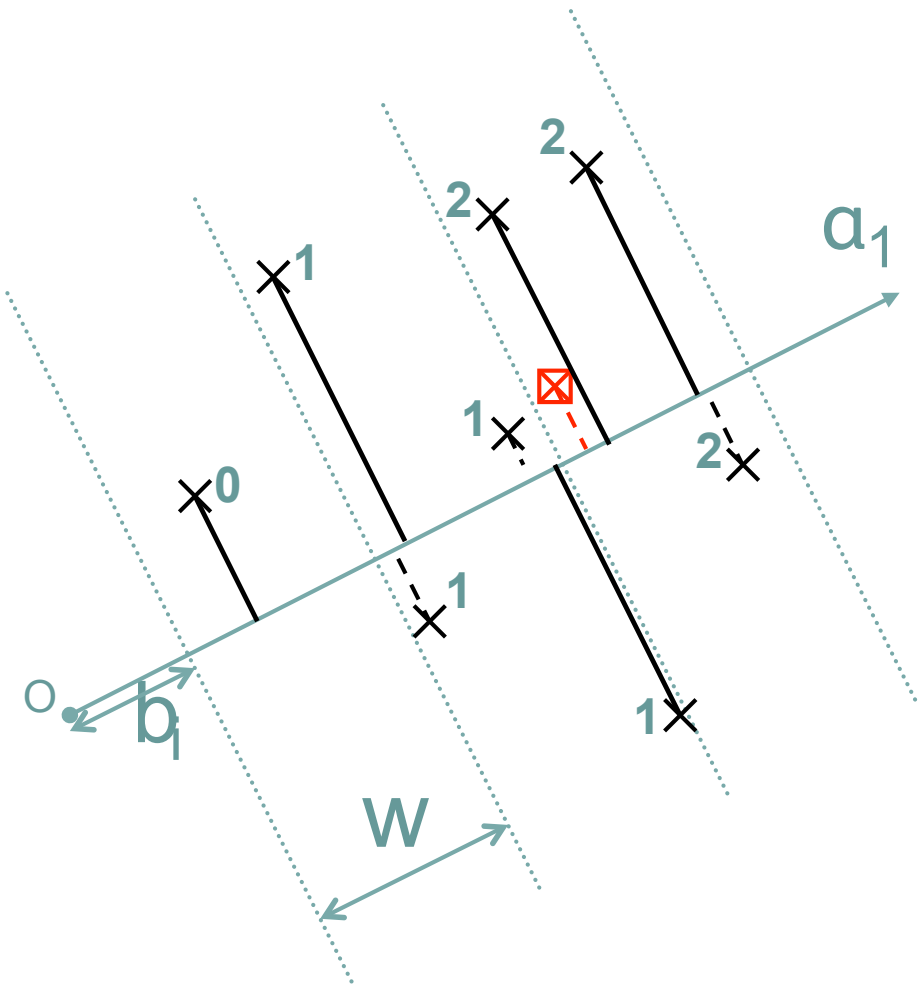
$$g_j(x) = \left(h_{\sigma_1(i)(x)}, \ldots, h_{\sigma_m}(x)\right)$$

3) For each g_j, compute two hash values universal hash functions: $u_1(.)$, $u_2(.)$

4) store the vector id in a hash table, as for an inverted file

E2LSH: hash function for Euclidean vectors

[Datar 04]



1) Projection on i=1…m random directions

$$h_i(x) = \left\lfloor \frac{a_i^\top x - b_i}{w} \right\rfloor$$

2) Construction of l hash functions: concatenate m indexes h_i per hash function

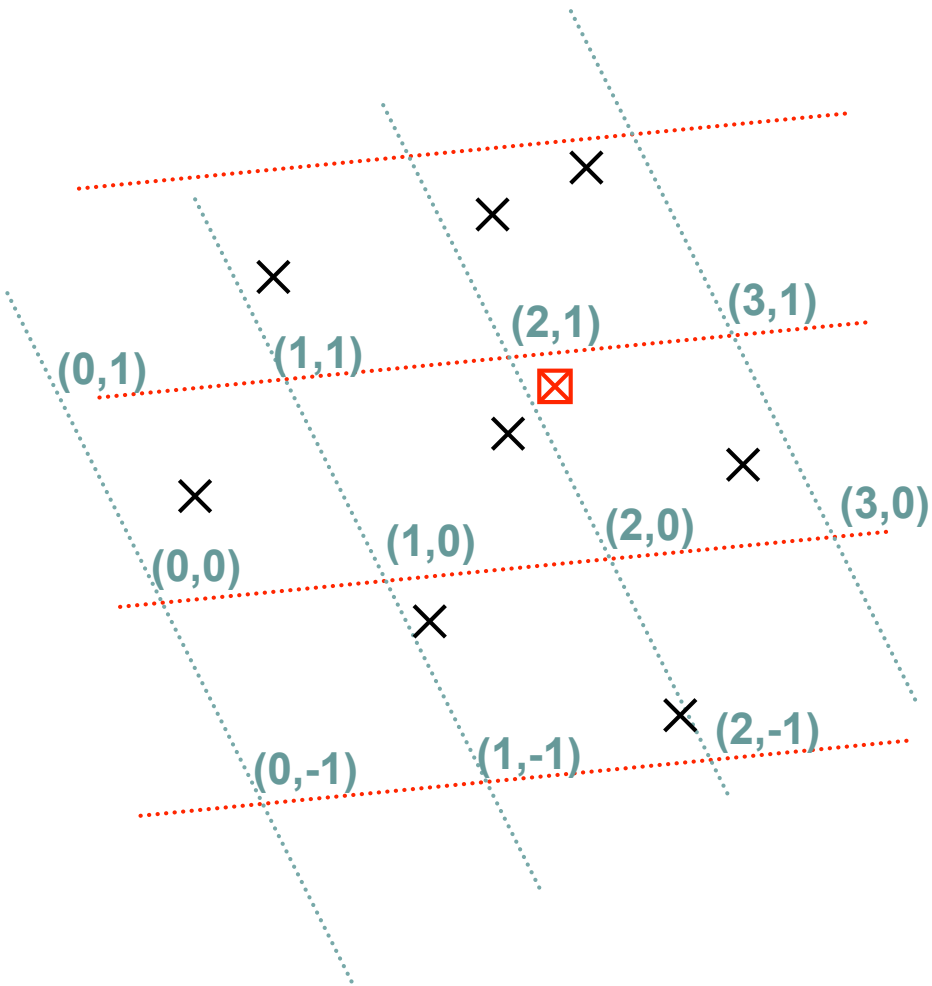
$$g_j(x) = \left(h_{\sigma_1(i)(x)}, \ldots, h_{\sigma_m}(x)\right)$$

3) For each g_j, compute two hash values universal hash functions: $u_1(.)$, $u_2(.)$

4) store the vector id in a hash table, as for an inverted file

E2LSH: hash function for Euclidean vectors

[Datar 04]



1) Projection on i=1…m random directions

$$h_i(x) = \left\lfloor \frac{a_i^\top x - b_i}{w} \right\rfloor$$

2) Construction of l hash functions: concatenate m indexes h_i per hash function

$$g_j(x) = \left(h_{\sigma_1(i)(x)}, \ldots, h_{\sigma_m}(x)\right)$$

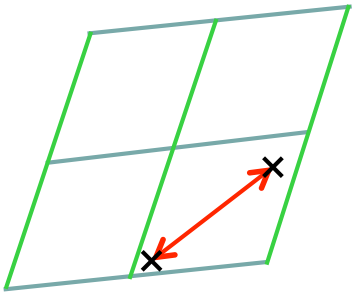
3) For each g_j, compute two hash values universal hash functions: $u_1(.)$, $u_2(.)$

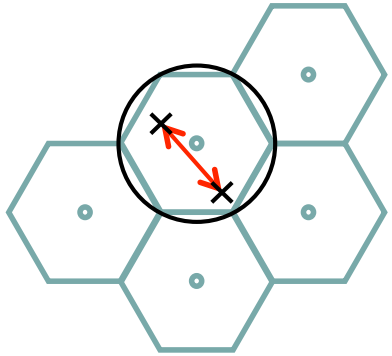
4) store the vector id in a hash table, as for an inverted file

Alternative hash functions

- Instead of using random projections
- Why not directly using a structured quantizer?
  - Vector quantizers: better compression performance than scalar ones

- Structured vector quantizer: Lattice quantizers [Andoni 06]
  - Hexagonal (d=2), E_8 (d=8), Leech (d=24)





