Vector search #8 – Quantization for lossy vector compression

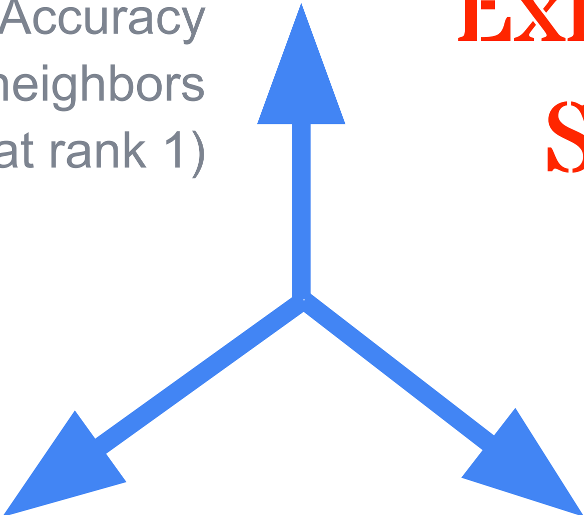
Princeton COS579A – Edo Liberty and Matthijs Douze – 2023-11-10

Tradeoffs of vector search

3-handed tradeoff

Accuracy
(% of actual nearest neighbors
found at rank 1)

Exhaustive
Search



Memory (RAM)
(bytes per vector)

Search speed
(ms per vector)

Pruning

Compression

In this class

- Mainly about the “compression” hand
- Fix size of representation
  - Because RAM is constrained
- Operating points between the two other hands

- Examples from Faiss
  - Implements many index types
  - Explore boundaries

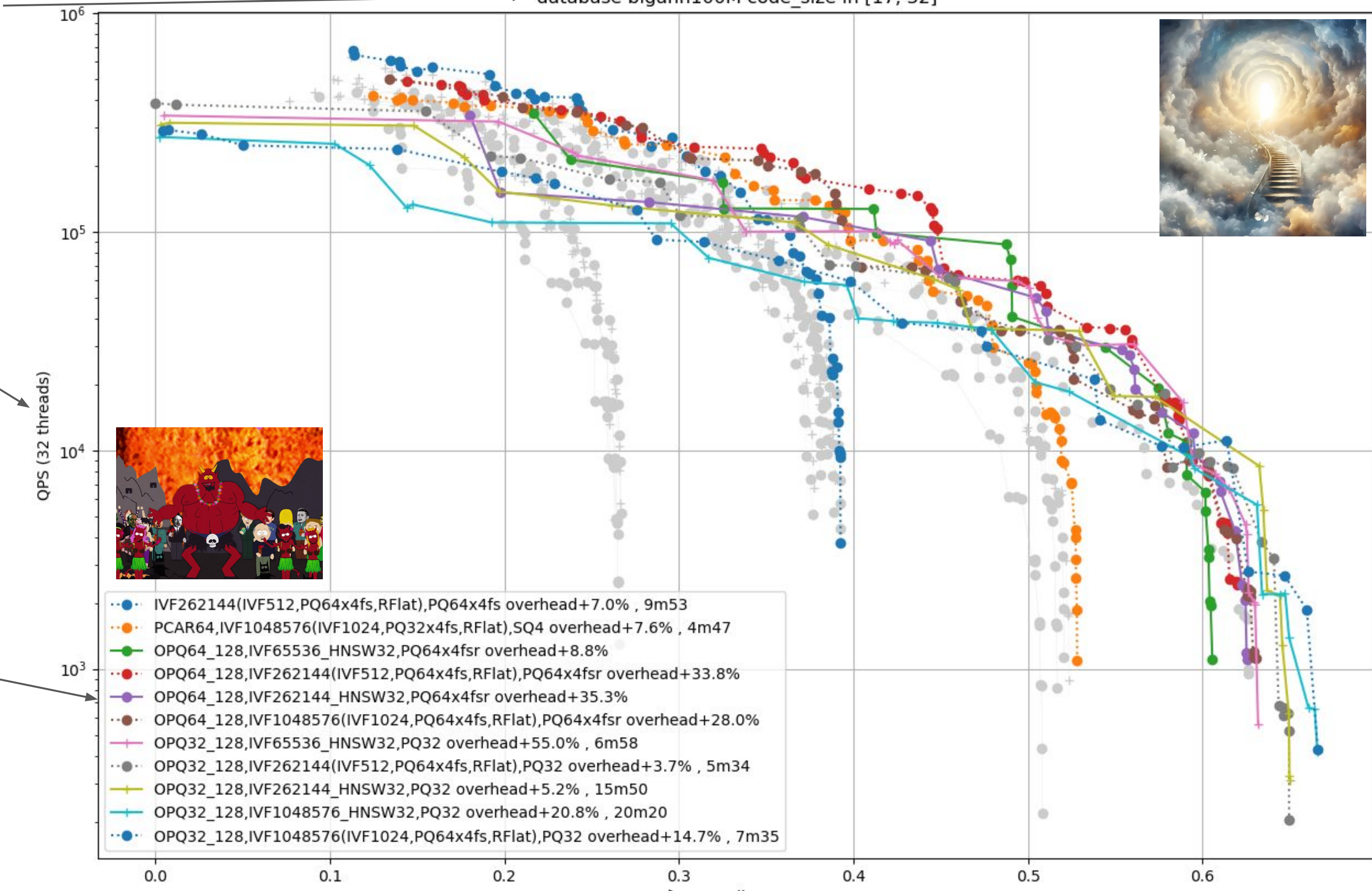
Next page: Faiss wiki
https://github.com/facebookresearch/faiss/wiki/Indexing-1G-vectors

Memory budget: max 32 bytes per vector

Speed: Queries per second (Log scale)

Each curve = one type of index

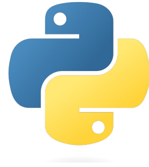
Accuracy: 1-recall@1



1-recall at 1

omitted: PCAR64,IVF262144_HNSW32,SQ4 IVF262144_HNSW32,PQ64x4fs OPQ64_128,IVF1048576(IVF1024,PQ64x4fs,RFlat),PQ64x4fs OPQ64_128,IVF262144_HNSW32,PQ64x4fs PCAR64,IVF65536_HNSW32,SQ4 OPQ64_128,IVF1048576_HNSW32,PQ64x4fs IVF65536_HNSW32,PQ64x4fs PCAR32,IVF65536_HNSW32,SQ8 PCAR32,IVF262144_HNSW32,SQ8 PCAR64,IVF262144(IVF512,PQ32x4fs,RFlat),SQ4 OPQ64_128,IVF1048576_HNSW32,PQ64x4fsr IVF1048576_HNSW32,PQ64x4fs PCAR32,IVF1048576(IVF1024,PQ16x4fs,RFlat),SQ8

Presented in conjunction with a Python notebook

- This sign: 
- Means there is related content in the notebook.

Vector quantization

Vector quantization: definition

- Quantization: map a vector to an integer
  - Input is (supposed to be) continuous
  - Output is discrete
- Integer ≡ bit array of fixed size ≡ code
- Reconstruction: inverse map
  - We recover only an approximation
  - The reconstruction is lossy
- Evaluation: Mean Squared Error
  - Because it has nice arithmetic properties…
  - Invariant with d-dim rotation

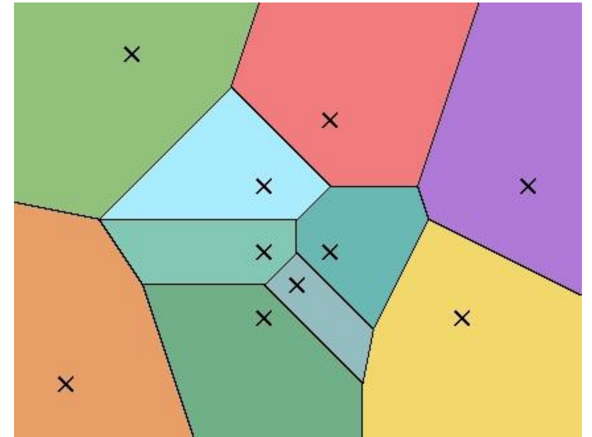
$$q : \mathbb{R}^d \mapsto \{0, ..., k-1\}$$

$$r : \{0, ..., k-1\} \mapsto \mathbb{R}^d$$

$$\mathrm{MSE} = \mathbb{E}_x \left[\| r(q(x)) - x \|^2 \right]$$

Codes and relationship with clustering

- Numbering is sequential
  - Otherwise do a mapping in the discrete domain
- Size of codes $\lceil \log_2(k) \rceil$



- Quantization cell
  - Locus of vectors that produce the same code

$$q^{-1}(\{i\}), \;\; i \in \{0, ..., k-1\}$$

- All quantization cells are a partition of input space
- From a discrete point of view: clustering
  - Reconstruction values are called “centroids”

Lloyd’s optimality conditions (reminder)

- To minimize the MSE for discrete sets
- 1: a vector should be assigned to the nearest reconstruction
  - Otherwise re-assign that vector: decrease MSE!
- 2: each centroid should be the center of mass of points assigned to it
  - Otherwise just move the center of mass: decrease MSE!
- Necessary, not sufficient

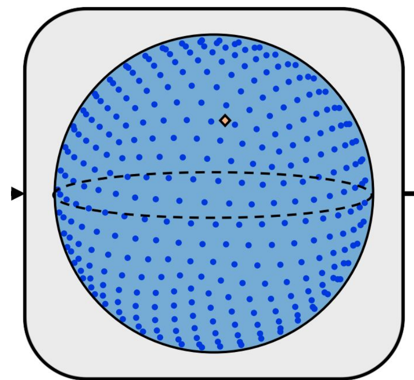
- The k-means algorithm
  - Iterate the two optimality conditions

