Billion-Scale Nearest Neighbor Search

CVPR 2023 Tutorial on **Neural Search in Action**, Part 2

Martin Aumüller

IT University of Copenhagen, maau@itu.dk

JUNE 18-22, 2023
CVPR
VANCOUVER, CANADA

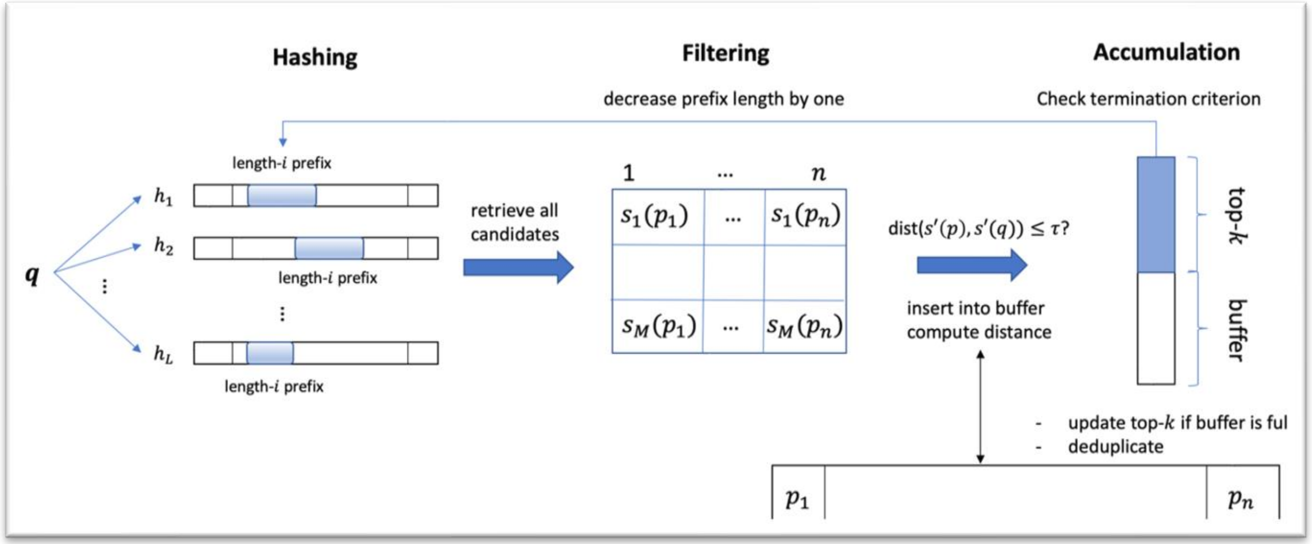
IT UNIVERSITY OF CPH

Martin Aumüller

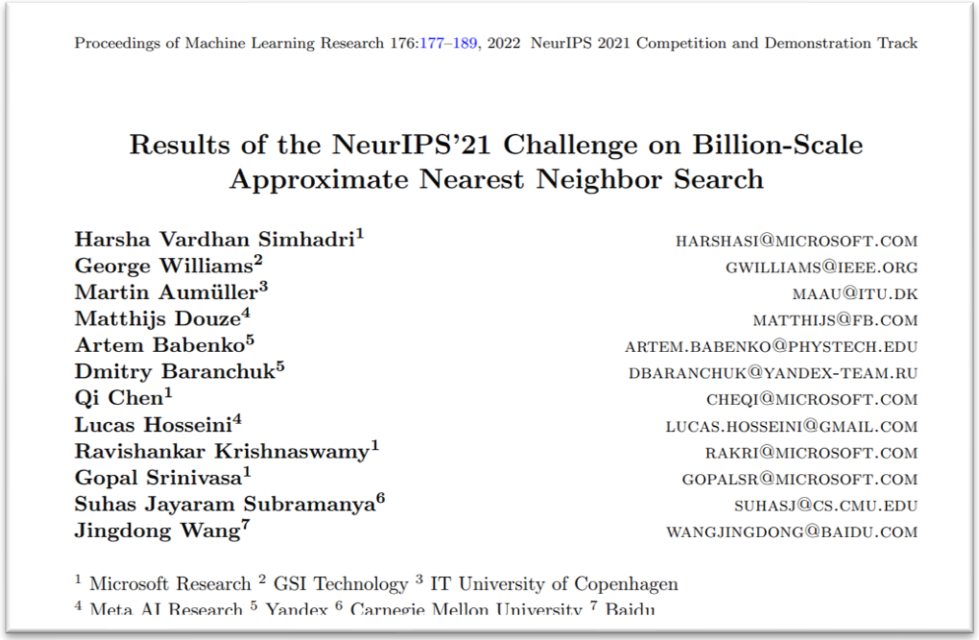
Associate Professor, IT University of Copenhagen, Denmark

http://itu.dk/people/maau @maumueller

- ✓ Similarity search using hashing
- ✓ Benchmarking & workload generation



PUFFINN
[Aumüller+, ESA 2019]



Proceedings of Machine Learning Research 176:177–189, 2022 NeurIPS 2021 Competition and Demonstration Track