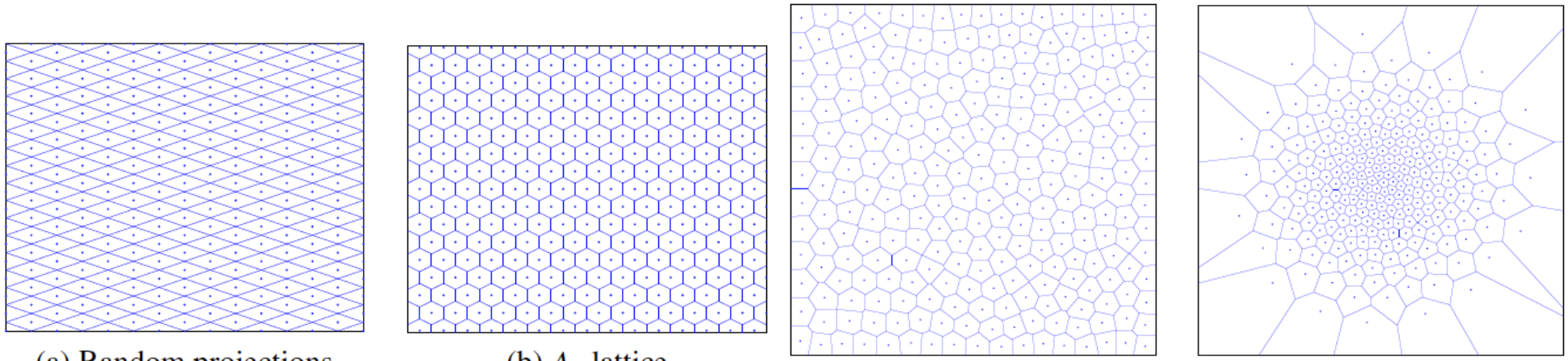
- But still: **lack of adaptation to the data**

Alternative hash functions – Learned

- Any hash function can be used in LSH
  - Just need a set of functions $f_j : \mathbb{R}^d \rightarrow \mathbb{K}$
  - Therefore, could be learned on sample examples

- In particular: k-means, Hierarchical k-means, KD-trees

From [Pauleve 10]

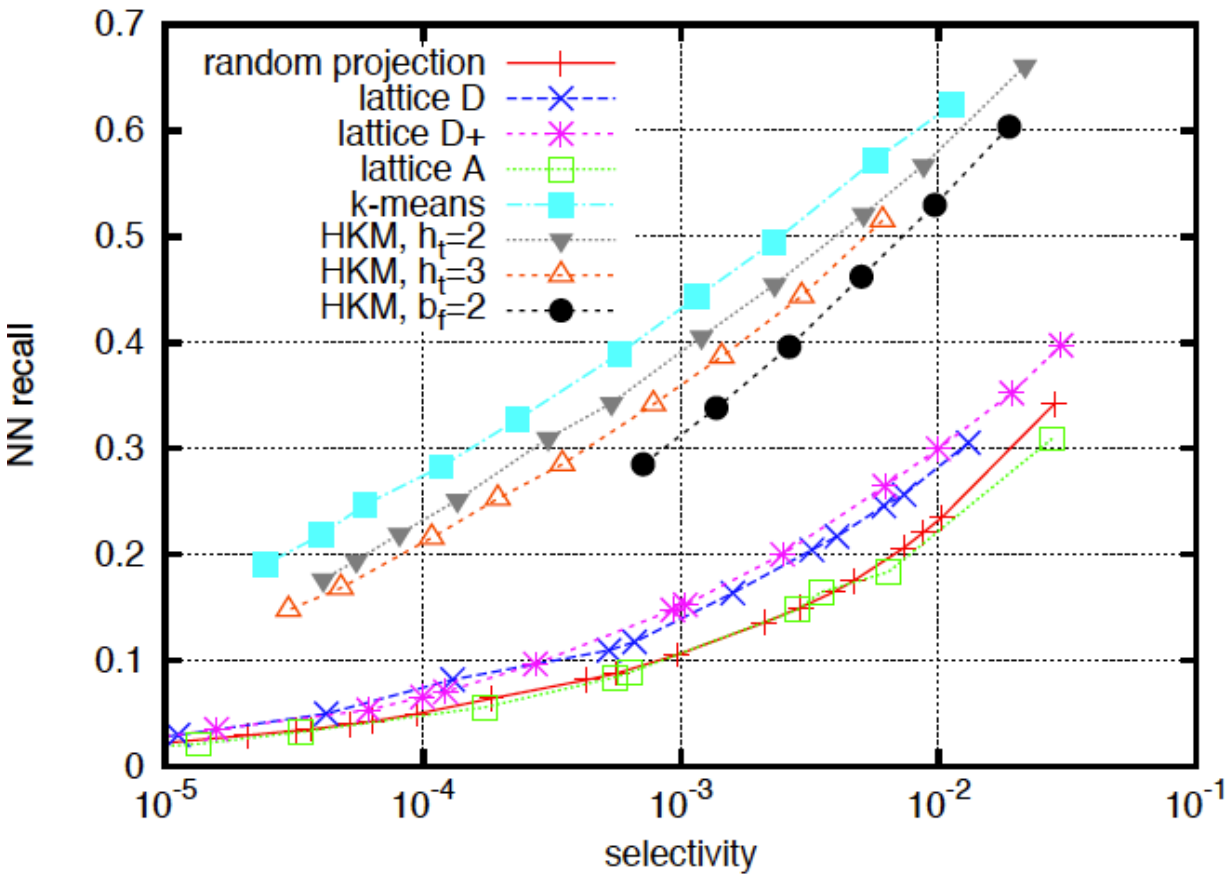


(a) Random projections (b) A_2 lattice (c) k-means Uniform distribution (d) k-means Gaussian distribution

- Better data adaptation than with structured quantizers

Alternative hash functions – Learned

- Significantly better than structured quantizers
- Example of search: quality for a **single** hash function



- BOV: k-means!
- HKM: loss compared with k-means [Nister 06]

Multi-probe LSH

- But multiple hash functions use a lot of memory
  - Per vector and per hash table: at least an id

- Multi-probe LSH [Lv 07]
  - Use less hash functions (possibly 1)
  - But probe several (closest) cells per hash function

    ⇨ save a lot of memory
  - Similar in spirit to Multiple-assignment with BOV

FLANN

- ANN package described in Muja's VISAPP paper [Muja 09]
  - Multiple kd-tree or k-means tree
  - With auto-tuning under given constraints

    Remark: self-tuned LSH proposed in [Dong 07]
  - Still high memory requirement for large vector sets

- Excellent package: high integration quality and interface!

- See http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN

FLANN - Fast Library for Approximate Nearest Neighbors

What is FLANN?