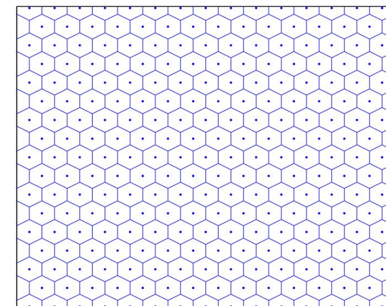
Cases where the optimal centroids are known

- Uniform distribution
  - A_x Lattice
- Uniform over a sphere
  - Integer lattice on sphere
- Unknown for Gaussian data

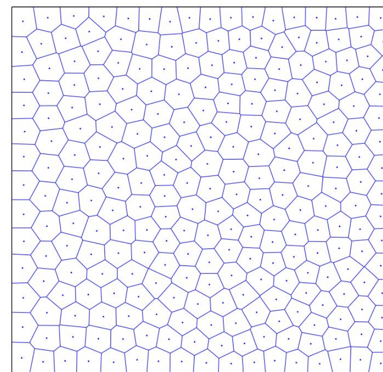
[Paulevé et al, Locality sensitive hashing: a comparison of hash function types and querying mechanisms, Pattern recog letters 2010]

[Sablayrolles et al, spreading vectors for similarity search, ICLR’19]

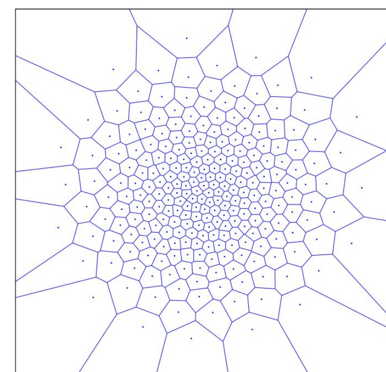




(b) A_2 lattice



(c) k-means
Uniform distribution



(d) k-means
Gaussian distribution

Figure 3: Voronoi regions associated with random projections (a), lattice A_2 (b) and a k-means quantizer (c,d).

How optimal is k-means?

Optimality of k-means: practical considerations

- On small scale, practical k-means is quite off from the global optimum
- On large scale we don’t know!
  - NP-hard is very hard

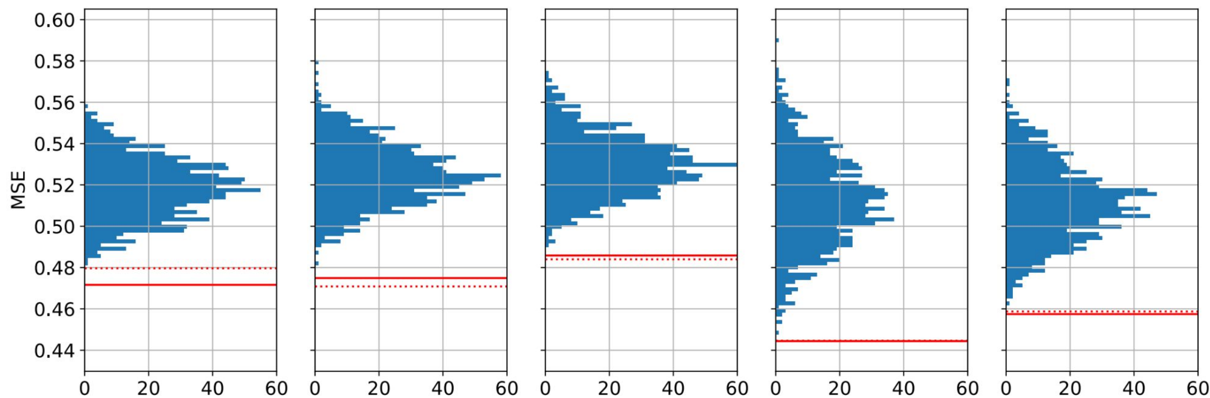
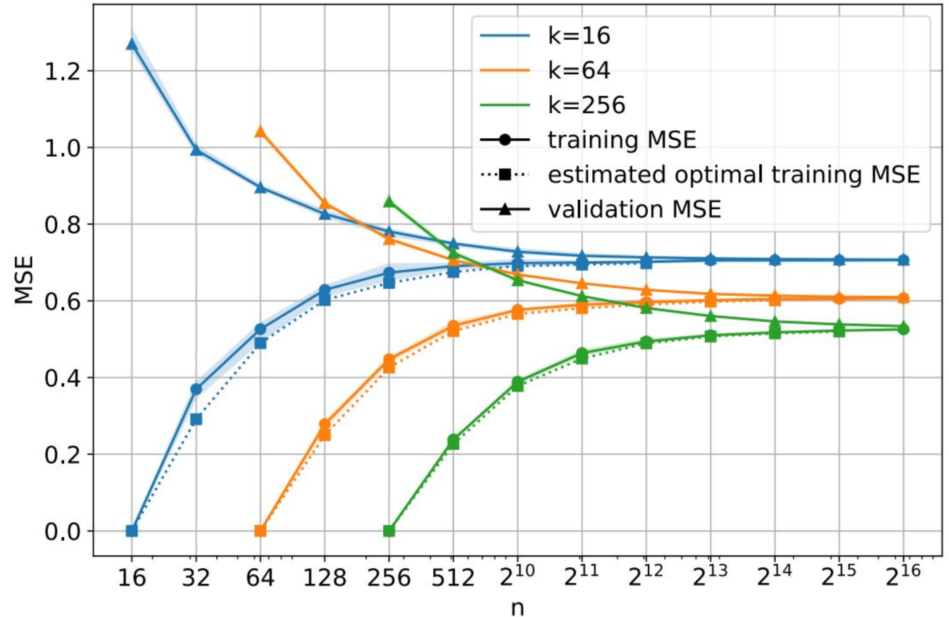


Figure 3: Histogram of MSE results for 1000 runs of k-means with $(n, k) = (24, 8)$. The experiment is run on 5 subsets of the same data distribution, hence the 5 plots. The exact global optimum is indicated in red. The estimate of this optimum from the k-means runs is the red dashed line.

[Augmented k-means, Touvron, Douze, Jégou, unpublished]

Difference between training and validation

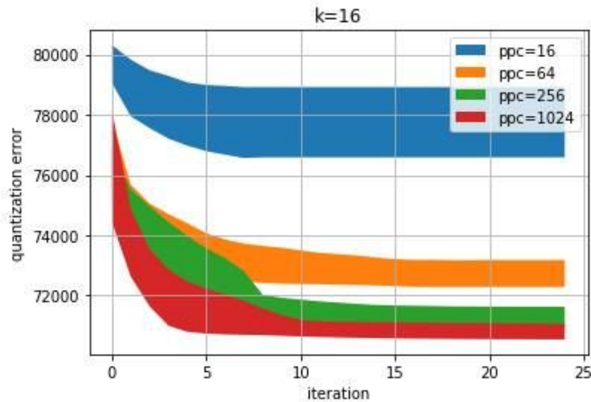
- MSE can be computed on
  - Training vectors → Lloyd's conditions
  - Validation → how it's going to be used in reality
- Small and large-data regime
- Relevant parameter is n/k
  - Nb training points per centroid

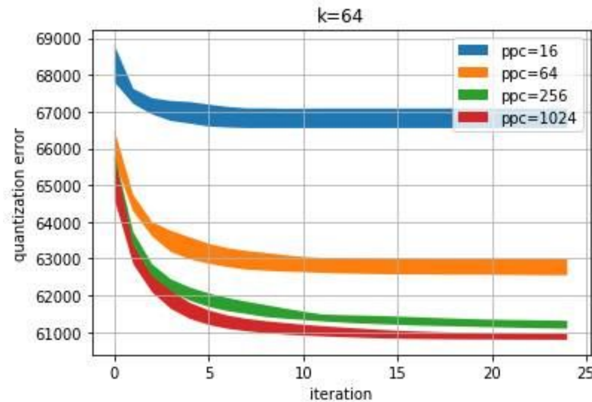


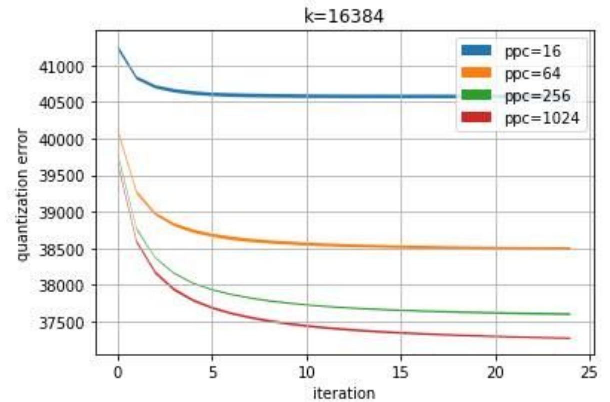
Overfitting ←→ train-val convergence

Optimality of k-means: practical considerations

- As a function of the Points Per Centroid (ppc)
- Validation MSE
- Line thickness: min and max of 10 k-means runs



