Approximate Nearest Neighbour Search

Partitioning

Distance evaluated only over a fraction of the data vectors

Sub-linear search
Large memory requirements
Distance evaluated on raw data (more memory or disk access)

LSH partitioning
k-d trees

Embedding

The distance is approximated

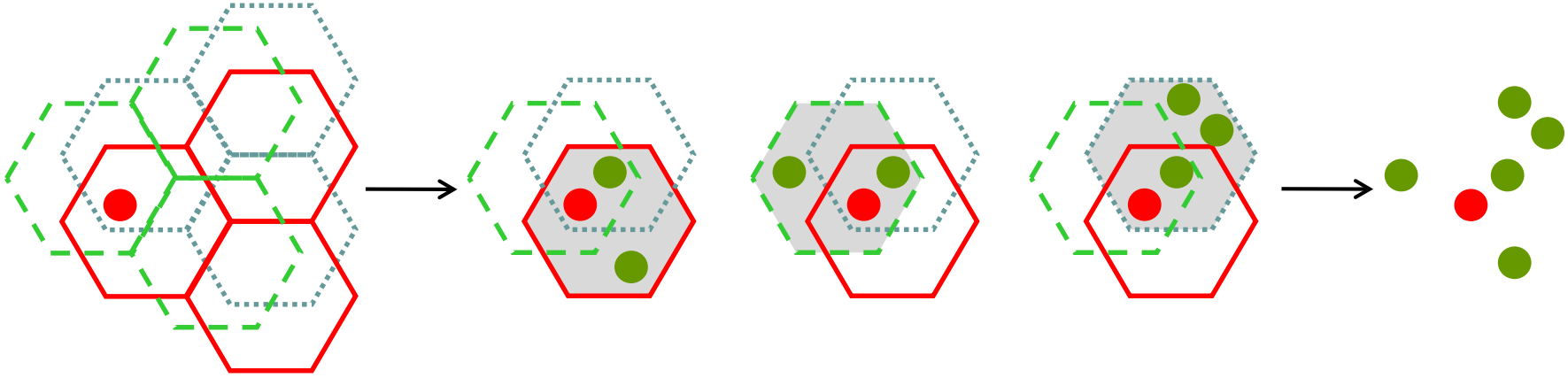
Linear search
Compact codes
Efficient evaluation of the (approx) distance

LSH binarization
Product Quantization
min-Hash

Partitioning

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

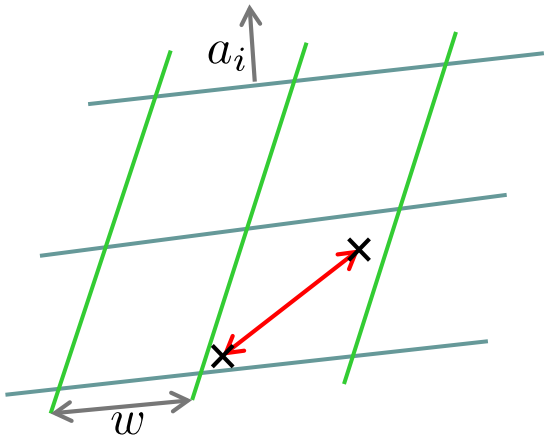


- At query time:
  - Compute the hash keys associated with the query
  - For each hash function, retrieve all the database vectors assigned to the same key (for this hash function)
  - Compute the exact distance on this short-list

Hash functions

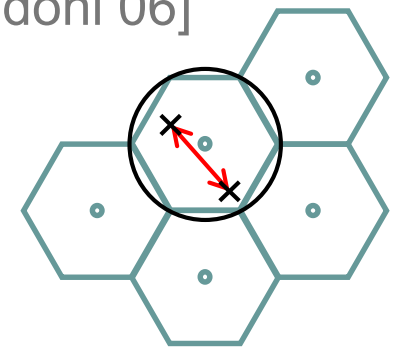
- Typical choice: use random projections

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



- 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)



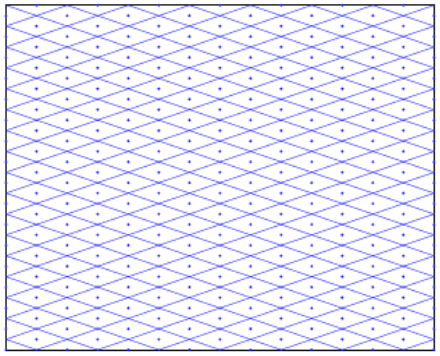
  - **Lack of adaptation to the data**

Alternative hash functions – Learned

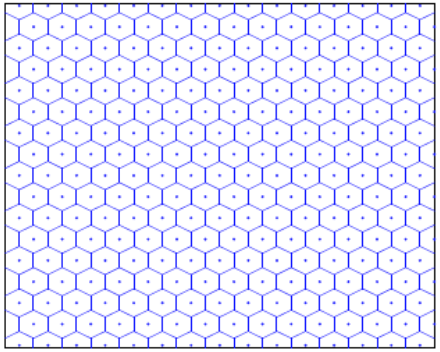
- Any hash function can be used in LSH/partitioning
  - Just need a set of functions $f_j : \mathbb{R}^d \rightarrow \mathbb{K}$
  - Therefore, could be learned on sample examples

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

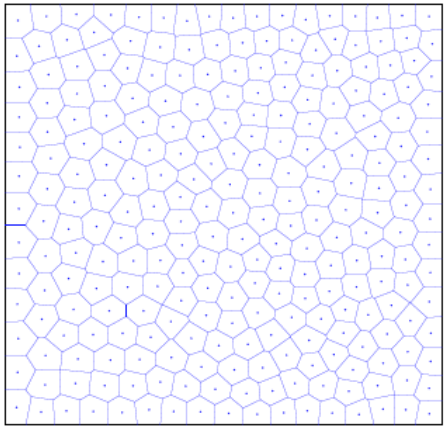
From [Pauleve 10]