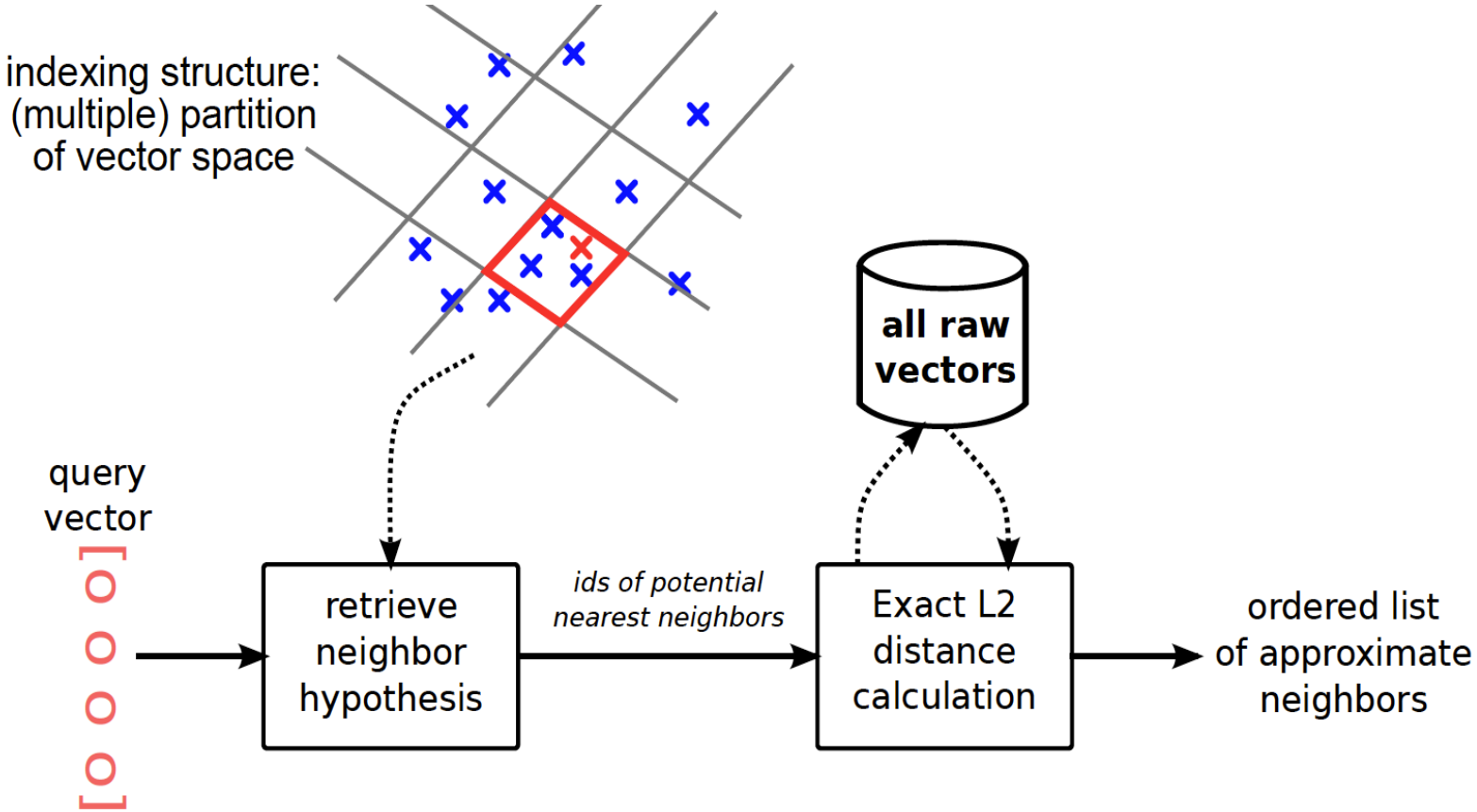
FLANN is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset.

FLANN is written in C++ and contains bindings for the following languages: C, MATLAB and Python.

News

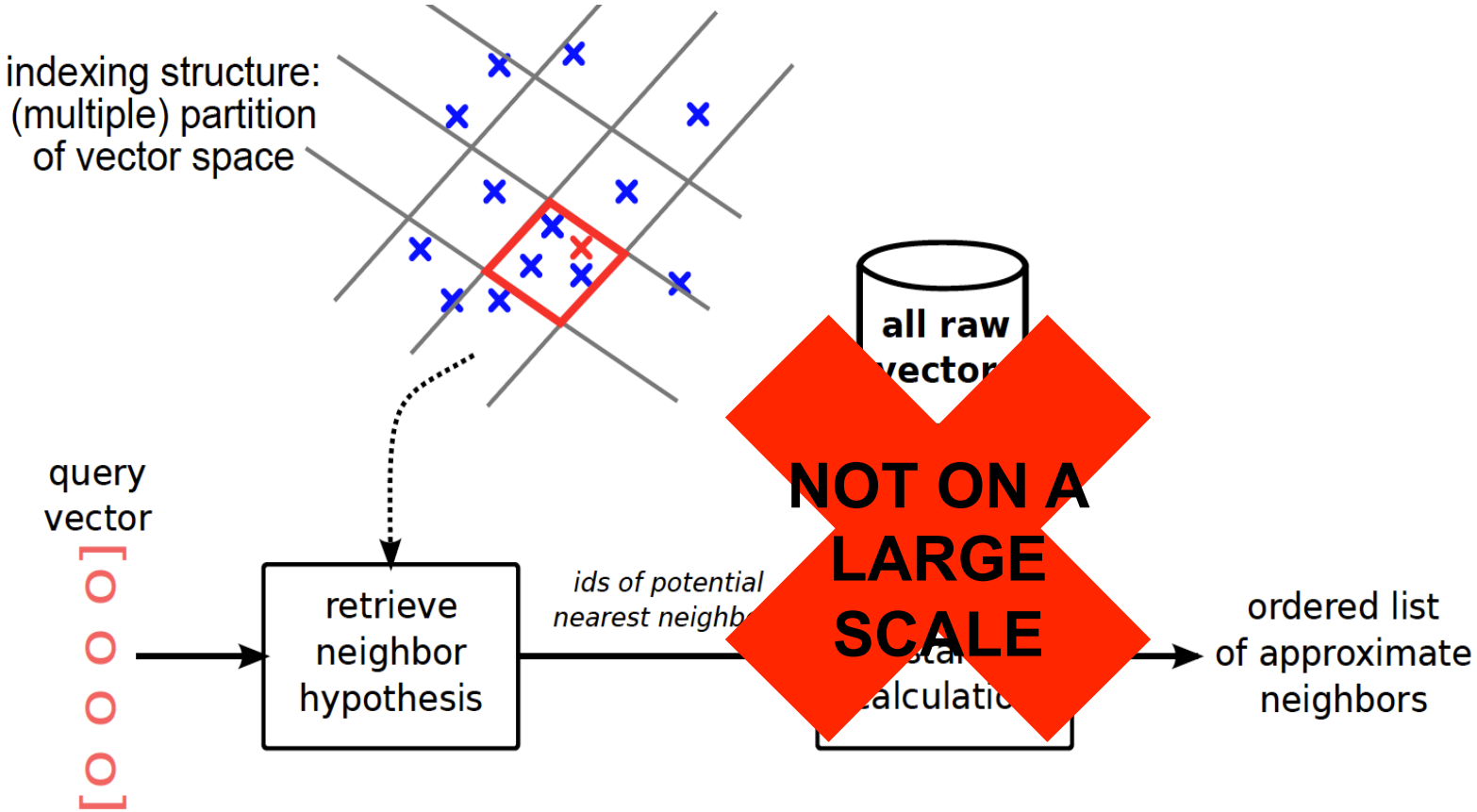
- (20 December 2011) Version 1.7.0 is out bringing two new index types and several other improvements.
- You can find binary installers for FLANN on the Point Cloud Library project page. Thanks to the PCL developers!
- Mac OS X users can install flann though MacPorts (thanks to Mark Moll for maintaining the Portfile)
- New release introducing an easier way to use custom distances, kd-tree implementation optimized for low dimensionality search and experimental MPI support
- New release introducing new C++ templated API, thread-safe search, save/load of indexes and more.
- The FLANN license was changed from LGPL to BSD.

Issue for large scale: final verification



- For this second ("re-ranking") stage, we need raw descriptors, i.e.,
  - either huge amount of memory → 128GB for 1 billion SIFTs
  - either to perform disk accesses → severely impacts efficiency

Issue for large scale: final verification



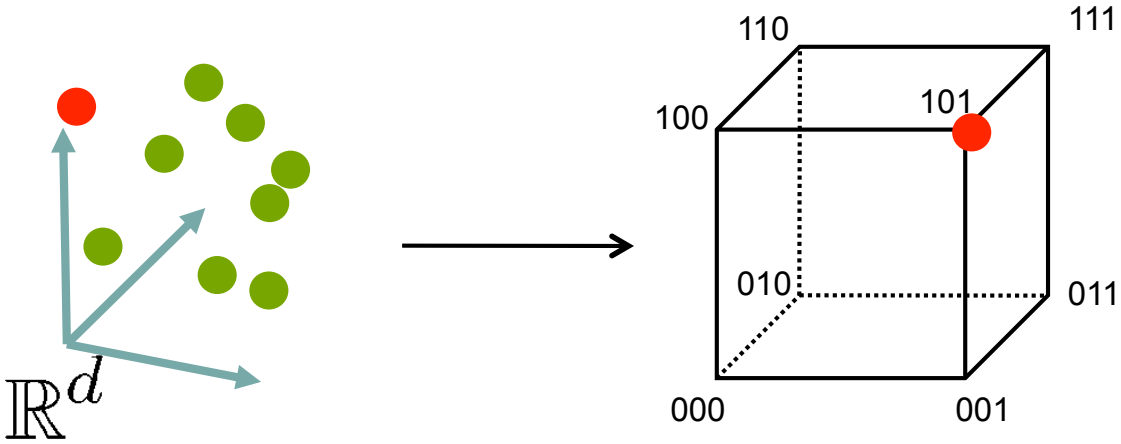
- Some techniques –like BOV– keep all vectors (no verification)