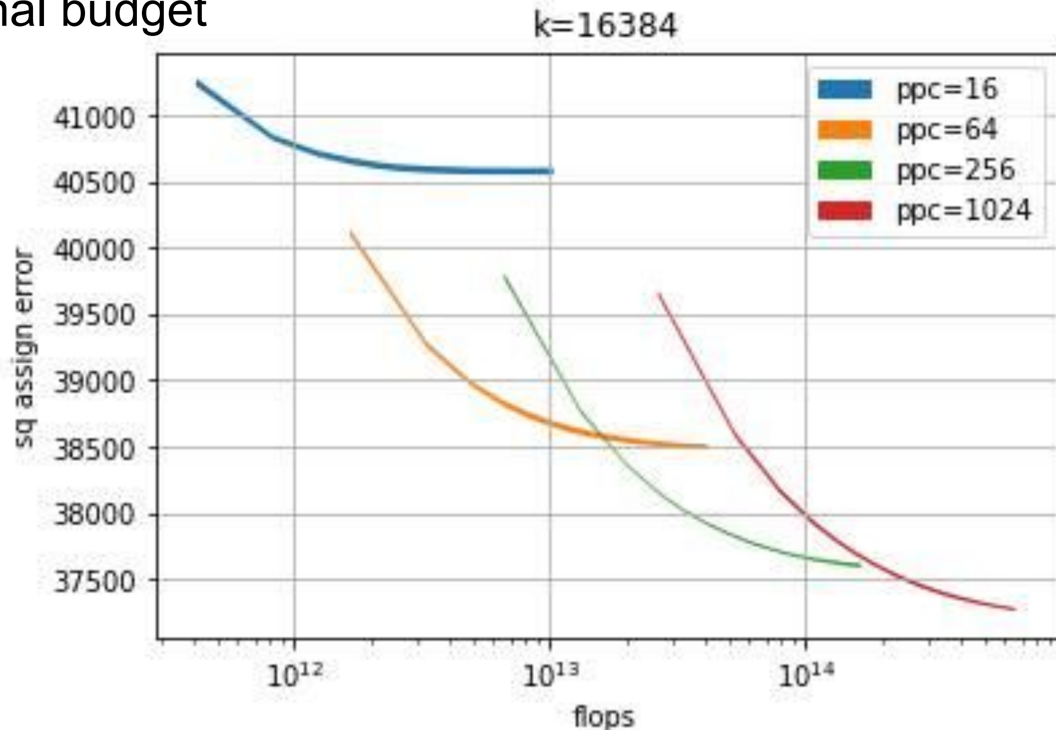




- It does not matter to have many initializations when k increases

Making good use of FLOPS

- Training a 100-centroid k-means on 1M vectors
  - Waste of resources?
- Given a certain computational budget
  - Balance nb of iterations and nb points per centroid



Scaling k-means

Scaling k-means

- Complexity
- Required to get larger codes
  - 3 bytes = 24 bits = 16M centroids
  - Not a very large code…
  - But a HUGE number of centroids
- Expensive stage is assignment
  - NNS problem
- Brute-force hardware scaling
  - CPU training
  - GPU training
  - Distributed training…

Matthijs Douze
March 8, 2019 ·

KMeans of 500M points to 10M centroids

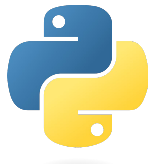
144 dim, 20 iterations, 10x8 GPUs: 22h.

This is the largest k-means optimization we have done so far with Faiss. It required a version where the training set is distributed over 10 machines. The k-means logic was also re-implemented in Python. Nothing fancy, just brute force.

[Nistér Stewenius, Scalable recognition with a vocabulary tree, CVPR'06]
[Muja, Lowe, Scalable Nearest Neighbor Algorithms for High Dimensional Data, PAMI'14]

Hierarchical k-means

- Inspired by bisection in 1D
- Run k-means recursively to subdivide
  - Iterate
- Often used for search
  - Basis of the FLANN library
- But sub-optimal in terms of MSE
  - See notebook



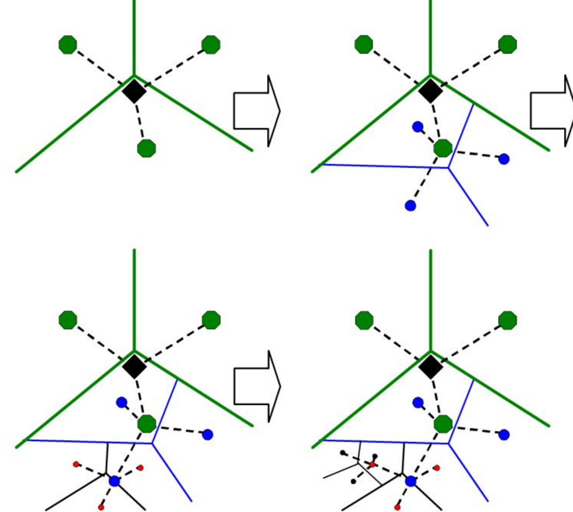


Figure 2. An illustration of the process of building the vocabulary tree. The hierarchical quantization is defined at each level by k centers (in this case $k = 3$) and their Voronoi regions.

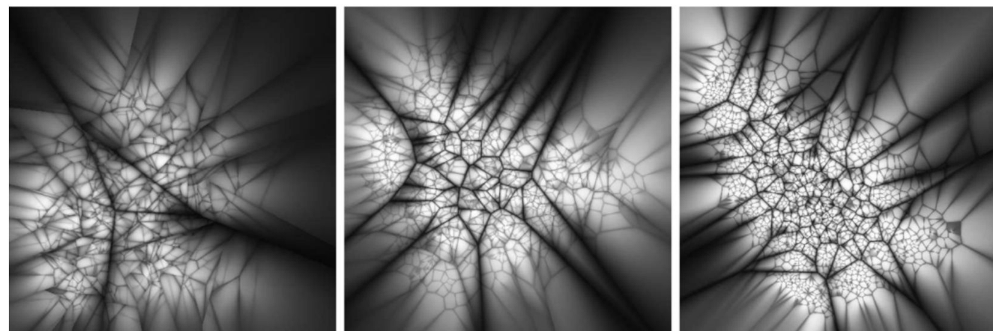
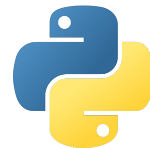


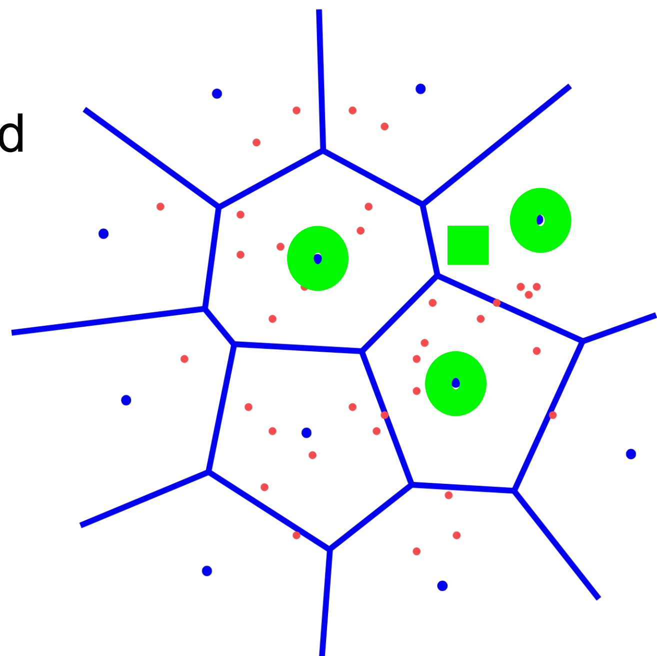
Fig. 3. Projections of priority search k-means trees constructed using different branching factors: 4, 32, 128. The projections are constructed using the same technique as in [26], gray values indicating the ratio between the distances to the nearest and the second-nearest cluster centre at each tree level, so that the darkest values (ratio $\approx$ 1) fall near the boundaries between k-means regions.

Vector quantization for search:
The inverted file

The Inverted File

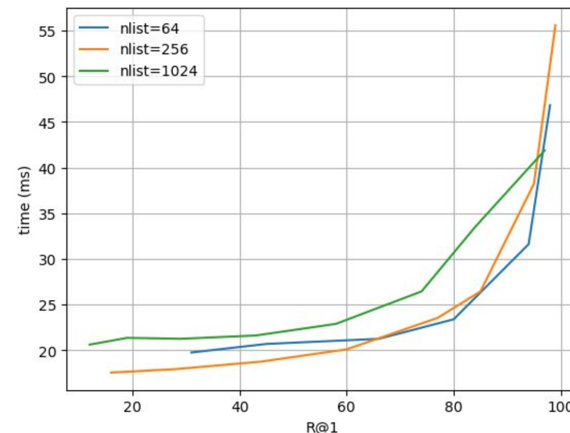


- Cluster the space into k clusters of vectors
  - assign vectors to nearest centroid
  - Aka. “coarse quantizer”
- index = inverted list structure
  - maps centroid id → list of vectors assigned to it
- search procedure:
  - 1. find np << k nearest centroids to query vector
  - 2. scan the lists corresponding to the np centroids
- Objective: reduce the number of distance computations!