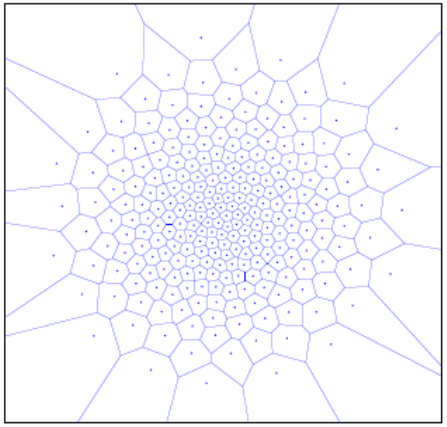
(a) Random projections



(b) A_2 lattice



(c) k-means
Uniform distribution

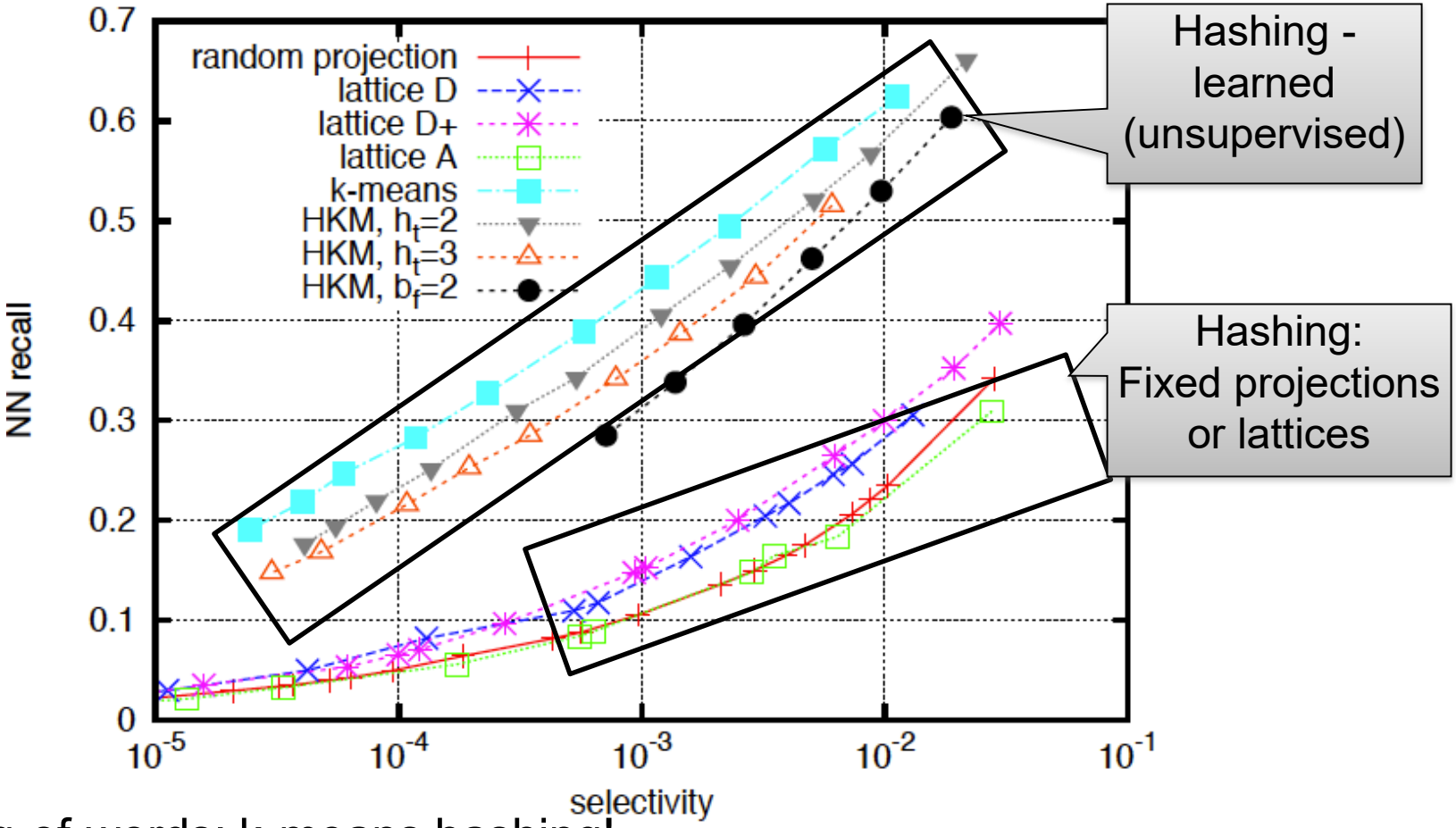


(d) k-means
Gaussian distribution

- Better data adaptation than with structured quantizers

Alternative hash functions – Learned

- Example of search: quality for a **single** hash function

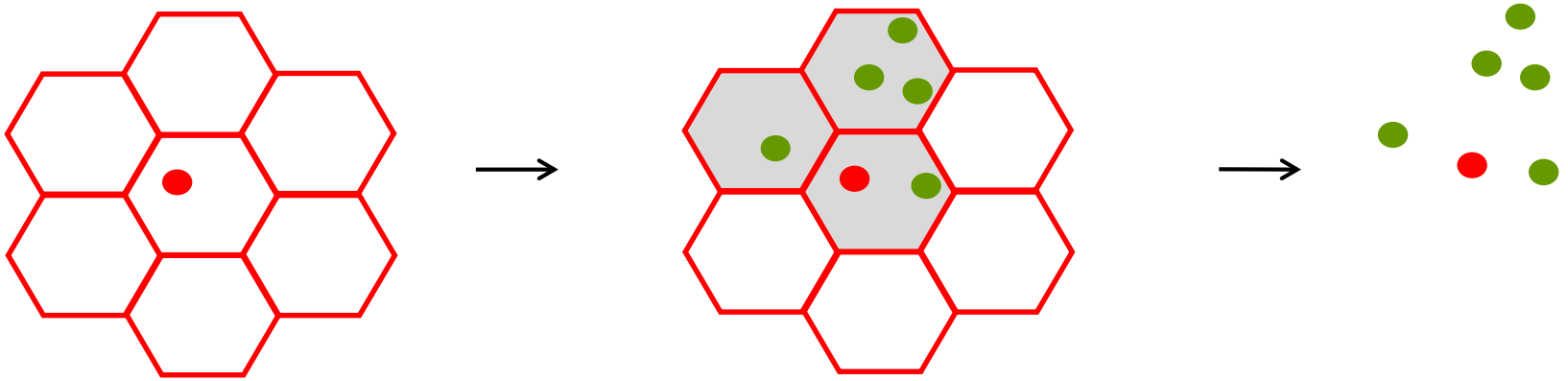


- Bag-of-words: k-means hashing!
- HKM: loss compared with k-means [Nister 06,Pauleve 10]

Multi-probe LSH [Lv 07]