- Better: use very short codes for the filtering stage
  - Hamming Embedding [Jegou 08] or Product Quantization [Jegou 11]

Outline

- Preliminary
- Locality Sensitive Hashing: the two modes
  - Hashing
  - **Embedding**
- Searching with Product Quantization

LSH for binarization [Charikar' 98, J.'08, Weiss'09, etc]



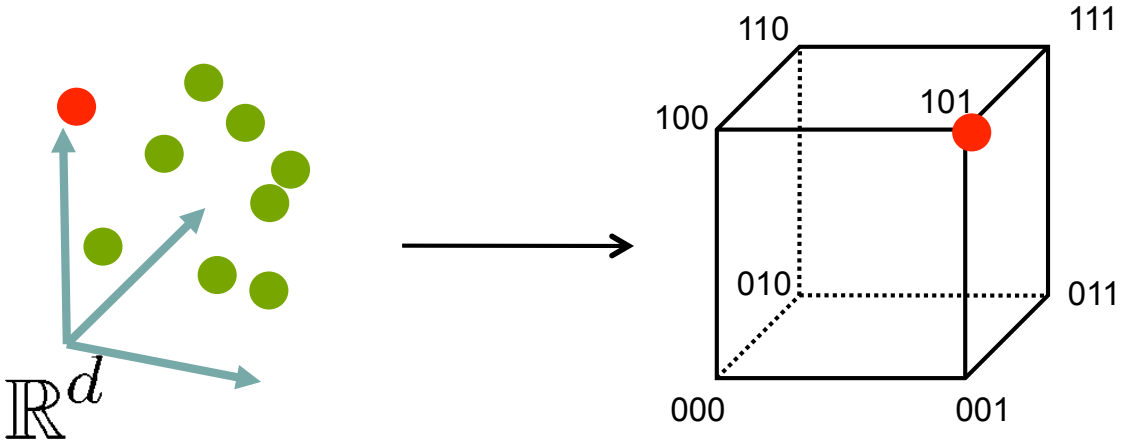
- Idea: design/learn a function mapping the original space into the compact Hamming space:

$$e : \mathbb{R}^d \rightarrow \{0,1\}^D$$
$$x \rightarrow e(x)$$

- Objective: neighborhood in the Hamming space try to reflect original neighborhood $\arg\min_i h\left(e(x), e(y_i)\right) \approx \arg\min_i d(x,y)$

- Advantages: compact descriptor, fast comparison

LSH for binarization [Charikar' 98, J.'08, Weiss'09, etc]



- Given B random projection direction a_i
- Compute a binary code from a vector x as

$$b_i(x) = \text{sign } a_i^\top x$$

$$b(x) = (b_1(x), \ldots, b_B(x))$$

- Spectral Hashing: theoretical framework for finding hash functions
- In practice: PCA + binarization on the different axis (based on variance)

LSH: the two modes – approximate guidelines

Partitioning technique

- **Sublinear** search
- Several hash indexes (integer)
- **Large memory overhead**
  - Hash table overhead (store ids)
- Need original vectors for re-ranking
  - **Need a lot of memory**
  - **Or to access the disk**
- Interesting when (e.g., FLANN)
  - Not too large dimensionality
  - Dataset small enough (memory)
- Very good variants/software (FLANN)

Binarization technique

- **Linear** search
- Produce a binary code per vector
- **Very compact**
  - bit-vectors, concatenated (no ids)
- **Very fast comparison**
  - Hamming distance (popcnt SSE4)
  - 1 billion comparisons/second
- Interesting
  - For **very high-dimensional** vectors
  - When memory is critical
- Simple to implement. Very active problems with many variants

Other topics on LSH

- More general metrics: Kernelized LSH [Kulis 09], RMMH [Joly 11], …

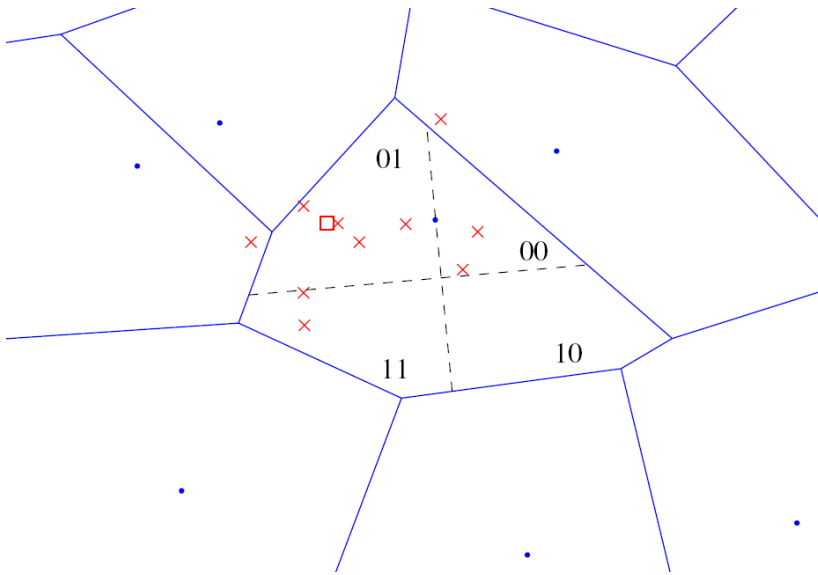
- Binary LSH (=searching binary vectors)
  - Like E2LSH but: random subset of bits instead of projections
  - In this CVPR: [Nourouzi 12]. **Exact** binary search variant!

- Optimized jointly with dimensionality reduction
  - PCA+random rotation or PCA+variance balancing [Jegou 10]
  - ITQ [Gong 11]

- Asymmetric distances
  - Idea: do **not** approximate the query – only the database is binarized
  - Proposed with sketches [Dong 08]
  - Significant improvement for any binarization technique [Gordo 11]

Hamming Embedding

- Introduced as an extension of BOV [Jegou 08]



- Combination of
  - A partitioning technique (k-means)
  - A binary code that refine the descriptor

Representation of a descriptor x

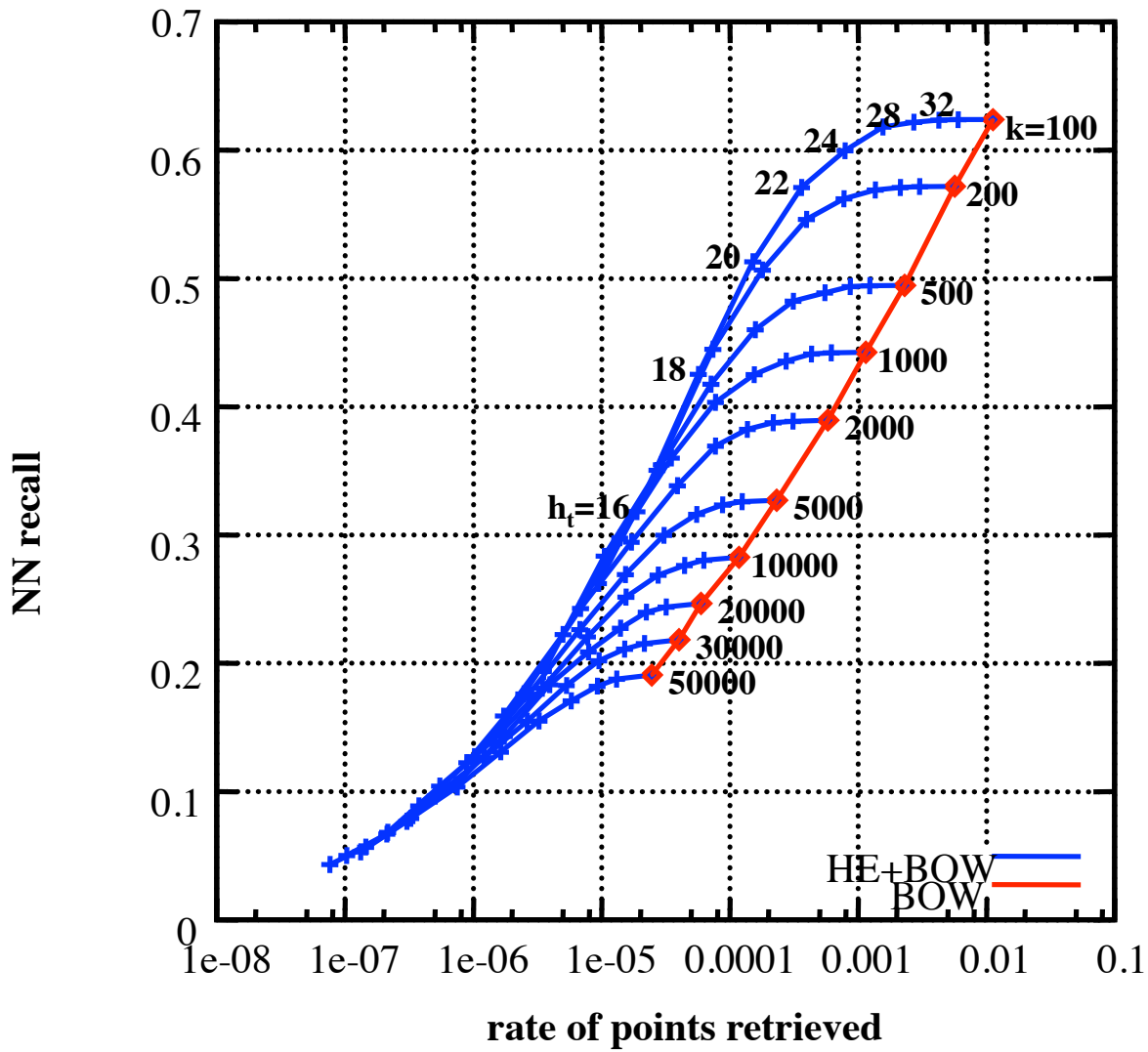
- Vector-quantized to q(x) as in standard BOV
- \+ short binary vector b(x) for an additional localization in the Voronoi cell