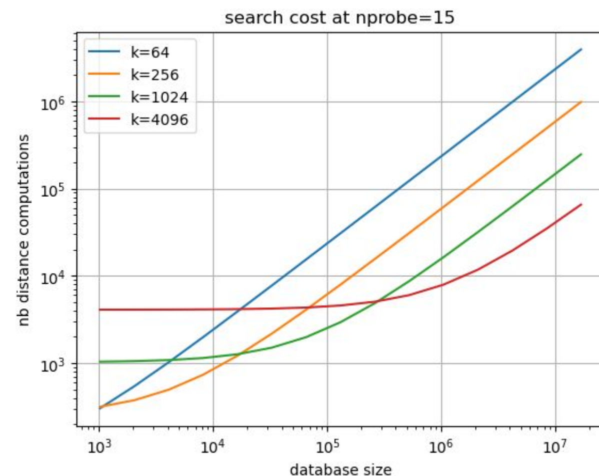


The Inverted File: number of centroids

- Tradeoffs
  - For high-accuracy regime it is not necessary to be very selective: small nb clusters
  - For low-accuracy regime it is better to filter more with clusters

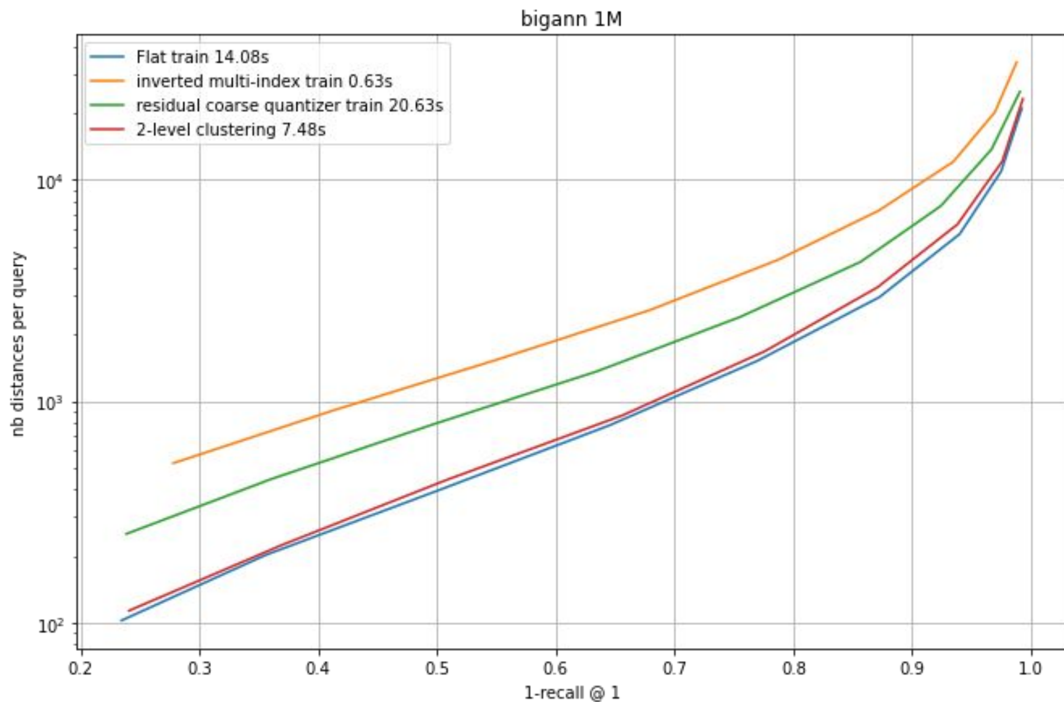


- As a function of database size:
  - Coarse quantization: k distance computations
  - Scanning: (assuming balanced clusters):
    nprobe * n / k
  - Exercise: find optimal k as a function of n for fixed nprobe



Comparing coarse quantizers

- Useful with 2 levels
  - Reduces complexity



Vector quantization for compression

Compression and search: asymmetric case

Collection:

Query:

embedding

$y_1, y_2, ..., y_n \in \mathbb{R}^d$

quantization

$c_1, ..., c_n \in \{1, ..., k\}$

Index

$x \in \mathbb{R}^d$

Result: $k - \mathrm{argmin}_{i=1..n} \|x - r(c_i)\|^2$

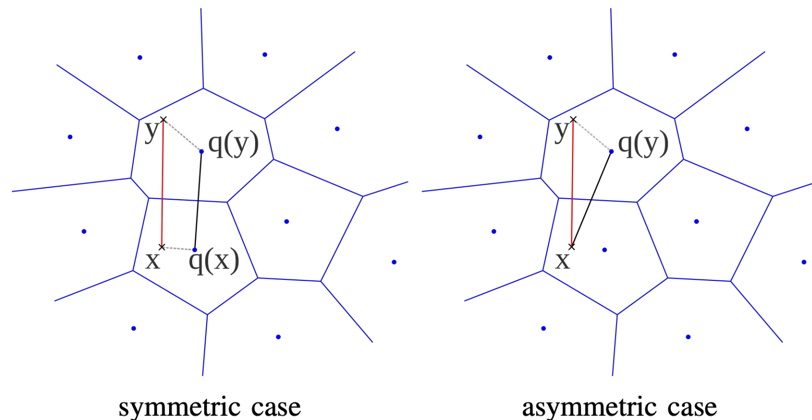
Symmetric vs. asymmetric comparison

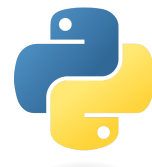
- Consider asymmetric setting
  - no constraint on storage of query
  - keep full query vector, encode database vectors $q(y)$
- Distance estimator
  - reproduction value of the quantizer: centroid
  - approximate distance

$$\|x - y\| \approx \|x - q(y)\|$$

  - approximate nearest neighbor

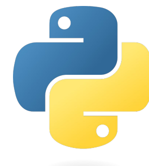
$$\text{argmin}_i \|x - q(y_i)\|$$





Distance computations with look-up tables

- For a given query x, there are k possible distances
  - Precompute a table!
  - At search time, look up the distances in the table.
  - No computations at search time, only look-ups
  - Useful if nb centroids << nb database elements



- However, this is pretty limited
  - Small codes – limited recall…

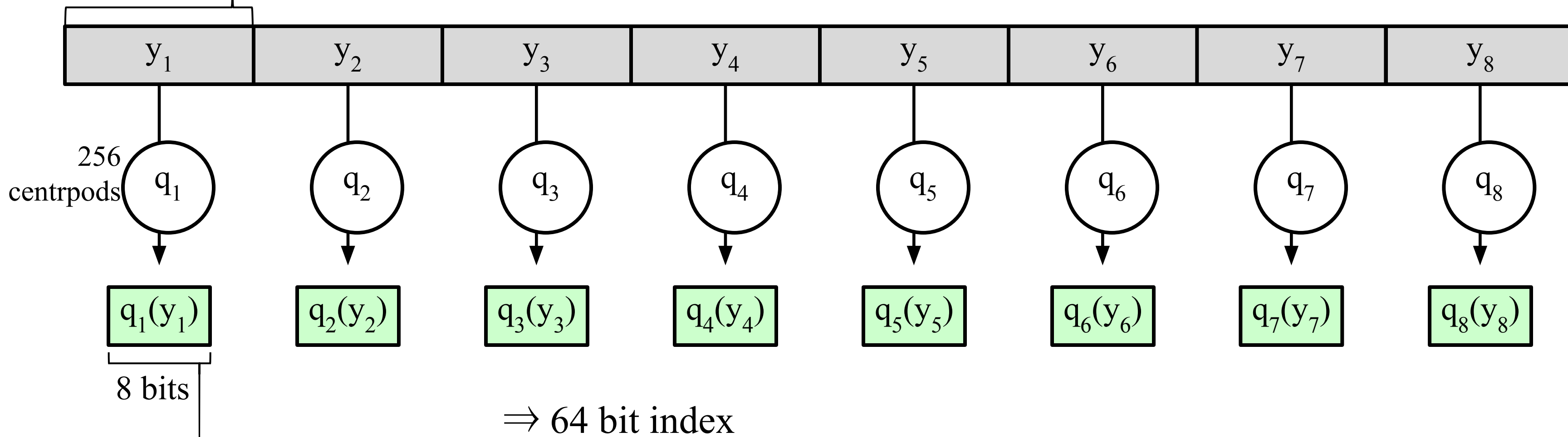
Multi-codebook quantization

Multiple-codebook quantization

- Combine multiple quantizers
  - Each has its codebook
  - Separate codes, total size $\sum_{m=1}^{M} \lceil \log_2(k_m) \rceil$
- In the following:
  - Product quantization
  - Additive quantization

[Jégou & al, Product quantization for nearest neighbor search, PAMI'11]

Product Quantization



Multiple-codebook quantization

- reconstruction value: concatenation of centroids

$$y \approx [q_1(y^1), ..., q_m(y^m)]$$

- distance computation: distance is additive

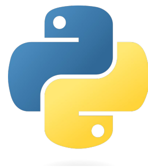
$$\|x - y\|^2 \approx \sum_{j=1}^{m} \|x^j - q_j(y^j)\|^2$$

- precompute *M* look-up tables!