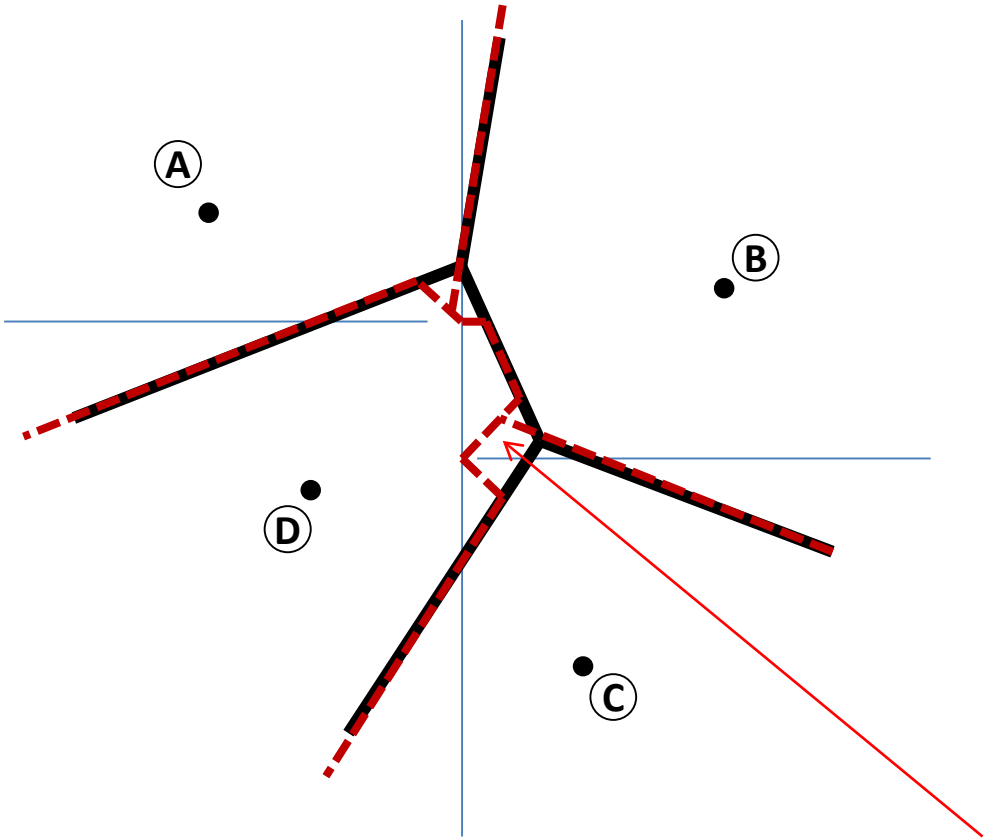
- Multiple hash functions use a lot of memory
  - Per vector and per hash table: at least an id

- Multi-probe LSH [Lv 07]: Use less hash functions (possibly 1)
  - Probe several (closest) cells per hash function
    ⇨ save a lot of memory
  - Similar to multiple/soft-assignment with BOW [J. 07, Philbin 08]



- See also: [Pauleve 10]

Partitioning by ANN



kd-tree depth 2

visiting two best bins

In this area only B and C are visited,
while the closest is D

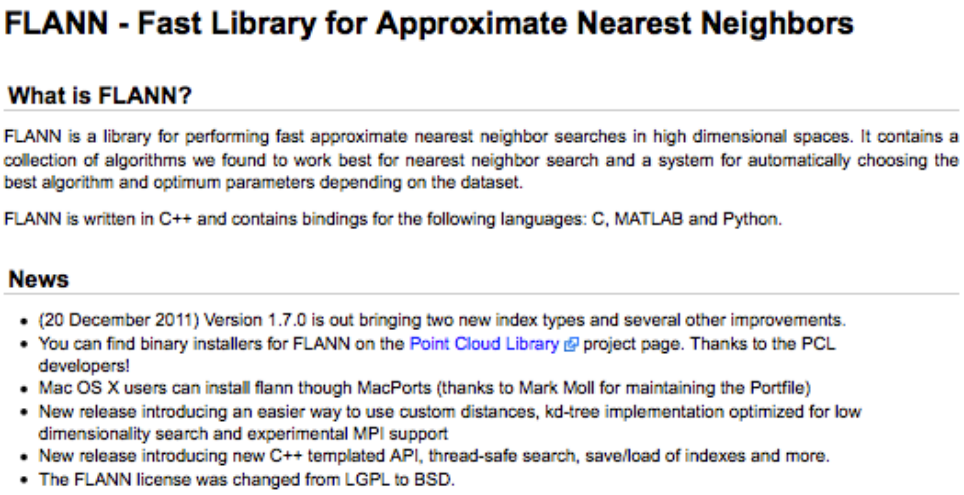
FLANN

- 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 08, Slaney 12]

- **Excellent package**: high integration quality and interface!
- Still **high memory requirement** for large vector sets

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



FLANN - Fast Library for Approximate Nearest Neighbors

What is FLANN?