- Two descriptors x and y match iif

$$f_{\mathrm{HE}}(x,y) = \begin{cases} (\text{tf-idf}(q(x)))^2 & \text{if } q(x) = q(y) \\ & \quad \text{and } h\left(b(x), b(y)\right) \leq h_t \\ 0 & \text{otherwise} \end{cases}$$

Where h(.,.) denotes the Hamming distance

ANN evaluation of Hamming Embedding



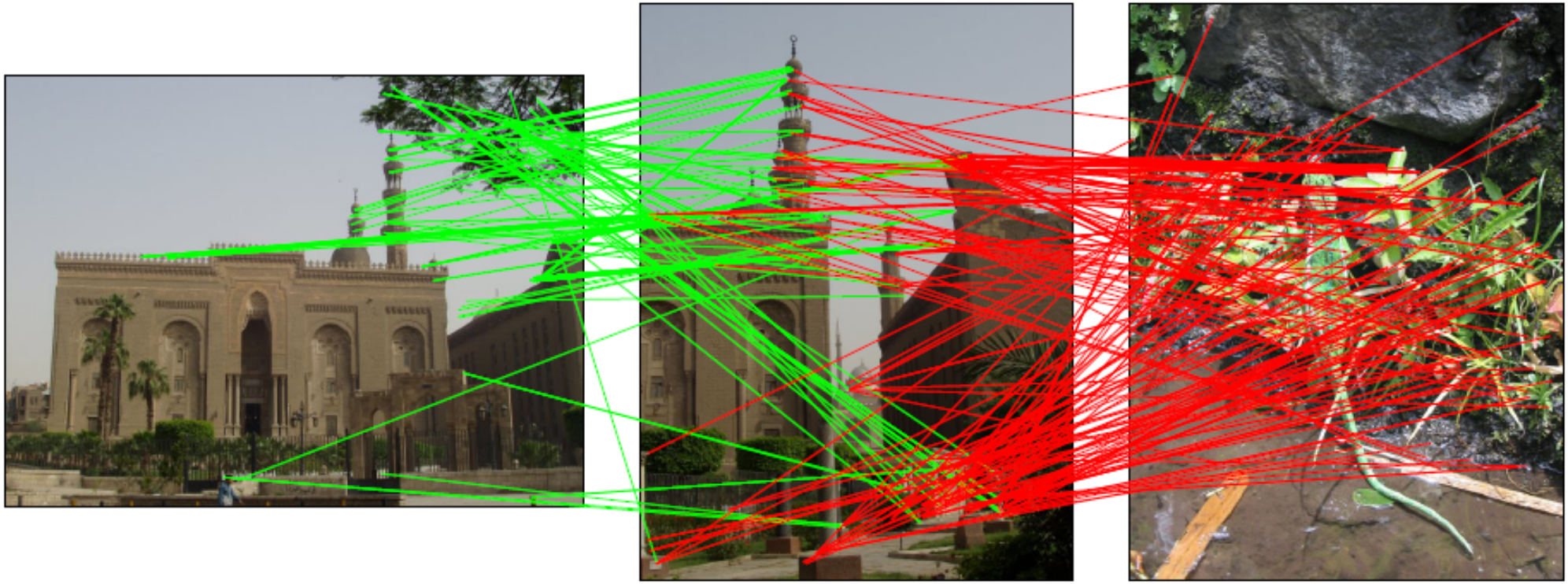
compared to BOW: at least 10 times less points in the short-list for the same level of accuracy

Hamming Embedding provides a much better trade-off between recall and remove false positives

Matching points - 20k word vocabulary

201 matches

240 matches



Many matches with the non-corresponding image!