Sizes & flops

	no compression	vector quantizer	product quantizer
code size	d	log2(k)	m*log2(k)
quantization cost	N/A	k*d	k*d
distance computation cost	d multiply-adds	one look-up, one add	m look-ups, m adds
number of distinct values	N/A	k	k^m

Product Quantizer tradeoffs



- For a given code size
  - Code_size = M * log2(k)
- Higher k (and lower M)
  - Better accuracy
  - Larger quantization tables (slower)
- When k → k^2 and M → M/2
  - The "small" PQ can be expressed in the "large" PQ
- Extremes:
  - M=1 → full k-means
  - k=1 → binary encoding

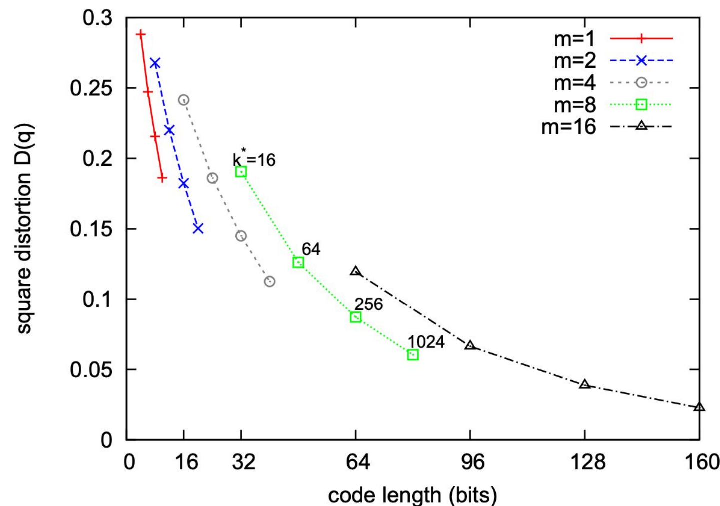
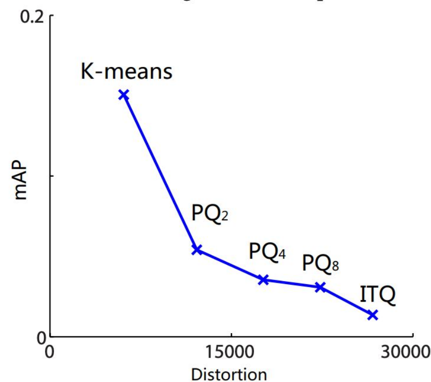


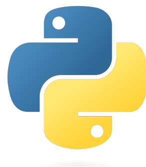
Fig. 1. SIFT: quantization error associated with the parameters m and k^*.



[Ge & al, Optimized Product Quantization, PAMI'13]

Optimized product quantization

- Basic PQ splits the vectors into arbitrary subvectors
  - Accuracy is dependent on how well the subvectors are decorrelated
  - Worst case: the vector actually repeats M times the same sub-vector!
    → then it becomes equivalent to a single k-means
- OPQ trains a rotation matrix to optimally decorrelate them
  - Iterative optimization



Step (i): Fix R and optimize $\{\mathcal{C}^m\}_{m=1}^M$.

Denote $\hat{\mathbf{x}} = R\mathbf{x}$ and $\hat{\mathbf{c}} = R\mathbf{c}$. Since R is orthogonal, we have $\|\mathbf{x} - \mathbf{c}\|^2 = \|\hat{\mathbf{x}} - \hat{\mathbf{c}}\|^2$. With R fixed, (4) then becomes:

$$\min_{\mathcal{C}^1,\ldots,\mathcal{C}^m} \sum_{\hat{\mathbf{x}}} \|\hat{\mathbf{x}} - \hat{\mathbf{c}}(i(\hat{\mathbf{x}}))\|^2, \quad s.t. \quad \hat{\mathbf{c}} \in \mathcal{C}^1 \times \ldots \times \mathcal{C}^M. \tag{5}$$

This is the same problem as PQ in (2). We can separately run k-means in each subspace to compute the sub-codebooks.

Step (ii): Fix $\{\mathcal{C}^m\}_{m=1}^M$ and optimize R.

Since $\|\mathbf{x} - \mathbf{c}\|^2 = \|R\mathbf{x} - \hat{\mathbf{c}}\|^2$, the sub-problem becomes:

$$\min_{R} \sum_{\mathbf{x}} \|R\mathbf{x} - \hat{\mathbf{c}}(i(\hat{\mathbf{x}}))\|^2, \quad s.t. \quad R^{\mathrm{T}}R = I. \tag{6}$$

The codeword $\hat{\mathbf{c}}(i(\hat{\mathbf{x}}))$ is fixed in this subproblem. It is the concatenation of the M sub-codewords of the sub-vectors in $\hat{\mathbf{x}}$. We denote $\hat{\mathbf{c}}(i(\hat{\mathbf{x}}))$ as $\mathbf{y}$. Given n training samples, we denote X and Y as two D-by-n matrices whose columns are the samples $\mathbf{x}$ and $\mathbf{y}$ respectively. Then we can rewrite (6) as:

$$\min_{R} \|RX - Y\|_{\mathrm{F}}^2, \quad s.t. \quad R^{\mathrm{T}}R = I, \tag{7}$$

Additive quantization

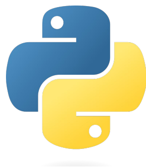
[Chen et al, Approximate Nearest Neighbor Search by Residual Vector Quantization, sensors'10]
[Liu Shicong & al, Improved Residual Vector Quantization for High-dimensional Approximate Nearest Neighbor Search, AAAI'15]

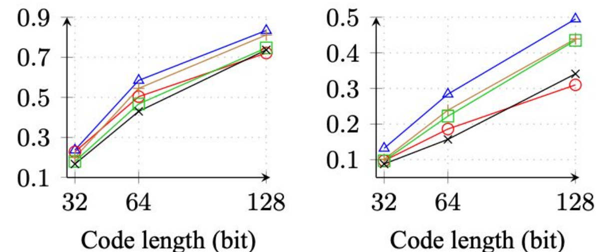
Residual quantization

- Simple principle:
  - Sequential encoding of residuals

$$q_1(x) \rightarrow q_2(x - r_1(q_1(x))) \rightarrow q_3(x - r_1(\ldots) - r_2(\ldots)) \rightarrow \ldots$$

- Does not work too well by default
  - Due to greedy search
  - Extend with beam search
-



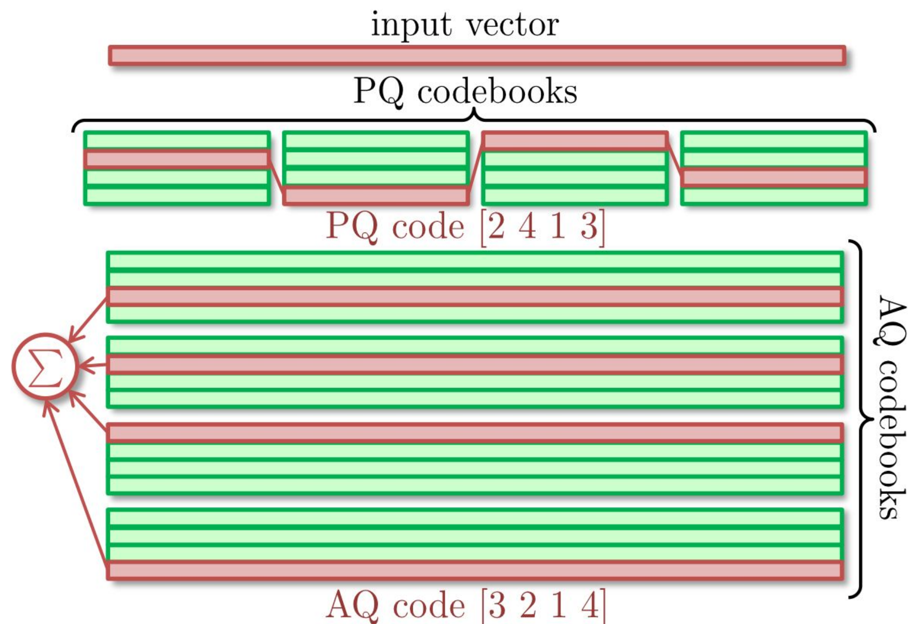


(a) Recall@4 on SIFT1M, different code length (b) Recall@4 on GIST1M, different code length

[Babenko & Lempitsky,Additive quantization for extreme vector compression, CVPR’14]

Additive quantization

- Use multiple codebooks
  - But in full dimension d
- More capacity
  - PQ is a special case of AQ (where subspaces are set to 0)
- Encoding is not easy
  - No independent optimization per sub-vector…
  - Complexity of exact solution grows exponentially with M
- They propose:
  - Build the code sequentially by picking the element in the remaining codebooks that most reduces the residual
  - Also use a beam search



["LSQ++: Lower running time and higher recall in multi-codebook quantization", Martinez, et al. ECCV 2018.]