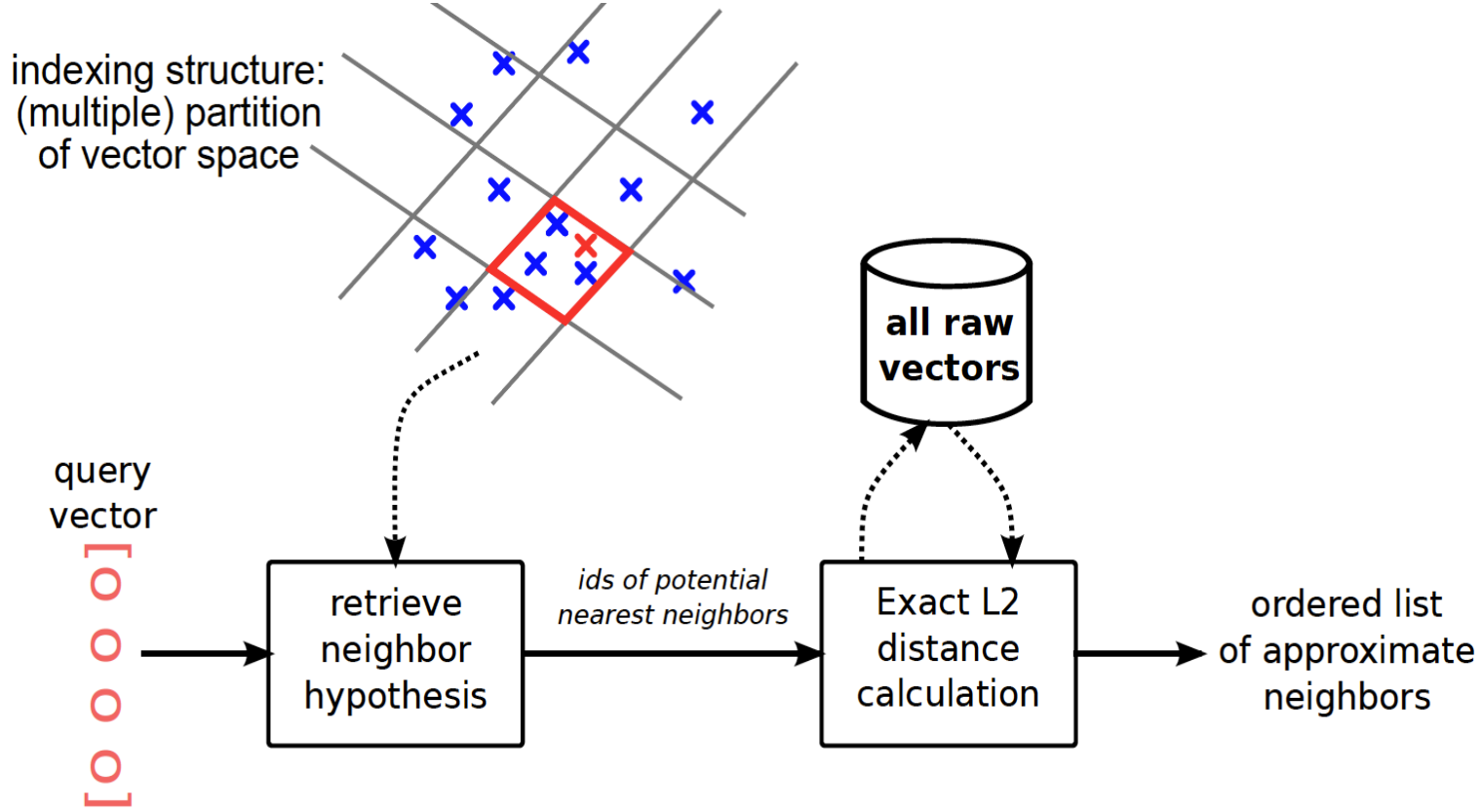
FLANN is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset.

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

News

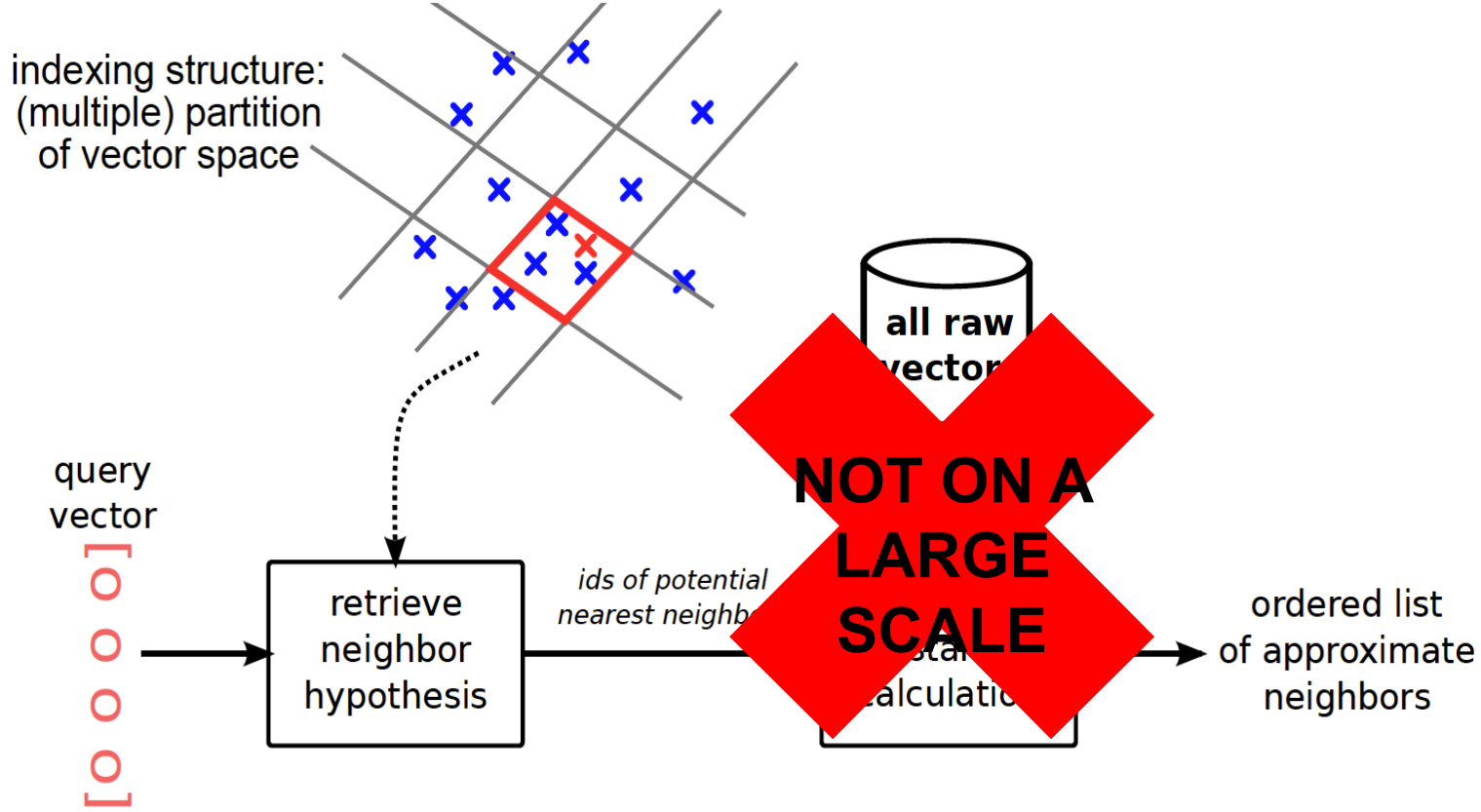
- (20 December 2011) Version 1.7.0 is out bringing two new index types and several other improvements.
- You can find binary installers for FLANN on the Point Cloud Library project page. Thanks to the PCL developers!
- Mac OS X users can install flann though MacPorts (thanks to Mark Moll for maintaining 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