Matching points - 200k word vocabulary

69 matches

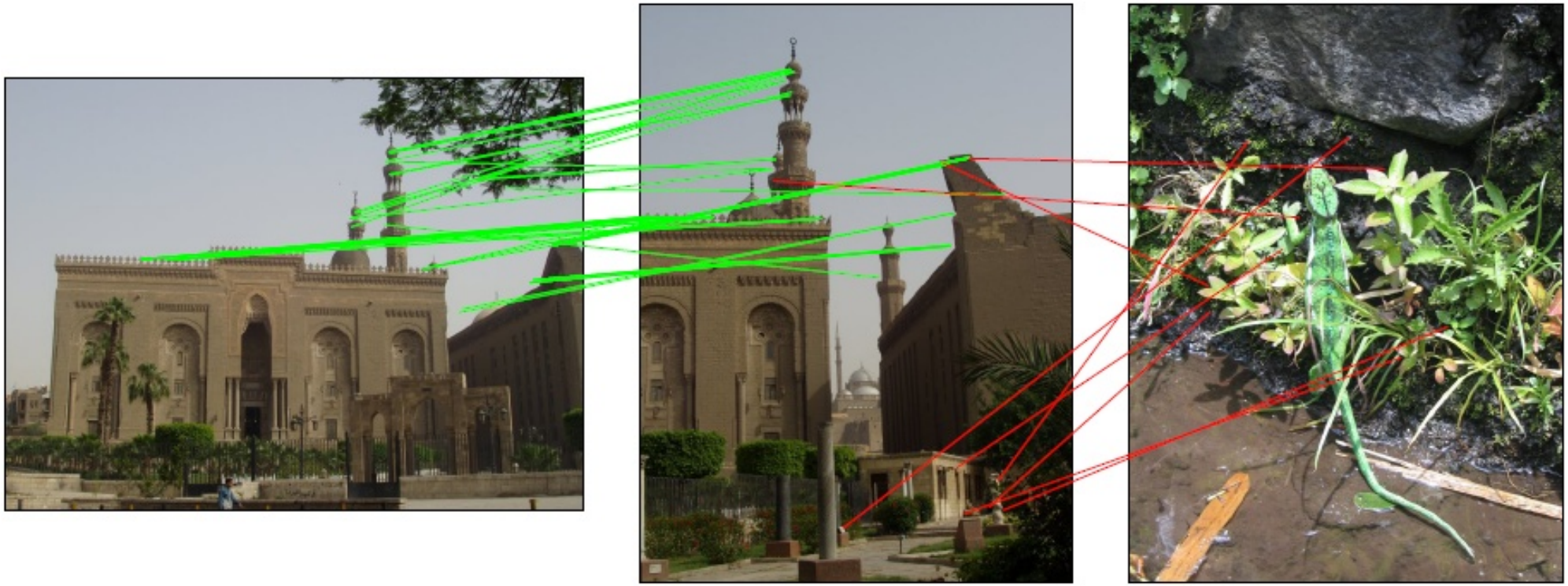
35 matches

Still many matches with the non-corresponding one

Matching points - 20k word vocabulary + HE

83 matches

8 matches

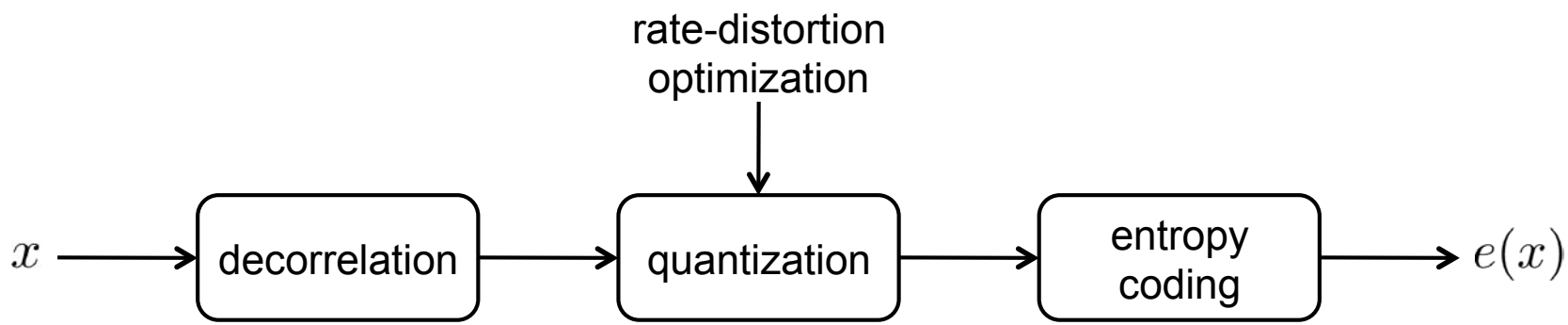


10x more matches with the corresponding image!

Outline

- Preliminary
- Locality Sensitive Hashing: the two modes
  - Hashing
  - Embedding
- **Searching with Product Quantization**

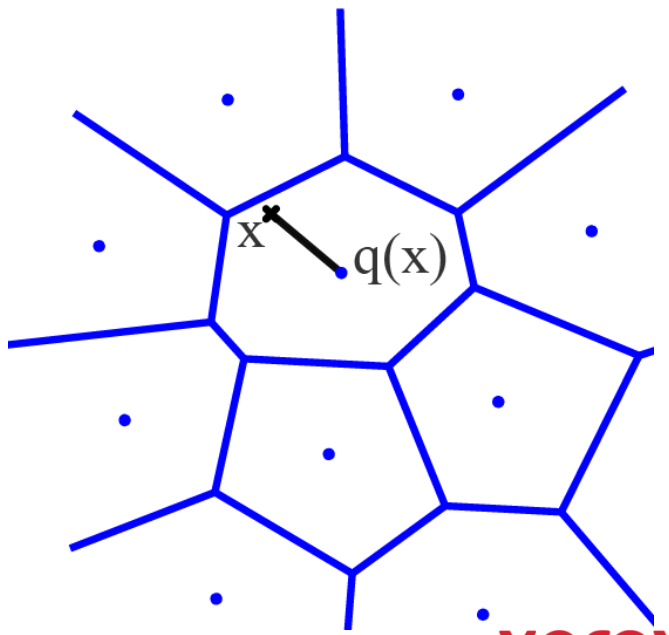
A typical source coding system



- Simple source coding system:
  - Decorrelation, e.g., PCA
  - Quantization
  - Entropy coding

- To a code e(x) is associated a unique reconstruction value q(x)
  ⇨ i.e., the visual word

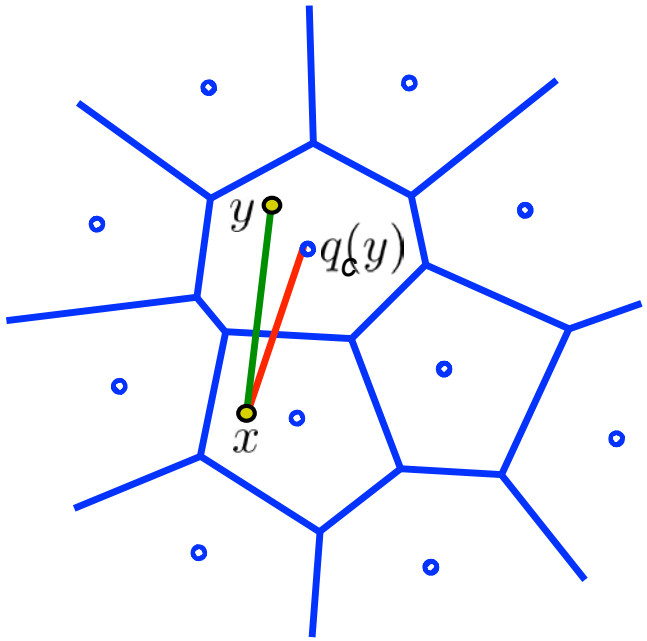
- Focus on quantization (lossy step)



Relationship between Reconstruction and Distance estimation

- Assume y quantized to $q_c(y)$
  x is a query vector

- If we estimate the distance by
  $$d(x, y) \approx d(x, q_c(y))$$



- Then we can show that:

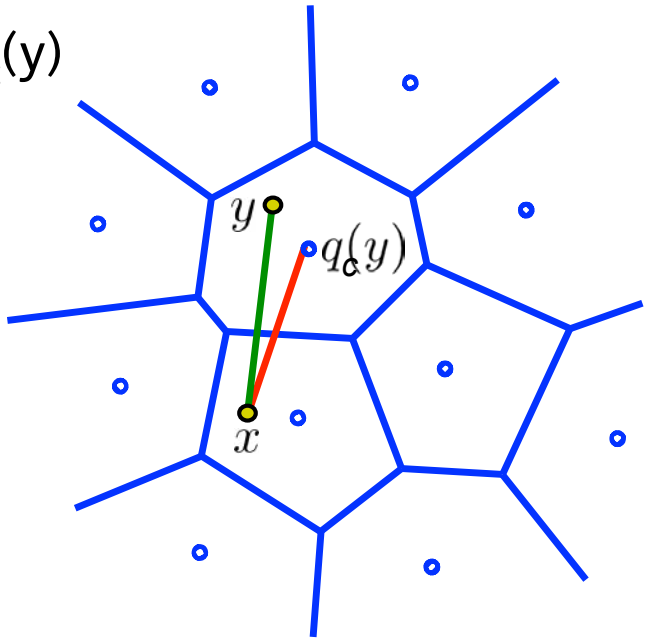
$$\mathbb{E}_Y[(d(x,y) - d(x,q_c(y)))^2] \leq \mathbb{E}_Y[(y - q_c(y))^2] = \mathrm{MSE}$$

i.e., the error on the square distance is statistically bounded by the quantization error

Searching with quantization [Jegou 11]

- Main idea: compressed representation of the database vectors
  - Each database vector y is represented by $q_c(y)$ where $q_c(.)$ is a **product quantizer**

$$d(x, y) \approx d(x, q_c(y))$$



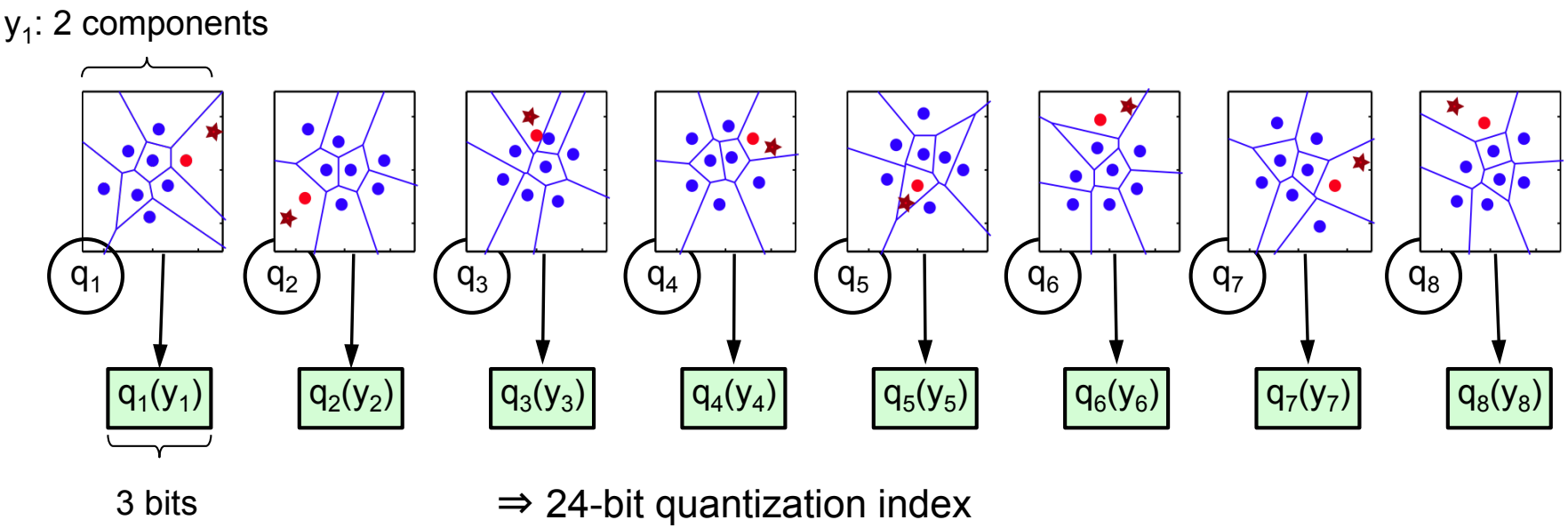
- Search = distance approximation problem