Local search quantization

- Start from an initial (OPQ…)
- Encoding with simulated annealing
  - Change one code, compute loss
  - Accept the new code with some probability
  - Decreasing probability of accepting sub-optimal solution over iterations
- At training time: iterate
  - Encoding training set using current codebook
  - Estimate codebook from codes and vectors → LS problem

$$\min_C \|X - CB\|_F^2;$$

Training vectors (d*n) Codebooks (d * (KM)) codes (KM * n, sparse)

Fast search with additive quantization

- With the decomposition

$$x \approx x' = T_1[i_1] + ... + T_M[i_M]$$

- We can pre-compute

$$\mathrm{LUT}_m[i] = \langle T_m[i], q \rangle \qquad \forall m = 1..M, i = 1..K_m$$

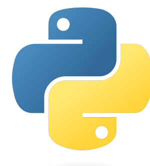
- Which allows to compute the dot product (fast!)

$$\langle q, x \rangle \approx \langle q, x' \rangle = LUT_1[i_1] + ... + LUT_m[i_M]$$

- But not L2 distances…
  - However, we can use

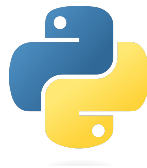
$$||q - x'||^2 = ||q||^2 + ||x'||^2 - 2\langle q, x' \rangle$$

Constant — stored (approx) — computed with LUTs



Scalar quantization

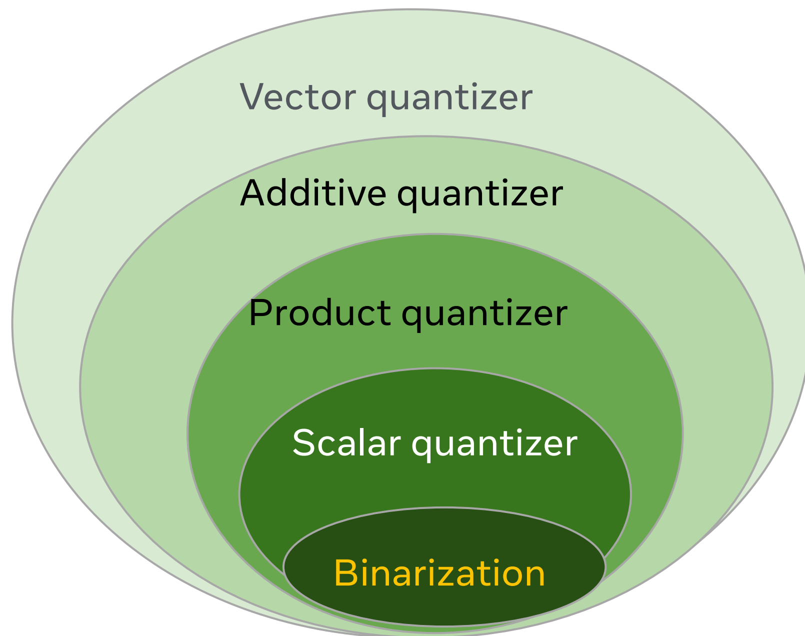
- Uniform scalar quantization
  - Map each dimension to an int8 (or int4…)
- Useful for moderate compression
  - Float32 → int8 or int4 (4x to 8x compression)
- Often used for quasi-lossless storage
- Fast and accurate
- Special case of PQ
  - 1 dimension per component
  - Uniform “centroids”
- Be careful with imbalanced dimensions
  - Eg. output of a PCA



Nesting of quantization methods

- Each quantizer has an assignment rule to a centroid
- Only the full VQ (k-means) represents the centroids explicitly
- Others have implicit centroids with more or less constraints / capacity
- Nested set of quantizers
  - More general – slower, more accurate
  - More specific – faster, less accurate

- In practice:
  - (uniform) scalar quantization loses almost no precision
  - PQ is a good tradeoff to reduce by < 4 bits per dim

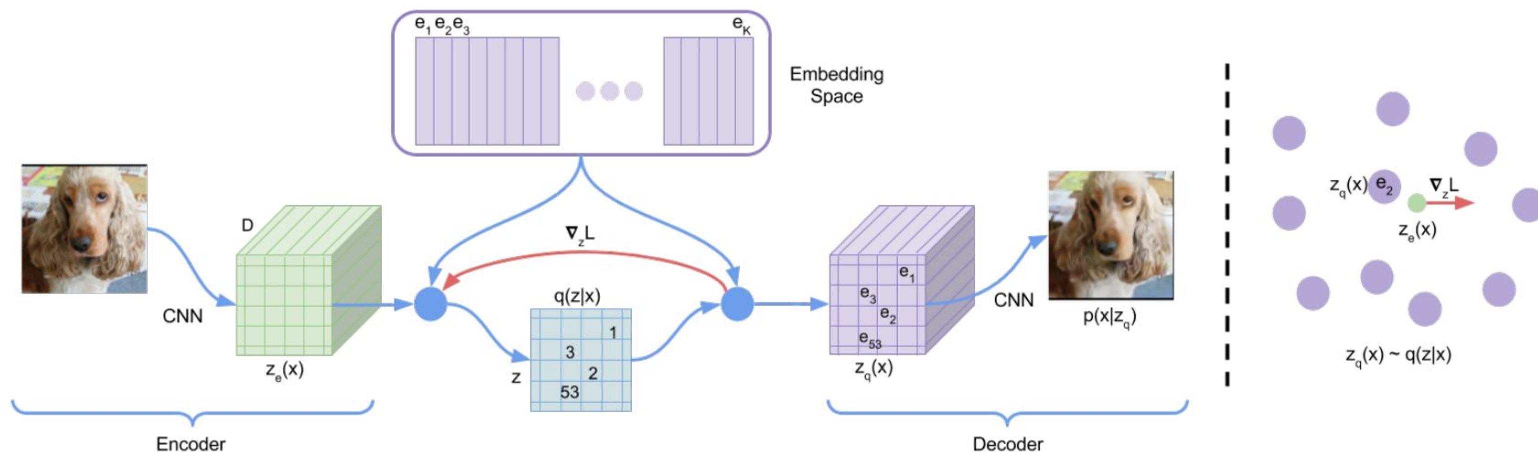


Neural quantization methods

[Neural Discrete Representation Learning, Van den Oord et al, NIPS'17]

Vector Quantised-Variational AutoEncoder

- Auto-encoder with a discrete bottleneck
- Applied to the image domain initially



- At encoding time: nearest centroid assignment

VQ-VAE training

- Loss
  - Uses stop-gradients (sg)
  - To keep some components fixed

$$L = \log p(x|z_q(x)) + \|\text{sg}[z_e(x)] - e\|_2^2 + \beta\|z_e(x) - \text{sg}[e]\|_2^2,$$

Data term

embeddings → codebook

codebook → embeddings

During forward computation the nearest embedding $z_q(x)$ (equation 2) is passed to the decoder, and during the backwards pass the gradient $\nabla_z L$ is passed unaltered to the encoder. Since the output

VQ-VAE results

- Impactful in the image (and other media) domain

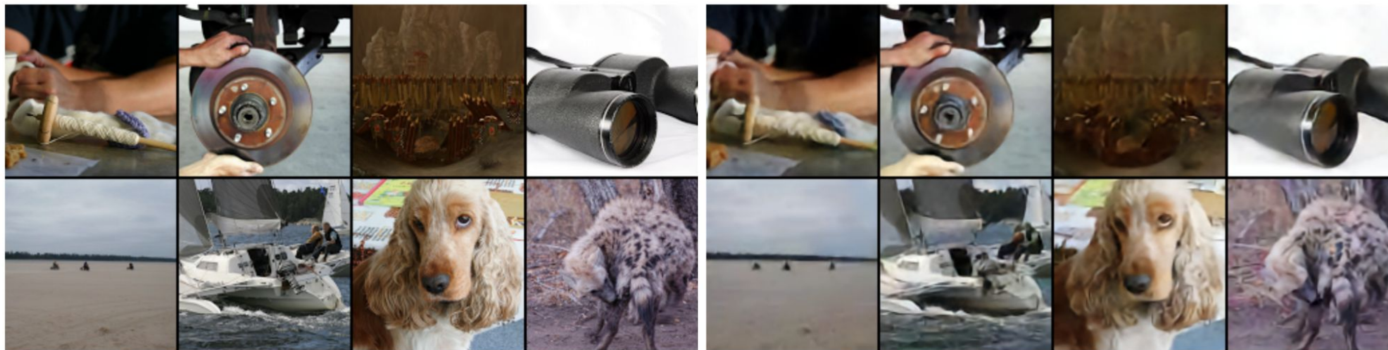


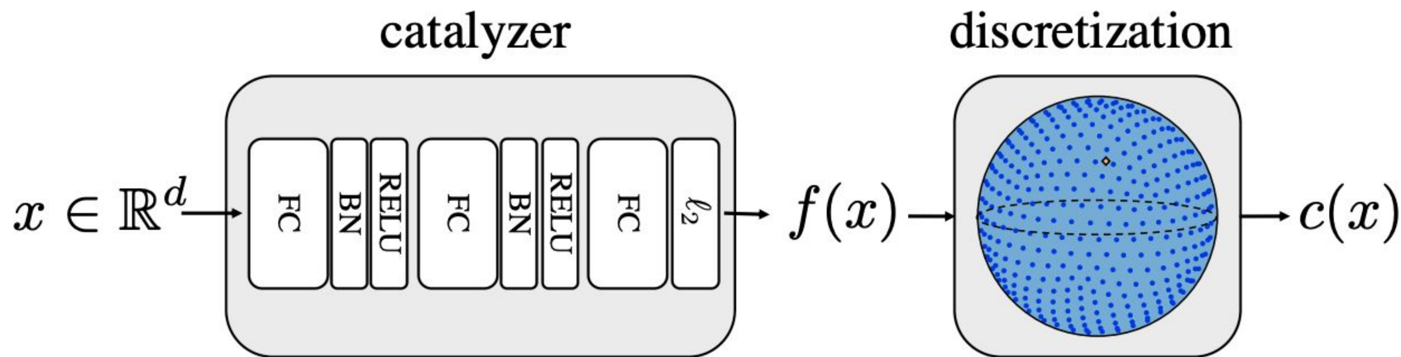
Figure 2: Left: ImageNet 128x128x3 images, right: reconstructions from a VQ-VAE with a 32x32x1 latent space, with K=512.