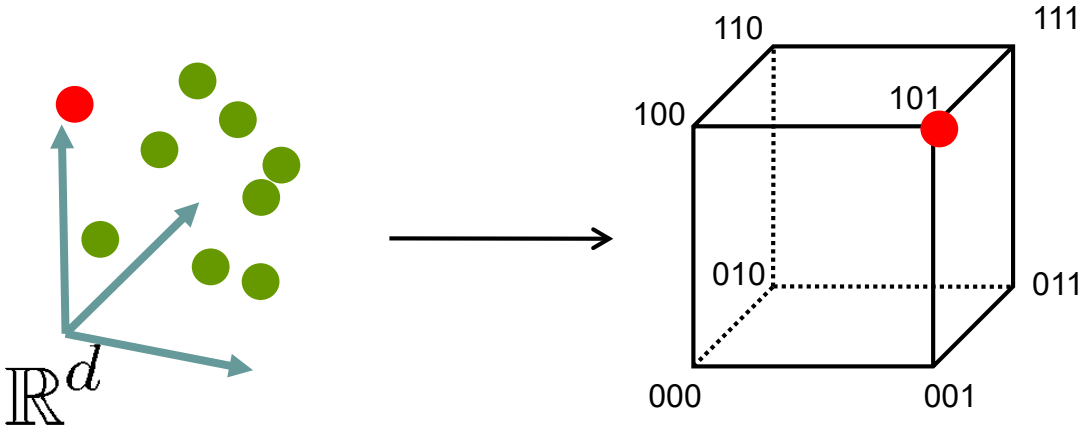


- Some techniques –like BOW– keep all vectors (no verification)

- Better: use very short codes for the filtering stage
  - See later in this presentation

Embedding

LSH for binarization



- Idea: design/learn a function mapping the original space into the compact Hamming space: $e : \mathbb{R}^d \rightarrow \{0,1\}^D$

  $x \rightarrow e(x)$

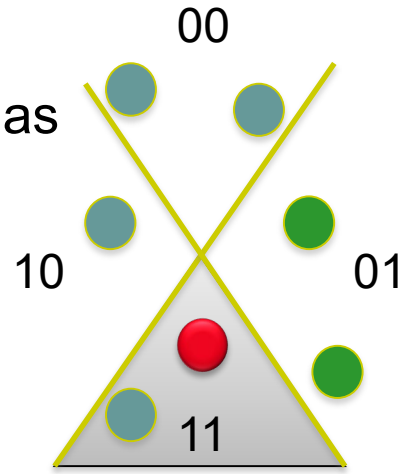
- Objective: neighborhood in the Hamming space try to reflect original neighborhood $\arg\min_i h\left(e(x), e(y_i)\right) \approx \arg\min_i d(x, y)$

- Advantages: compact descriptor, fast distance computation

Locality Sensitive Hashing (LSH)

- Given L random projection directions w_i
- For a given vector x, compute a bit for each direction, as

$$b_j(\mathbf{x}) = \operatorname{sign} \mathbf{w}_j^\top \mathbf{x}$$
$$\mathbf{b}(\mathbf{x}) = [b_1(\mathbf{x}), \ldots, b_L(\mathbf{x})]$$



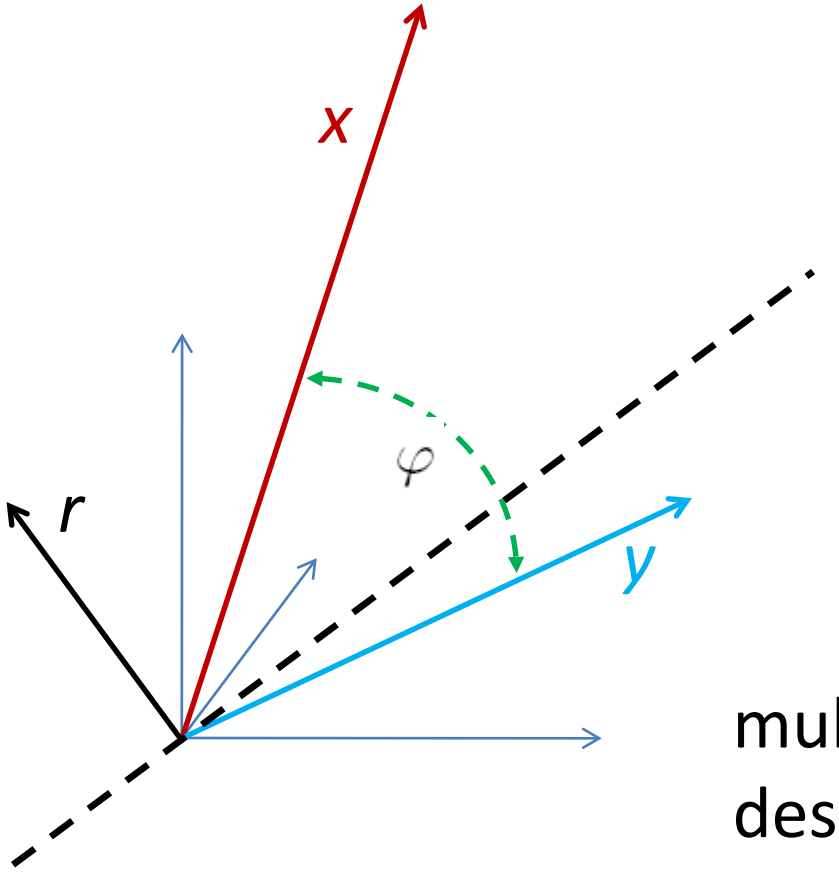
- Property: For two normalized vectors x and y:

$$\mathbb{P}(b_j(\mathbf{x}) \neq b_j(\mathbf{y})) = \frac{\theta}{\pi}$$
$$\mathbb{E}(d_{\mathsf{h}}(\mathbf{b}(\mathbf{x}), \mathbf{b}(\mathbf{y}))) = L\mathbb{P}(b_j(\mathbf{x}) \neq b_j(\mathbf{y}))$$

- The Hamming distance is related *in expectation* to the angle as

$$\hat{\theta} = \frac{\pi}{L} d_{\mathsf{h}}(\mathbf{b}(\mathbf{x}), \mathbf{b}(\mathbf{y}))$$ [Charikar 02]

Random Projections (Separation by a Random Hyperplane)