- **The key**: Estimate the distances **in the compressed domain** such that
  - Quantization is fast enough
  - Quantization is precise, i.e., many different possible indexes (ex: 2^{64})

- Regular k-means is not appropriate: not for $k=2^{64}$ centroids

Product Quantizer

- Vector split into m subvectors: $y \rightarrow [y_1 | \ldots | y_m]$
- Subvectors are quantized separately
- Example: y = 16-dim vector split in 8 subvectors of dimension 16



- In practice: 8 bits/subquantizer (256 centroids),
  - SIFT: m=4-16
  - VLAD/Fisher: 4-128 bytes per indexed vector

Asymmetric distance computation (ADC)

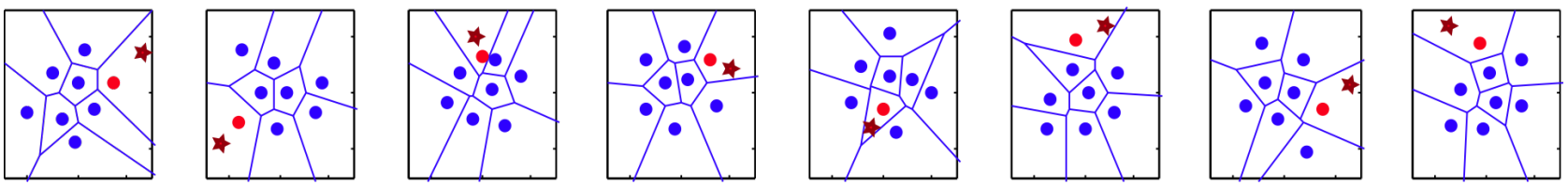
- Compute the square distance approximation in the compressed domain

$$d\left(x,y\right)^2 \approx \sum_{i=1}^{m} d\left(x_i, q_i(y_i)\right)^2$$

- To compute distance between query x and **many** codes
  - compute $d\left(x_i, c_{i,j}\right)^2$ for each subvector x_i and all possible centroids

  → stored in look-up tables

  → fixed cost for quantization
  - for each database code: sum the elementary square distances



- Each 8x8=64-bits code requires only **m=8 additions per distance**
- IVFADC: combination with an inverted file to avoid exhaustive search

Combination with an inverted file system

ALGORITHM

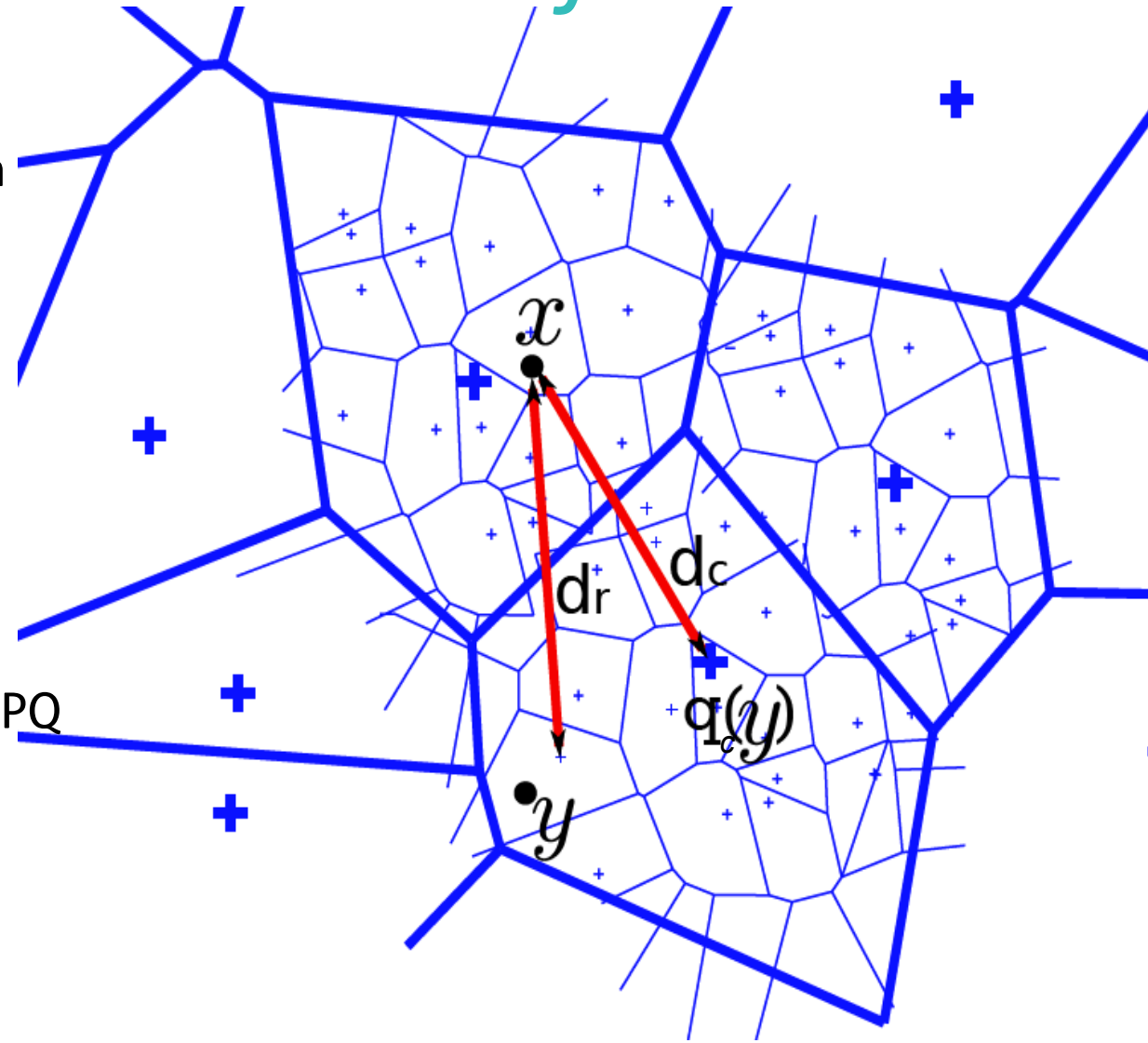
① Coarse k-means hash function

Select k' closest centroids c_i and corresponding cells

② Compute the **residual vector** $x-c_i$ of the query vector

③ Encode the residual vector by PQ

④ Apply the PQ search method. Distance is approximated by $d(x,y) = d(x-c_i, q(y-c_i))$