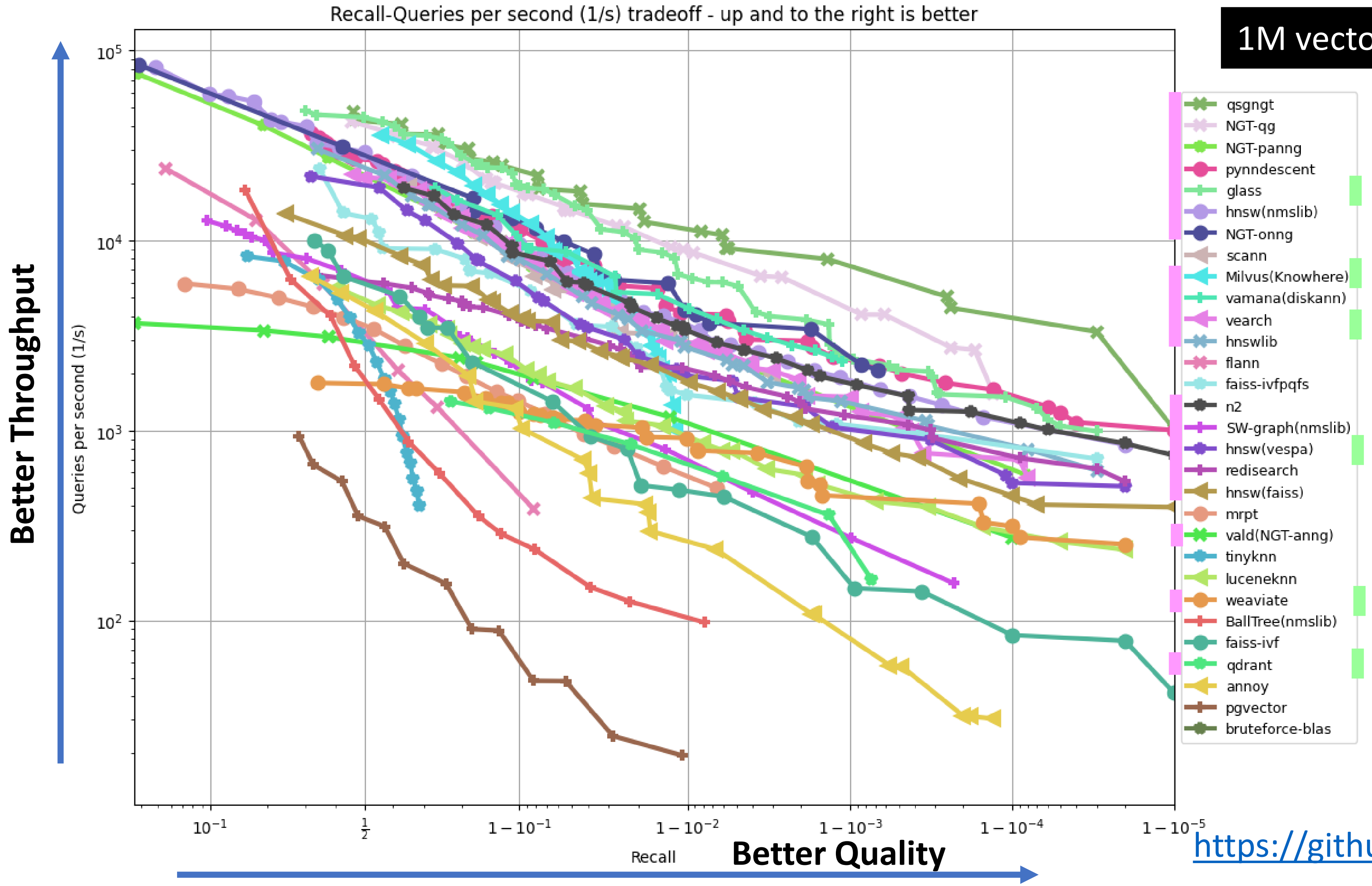
Results of the NeurIPS'21 Challenge on Billion-Scale Approximate Nearest Neighbor Search

Harsha Vardhan Simhadri[1]	HARSHASI@MICROSOFT.COM
George Williams[2]	GWILLIAMS@IEEE.ORG
Martin Aumüller[3]	MAAU@ITU.DK
Matthijs Douze[4]	MATTHIJS@FB.COM
Artem Babenko[5]	ARTEM.BABENKO@PHYSTECH.EDU
Dmitry Baranchuk[5]	DBARANCHUK@YANDEX-TEAM.RU
Qi Chen[1]	CHEQI@MICROSOFT.COM
Lucas Hosseini[4]	LUCAS.HOSSEINI@GMAIL.COM
Ravishankar Krishnaswamy[1]	RAKRI@MICROSOFT.COM
Gopal Srinivasa[1]	GOPALSR@MICROSOFT.COM
Suhas Jayaram Subramanya[6]	SUHASJ@CS.CMU.EDU
Jingdong Wang[7]	WANGJINGDONG@BAIDU.COM