$h(x) = \text{sign}(r^T x)$

r uniformly distributed on a unit hypersphere

$P[h(x) = h(y)] = 1 - \varphi / \pi$

multiple hash function give a binary descriptor – Hamming distance

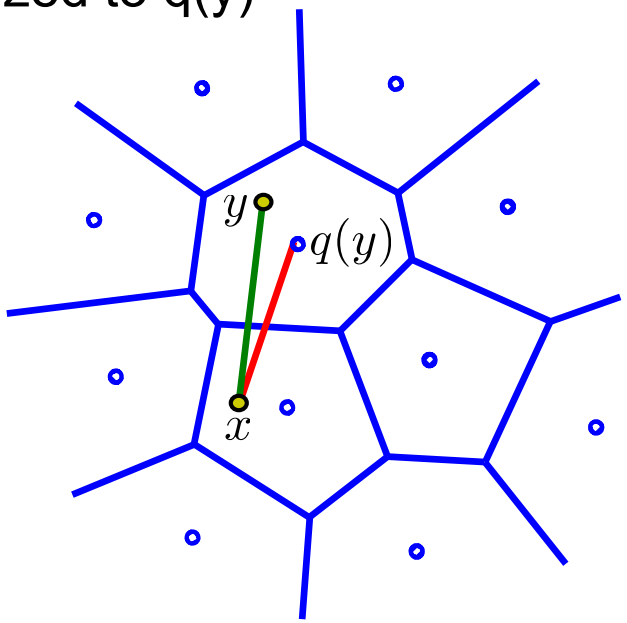
[*Goemans and Williamson 1995, Charikar 2004*]

Product Quantization

search ≈ distance estimation

- x is a query vector, database vector y quantized to q(y)

$$d(x,y)^2 \approx d(x,q(y))^2$$



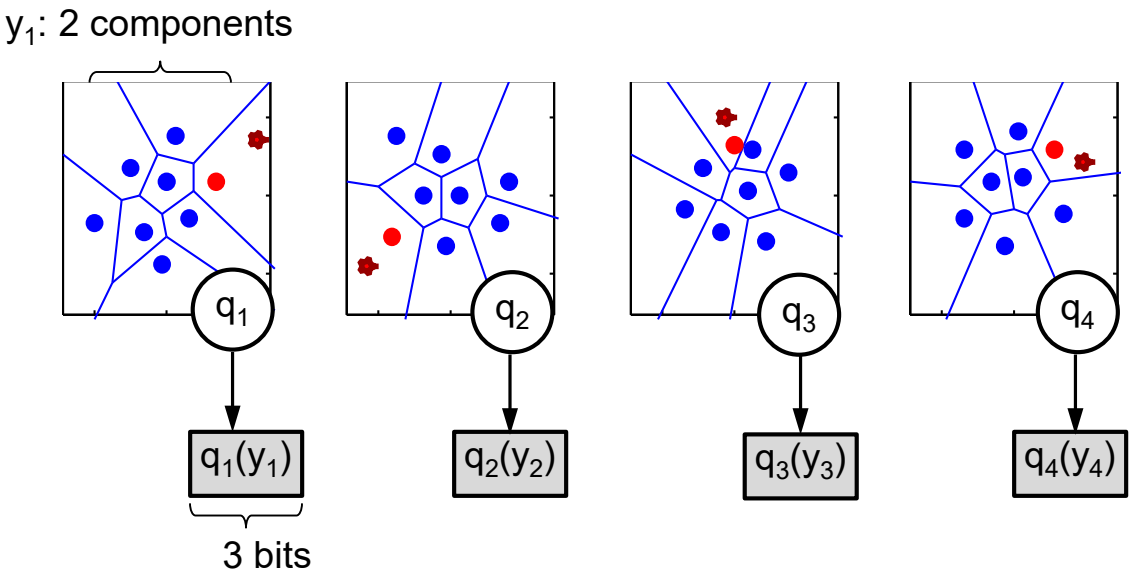
The error on square distances is statistically bounded by the quantization error

- **How to design q ?**
  - quantization should be 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
- Toy example: y = 8-dim vector split into 4 subvectors of dimension 2



8^4=4,096 centroids induced

for a quantization cost equal to that of 8 centroids

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

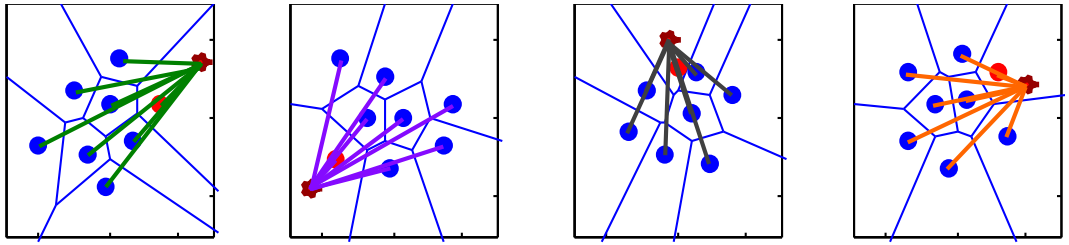
PQ: distance computation

- Estimate distances
  in the compressed domain

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

- To compute distances between query x and **many** codes:

I-



→ Precompute all distances between query subvectors and centroids: $d(x_i, c_{i,j})^2$

$c_{1,1}$	1.20	$c_{2,1}$	0.70	$c_{3,1}$	0.15	$c_{4,1}$	1.62
$c_{1,2}$	2.30	$c_{2,2}$	3.01	$c_{3,2}$	0.91	$c_{4,2}$	0.35
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$c_{1,8}$	0.34	$c_{2,8}$	2.84	$c_{3,8}$	1.29	$c_{4,8}$	1.44

→ Stored in look-up tables computed per query descriptor

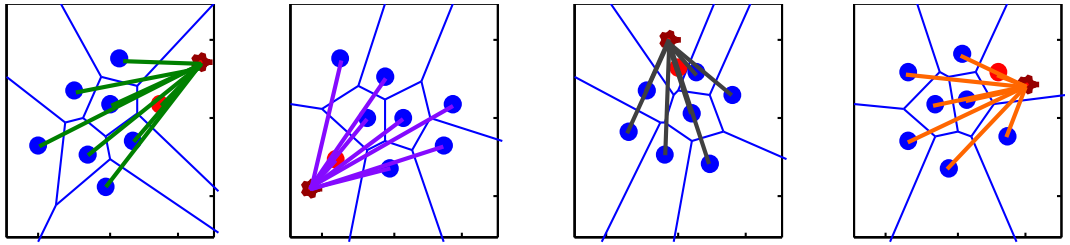
PQ: distance computation

- Estimate distances
  in the compressed domain

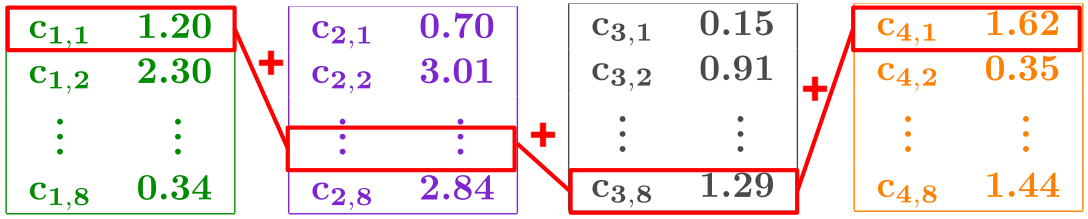
$$d(x,y)^2 \approx \sum_{i=1}^{m} d(x_i, q_i(y_i))^2$$