- In the vector domain, limited impact
  - Not competitive with multi-codebook quantizers
  - Hard to take advantage of an encoder in a space that is already vectorial

[Spreading vectors for similarity search, Sablayrolles et al, ICLR'19]

The catalyzer

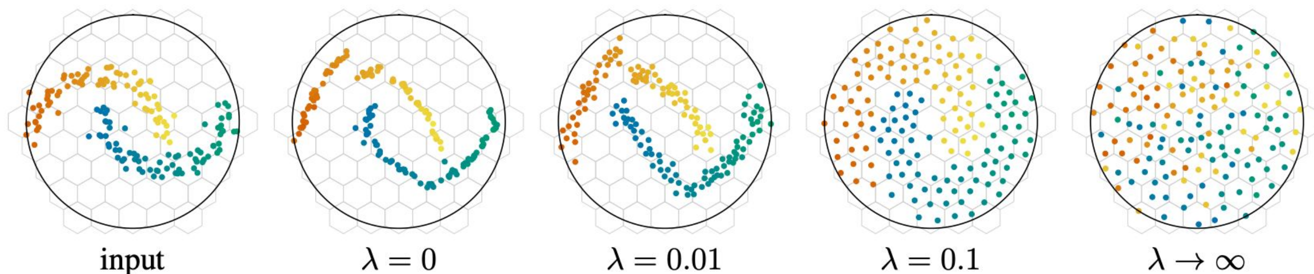
- Transform input vectors to make them easier to index
- Map with trained function
  - Conserves neighborhood relations
- Fixed quantizer
  - We have good quantizers for uniform data
  - Centroids = intersection of the integer grid with a sphere



The catalyzer

- Loss that enforces a uniform output distribution

$$\mathcal{L}_{\text{model}} = \mathcal{L}_{\text{rank}} + \lambda \mathcal{L}_{\text{KoLeo}},$$

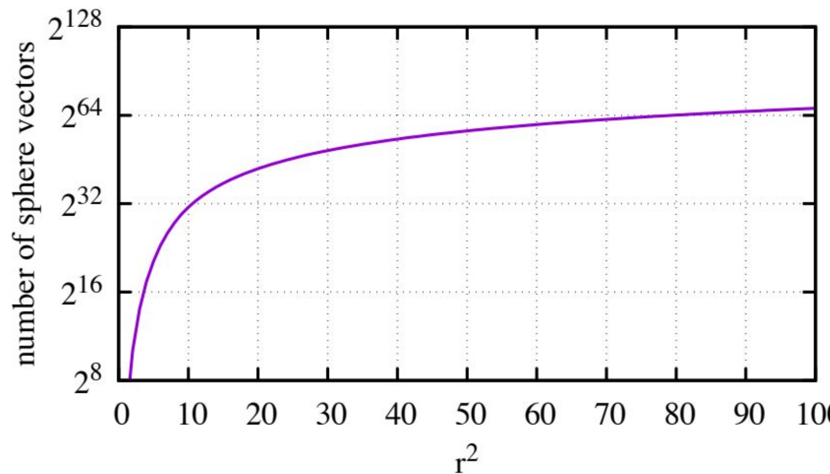


- Local loss based on local entropy estimator (Kozachenko-Leononenko)
  - Based on the log of the distance to the nearest neighbor

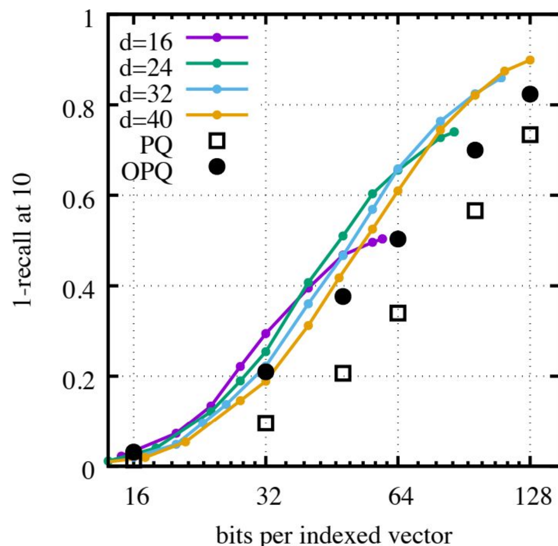
$$\mathcal{L}_{\text{KoLeo}} = -\frac{1}{n} \sum_{i=1}^{n} \log(\rho_{n,i})$$

The catalyzer: lattice quantizer

- Number of centroids as a function of radius in 24 dim
- Assignment is tricky…



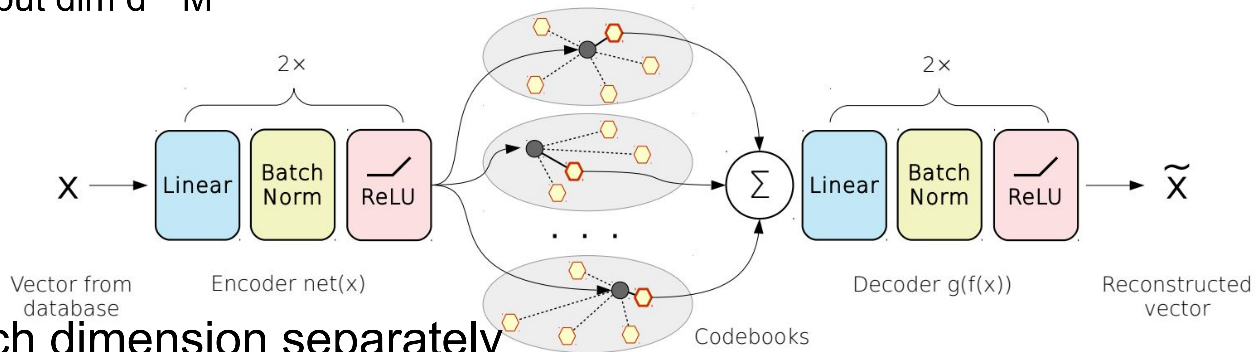
- Good perf w.r.t. PQ



[Morozov & Babenko Unsupervised Neural Quantization for Compressed-Domain Similarity Search, ICCV'19]

Unsupervised Neural Quantization

- Inspired by additive quantizers
- Use a neural net to map to M codebooks
  - Total output dim d * M



- Encode each dimension separately
- Reconstruct with
  - Additive quantizer
  - Decoder network
- Enforce that intermediate additive quantizer gives relatively good neighbors

UNQ training

- Loss

$$L = L_1 + \alpha \cdot L_2 + \beta \cdot \frac{1}{M} \sum_{m=1}^{M} CV^2(i_m)$$

MSE loss　　triplet loss　　avoid code collapse

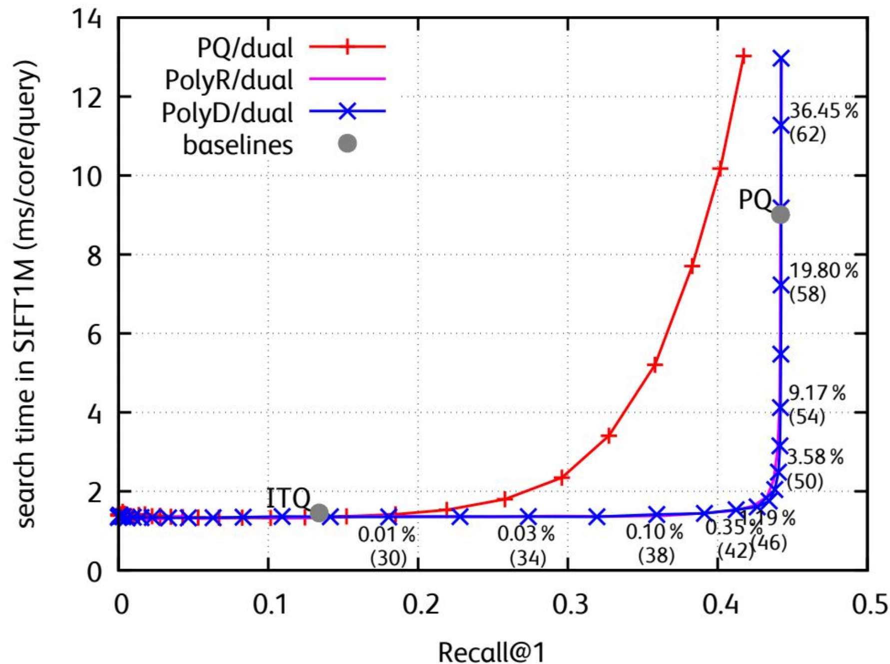
- 2-level search:
  - Search with intermediate representation first: additive, fast
  - Re-rank shortlist with full decoding

Polysemous codes

[Douze et al, Polysemous codes, ECCV'16]

Several decoders

- Family of methods where codes can be decoded in two ways
  - Fast / inaccurate
  - Slow / accurate
- Polysemous codes paper:
  - Fast: binary search
  - Slow: product quantization
  - Speed factor 6
- UNQ:
  - Fast: additive quantizer
  - Slow: full decoding
  - Speed factor 8.5
- At search time
  - Shortlist with fast codes
  - Rerank with slow codes (100-1000)



[Amara et al, Nearest neighbor search with compact codes: A decoder perspective, ICMR'22]

Arbitrary decoders

- Given an encoder
  - Generate codes for a training set
- Fit an arbitrarily large decoder
- Higher-level multi-codebook quantizer
  - cf. the least-squares decoder
- Neural net
  - easy to train
  - no discretization, strong supervision