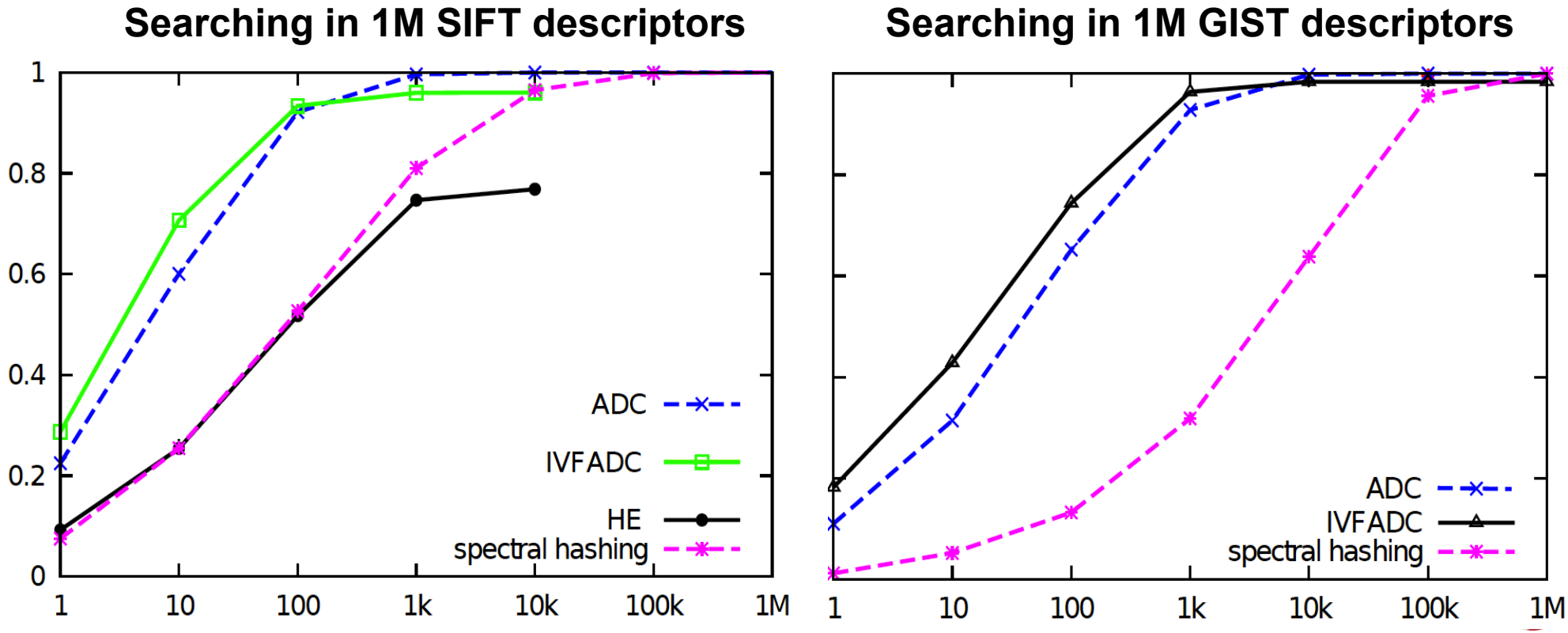


Example timing: 3.5 ms per vector for a search in 2 billion vectors

Performance evaluation

- Comparison with other memory efficient approximate neighbor search techniques, i.e., binarization techniques
  - Spectral Hashing [Weiss 09] – exhaustive search
  - Hamming Embedding [Jegou 08] – non exhaustive search

- Performance measured by searching 1M vector (recall@R, varying R)

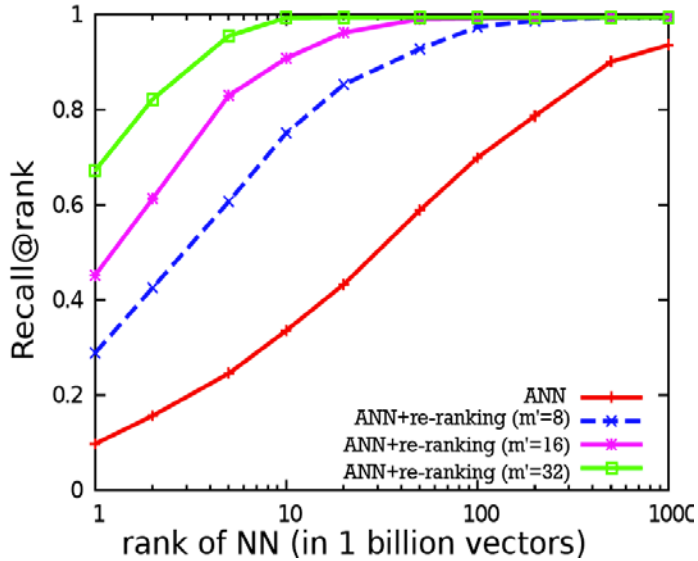


Variants

- Adapted codebook for residual vectors [Uchida 11]
  - Learn the product quantizer separately in the different cells

- Re-ranking with source coding [Jegou 11]
  - Exploit the explicit reconstruction of PQ
  - Refine the database vector by a short code

$$\hat{y} = \mathrm{q_c}(y) + \mathrm{q_r}(\mathrm{r}(y))$$



- In this CVPR:
  - The “multiple inverted index” [Babenko 12]
  - Replace the coarse k-means by a product quantizer
  - + priority selection of cells
  - Then apply PQ distance estimation

Product Quantization: some applications

- PQ search was first proposed for searching local descriptors [J'09-11], i.e., to replace bag-of-words or Hamming Embedding

- [J'10]: Encoding a global image representation (Vlad/Fisher)

- [Gammeter et al'10]: Fast geometrical re-ranking with local descriptors

- [Perronnin et al.'11]: Large scale classification (Imagenet)
  - Combined with Stochastic Gradient Descent SVM
  - Decompression on-the-fly when feeding the classifier
  - Won the ILSVRC competition in 2011

- [Vedaldi and Zisserman'12] – CVPR
  - Learning in the PQ-compressed domain
  - Typically *10 acceleration with no loss

Conclusion

Nearest neighbor search is a key component of image indexing systems

Product quantization-based approach offers

- Competitive search accuracy
- Compact footprint: few bytes per indexed vector

Tested on 220 million video frames, 100 million still images

extrapolation for 1 billion images: 20GB RAM, query < 1s on 8 cores

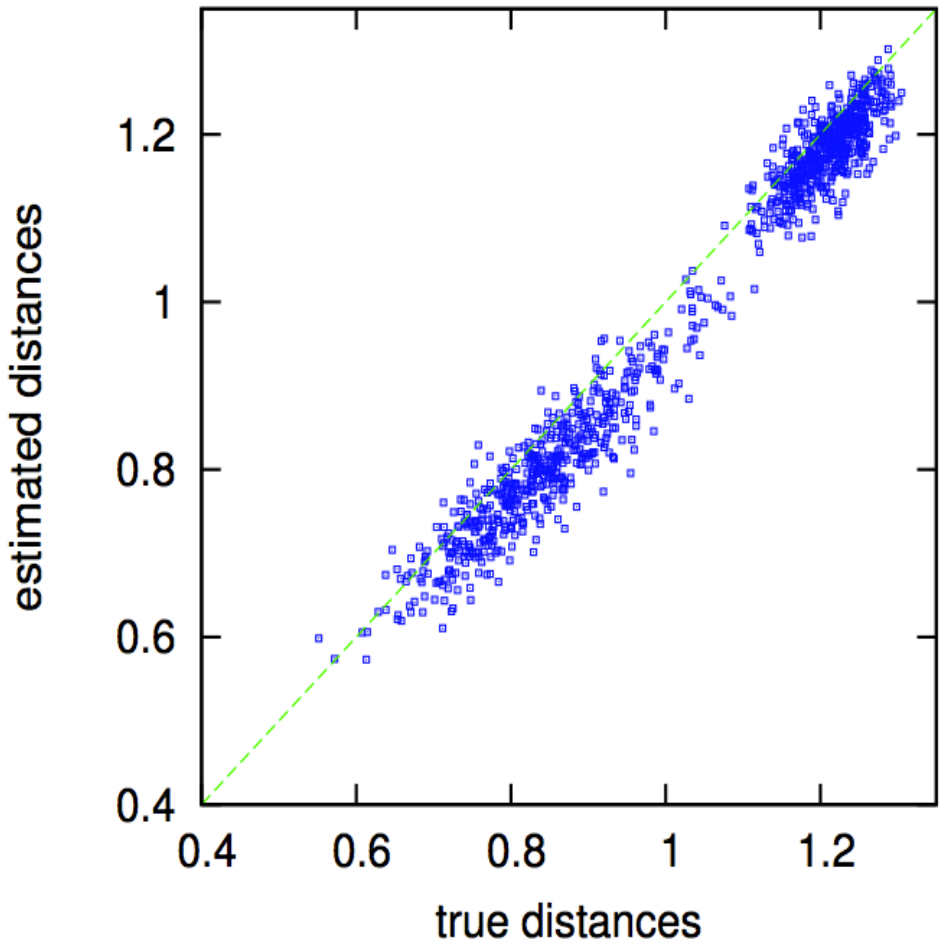
Tested on audio, text descriptors, etc

Toy Matlab package available on my web page

10 million images indexed on my laptop:

21 bytes per indexed image

Estimated distances versus true distances



- The estimator $d(X,q(Y))^2$ of $d(X,Y)^2$ is biased:
  - The bias can removed by quantization error terms
  - but does not improve the NN search quality