- To compute distances between query x and **many** codes:

I-



→ Precompute all distances between query subvectors and centroids: $d(x_i, c_{i,j})^2$



→ Stored in look-up tables computed per query descriptor

II- For each database vector: sum the elementary square distances

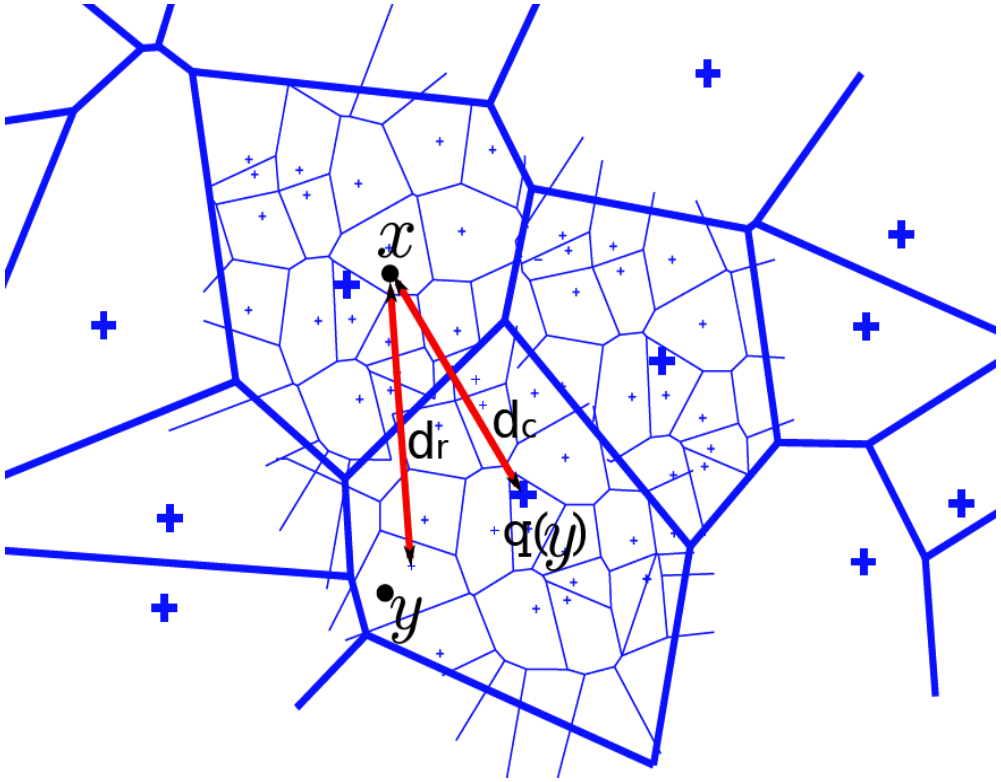
- **m-1 additions per distance**

IVFADC: integration with coarse partitioning

- PQ computes a distance estimate per database vector

- To improve scalability: combine distance estimation with a more traditional indexing structure

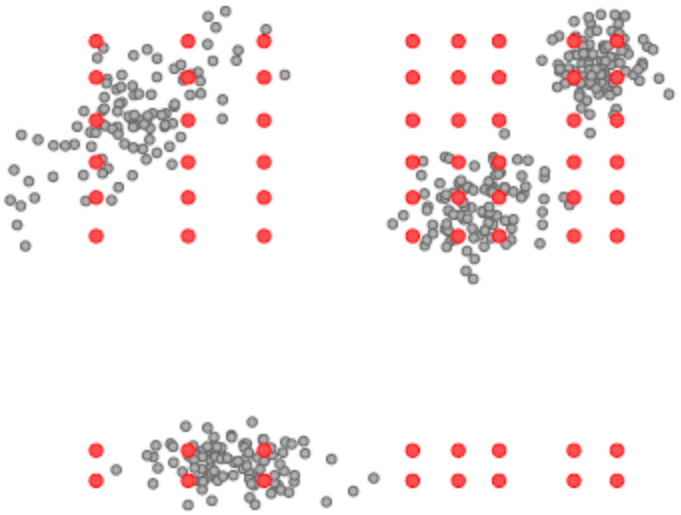
  → avoid exhaustive search



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

Product quantization

[Jégou et al. '11]



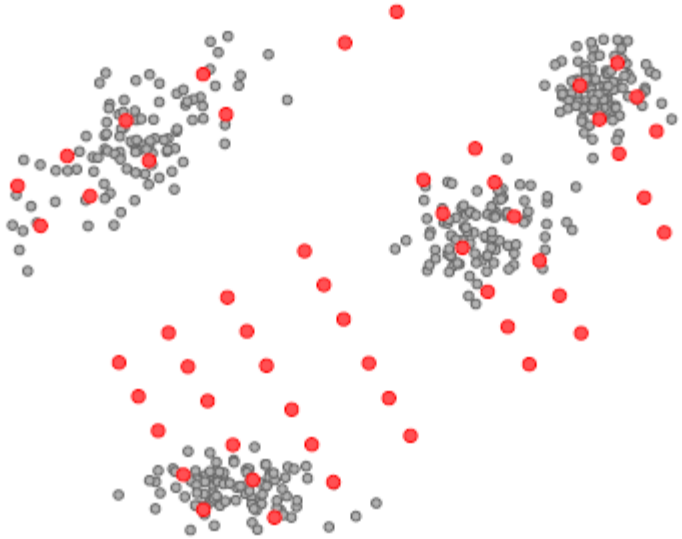
$$\text{minimize} \quad \sum_{\mathbf{x}\in\mathcal{X}} \min_{\mathbf{c}\in\mathcal{C}} \|\mathbf{x}-\mathbf{c}\|^2$$

$$\text{subject to} \quad \mathcal{C} = \mathcal{C}^1 \times \cdots \times \mathcal{C}^m$$

Slide courtesy of Yannis Avrithis

Optimized product quantization

[Ge et al. '13]