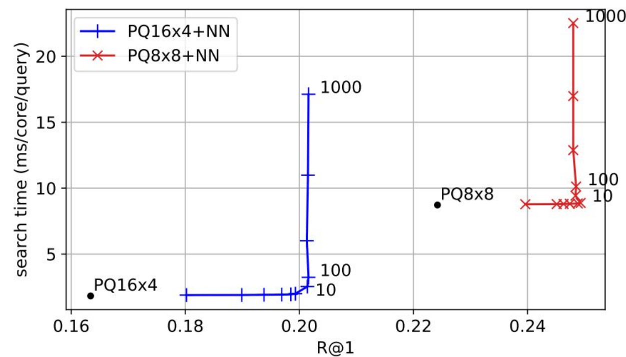
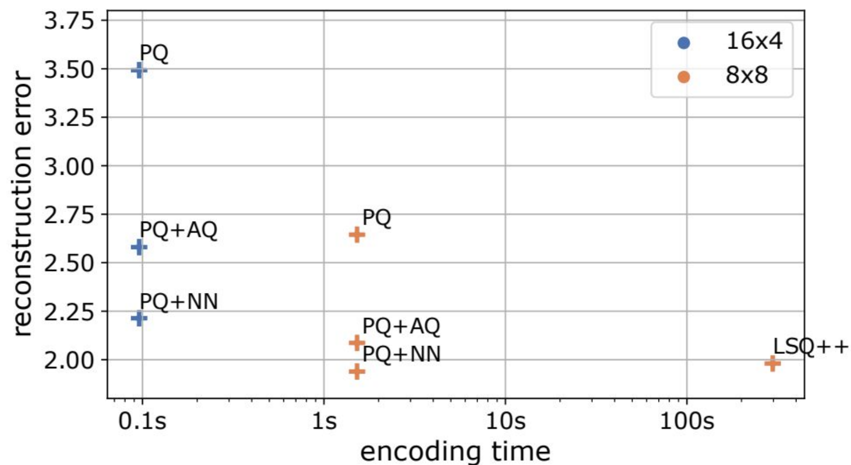


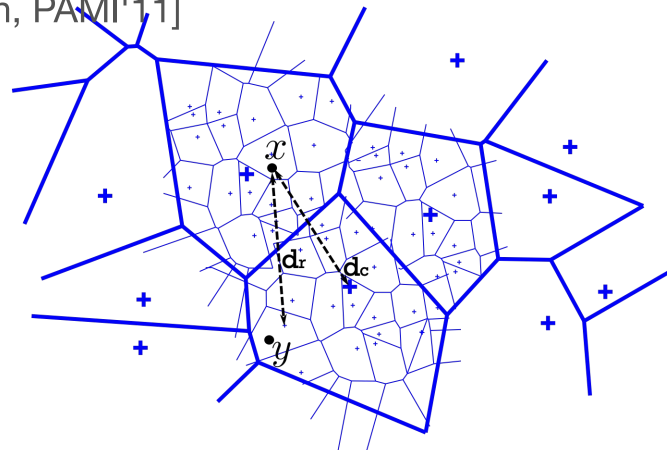
Figure 4: Accuracy vs. search time on the BigANN1M dataset when re-ranking: The NN decoder re-orders the top PQ results.



Practical implementation: IVFPQ

[Jégou & al, Product quantization for nearest neighbor search, PAMI'11]

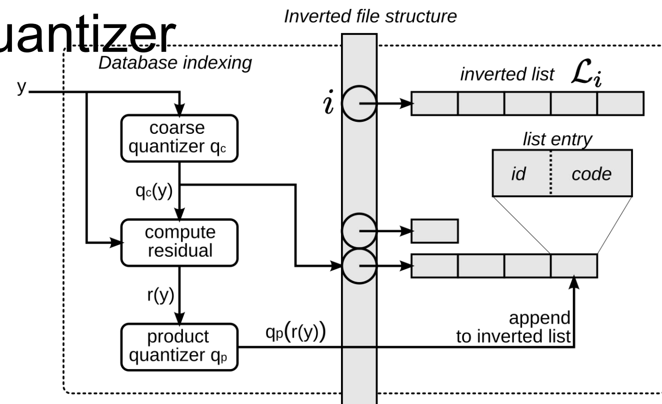
Multiple quantization levels



- Coarse quantizer + residual quantizer
  - reproduction value:

$$y \approx q_\mathrm{c}(y) + q_\mathrm{f}(y - q_\mathrm{c}(y))$$

- coarse = k-means quantizer, fine = product quantizer
  - inverted file structure



- At search time: select subset of centroids
  - within inverted list: normal PQ search with look-up tables

$$\mathrm{argmin}_i \| \underbrace{x - q_\mathrm{c}(y_i)}_{\text{query}} - \underbrace{q_\mathrm{f}(y_i - q_\mathrm{c}(y_i))}_{\text{PQ codes}} \|$$

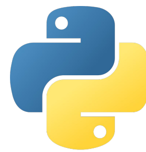
[Johnson & al, , PAMI'11]

At the basis of many implementations

- Faiss IndexIVFPQ
- SCANN
- ANNS search over the coarse quantizer centroids
  - ANNS improvements benefit the search
- Relatively easy to optimize in hardware
  - GPU
  - SIMD

At the basis of many implementations

- Faiss IndexIVFPQ
- SCANN
- ANNS search over the coarse quantizer centroids
  - ANNS improvements benefit the search
- Relatively easy to optimize in hardware
  - GPU
  - SIMD



["Quicker ADC : Unlocking the Hidden Potential of Product Quantization with SIMD", André et al, PAMI'19]

SIMD implementation of search

- PQ search bottleneck: memory look-ups
- Store look-up tables in registers

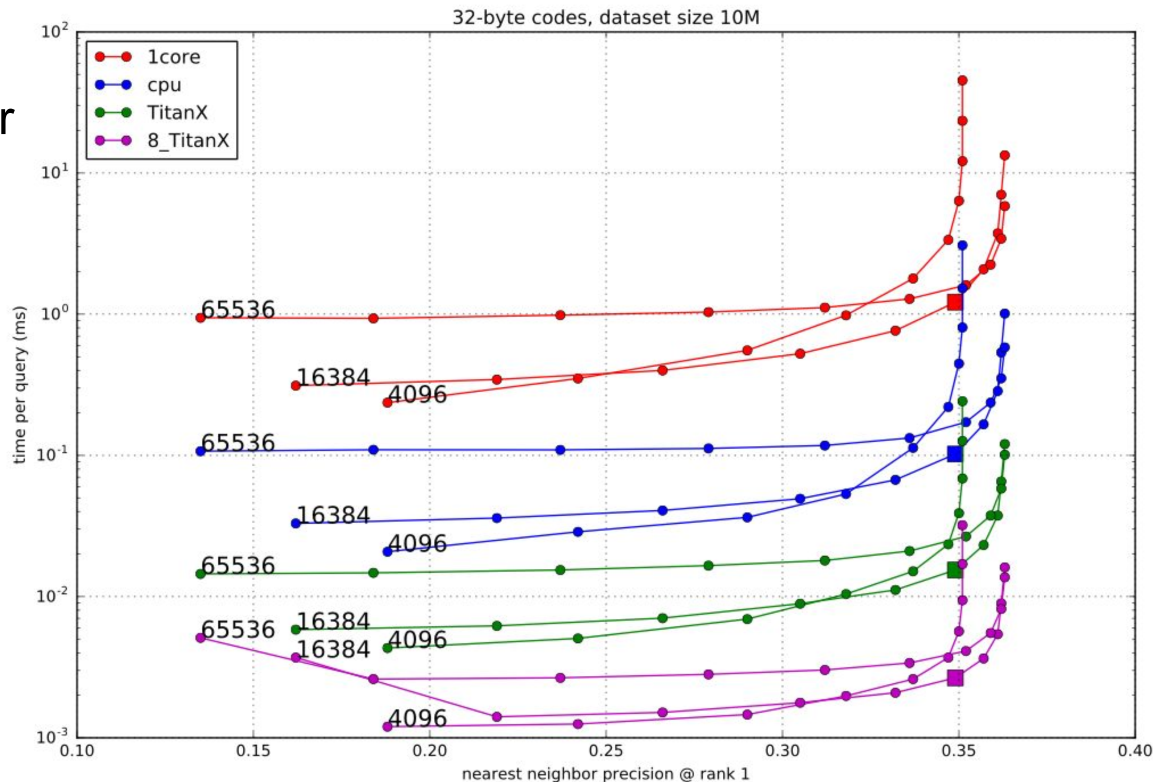
	quantizer m																quantizer m + 1															
address (bytes)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
bits 0..3	**0**	**8**	**1**	**9**	**2**	**10**	**3**	**11**	**4**	**12**	**5**	**13**	**6**	**14**	**7**	**15**	**0**	**8**	**1**	**9**	**2**	**10**	**3**	**11**	**4**	**12**	**5**	**13**	**6**	**14**	**7**	**15**
bits 4..7	**16**	**24**	**17**	**25**	**18**	**26**	**19**	**27**	**20**	**28**	**21**	**29**	**22**	**30**	**23**	**31**	**16**	**24**	**17**	**25**	**18**	**26**	**19**	**27**	**20**	**28**	**21**	**29**	**22**	**30**	**23**	**31**

- Requires very small codebooks: log(K) = 4 bits
- Fast / inaccurate
  - Useful to do re-ranking

[Billion-scale similarity search with GPUs. Johnson et al, Trans big data, 2021]

GPU search

- Particularly efficient for brute force search
- Fast arithmetic
- High mem bandwidth
- BUT high latencies
  - Hard to implement branching algorithms



Conclusion

- Vector compression is required for large scale
- Always keep an eye on traeoffs
  - There are always a few “active” constraints

- Next class By Harsha on graph indexes
  - Very efficient and versatile
  - Use lots of memory → compression is useful!

END