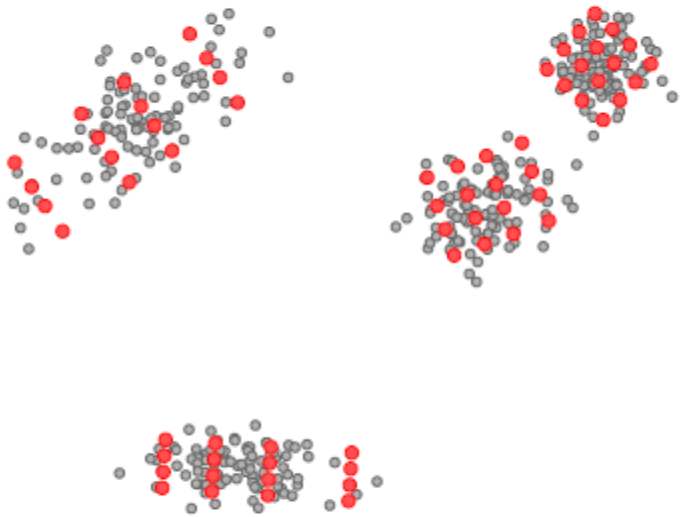


$$\begin{aligned} \text{minimize} \quad & \sum_{\mathbf{x}\in\mathcal{X}} \min_{\hat{\mathbf{c}}\in\hat{\mathcal{C}}} \|\mathbf{x} - R^\top \hat{\mathbf{c}}\|^2 \\ \text{subject to} \quad & \hat{\mathcal{C}} = \mathcal{C}^1 \times \cdots \times \mathcal{C}^m \\ & R^\top R = I \end{aligned}$$

Slide courtesy of Yannis Avrithis

Locally optimized product quantization

[Kalantidis & Avrithis '14]

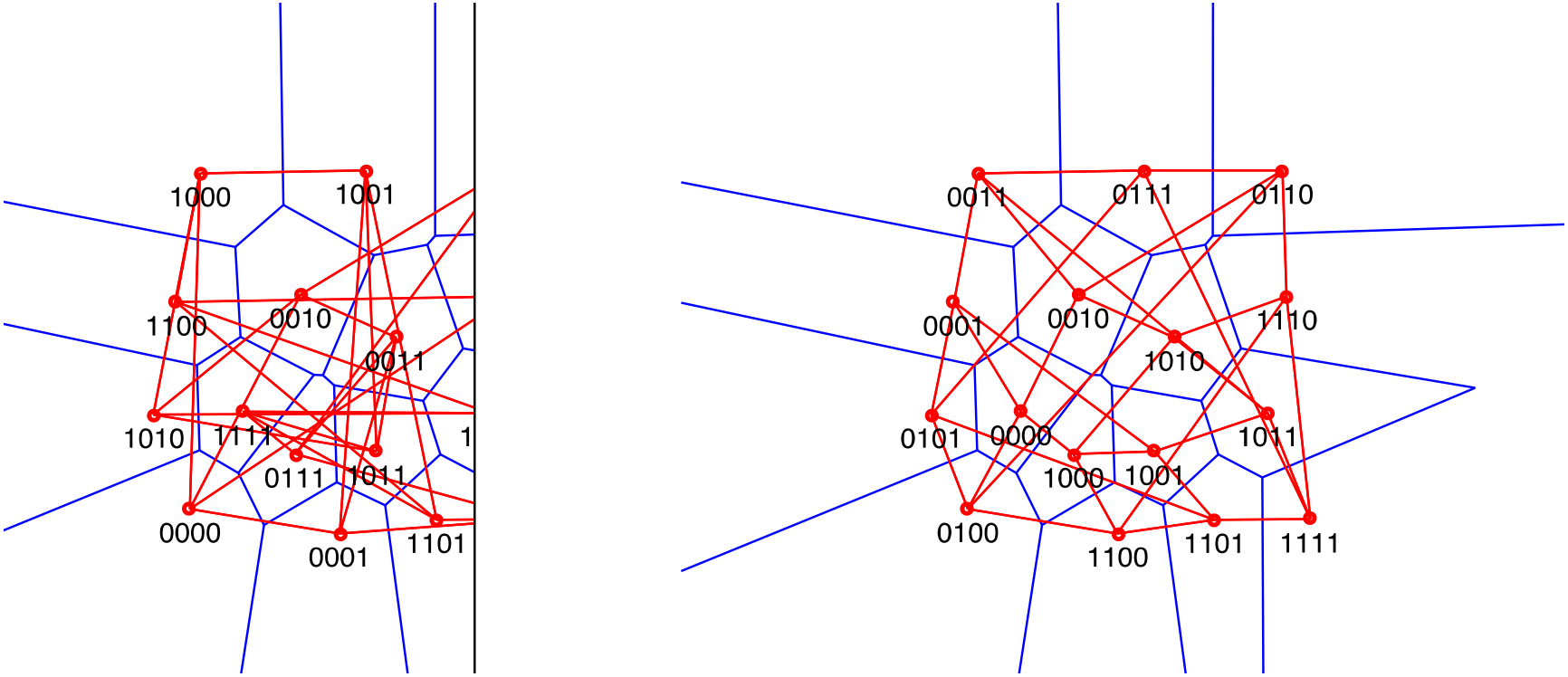


- compute residuals $r(\mathbf{x}) = \mathbf{x} - Q(\mathbf{x})$ on coarse quantizer Q
- collect residuals $\mathcal{Z}_i = \{r(\mathbf{x}) : Q(\mathbf{x}) = \mathbf{c}_i\}$ per cell
- train $(R_i, q_i) \leftarrow \mathrm{OPQ}(\mathcal{Z}_i)$ per cell

Slide courtesy of Yannis Avrithis

Polysemous codes

- Given a k-means quantizer
- Learn a permutation: such that the binary comparison reflects centroid distances
- Done for each subquantizer
- Optimized with simulated annealing



[Polysemous codes, Douze, J., Perronnin, ECCV'16]

FAISS package

FAISS: Fast similarity search

8.5x faster than prior State-of-the-art

Search through:

1M images in 20 micro-seconds SIFT Features R@1=0.5

1B images in 17.7 micro-seconds SIFT Features R@10=0.376

1B ConvNet features in 13.3 micro-seconds R@1=0.45

Python and C++ packages on Github

Image-interpolation in semantic-space: brute-force and fast

Figure 6: Path in the k-NN graph of 95 million images from YFCC100M. The first and the last image are given; the algorithm computes the smoothest path between them.

Jeff Johnson, Matthijs Douze, Herve Jegou: **Billion scale similarity search with GPUs**

Approximate Nearest Neighbour
(a biref into)

Ondrej Chum

slide credits: Herve Jegou, Yannis Avrithis