[1] Microsoft Research [2] GSI Technology [3] IT University of Copenhagen
[4] Meta AI Research [5] Yandex [6] Carnegie Mellon University [7] Baidu

Billion-Scale ANN Challenge
[Aumüller+, NeurIPS 21, Competition]

From Million-Scale to Billion-Scale ANN



1M vectors, GloVe word embeddings

Graph-based

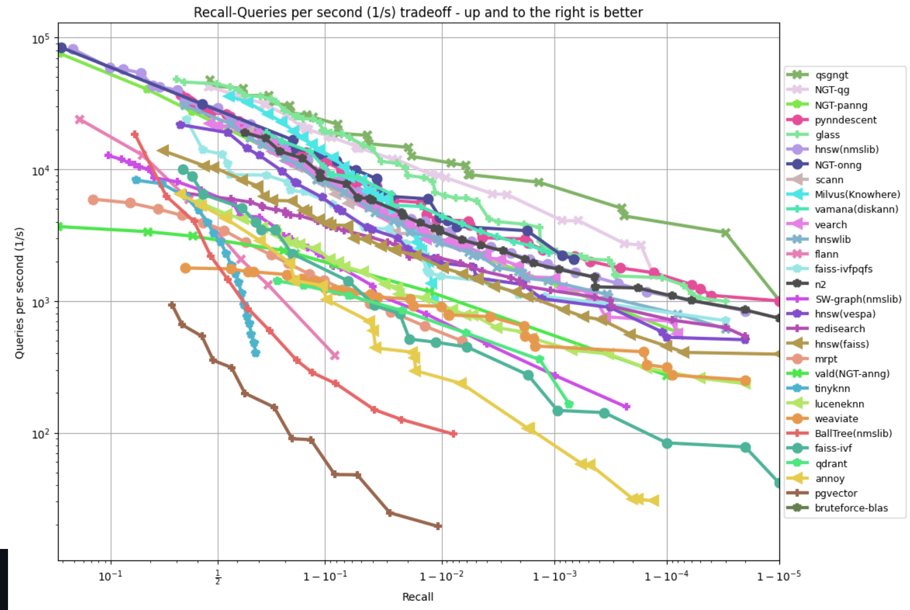
Service

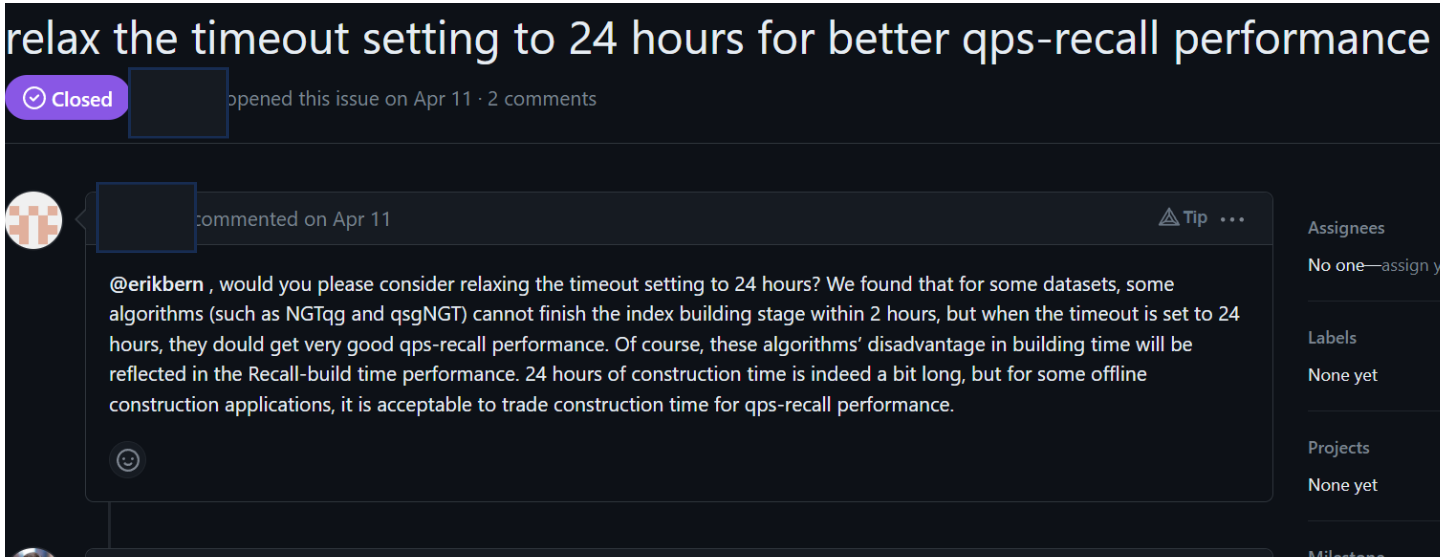
https://github.com/erikbern/ann-benchmarks

From Million-Scale to Billion-Scale ANN

Rules

- Index building + searching **single-threaded**
- **2 hours time limit**, container killed afterwards

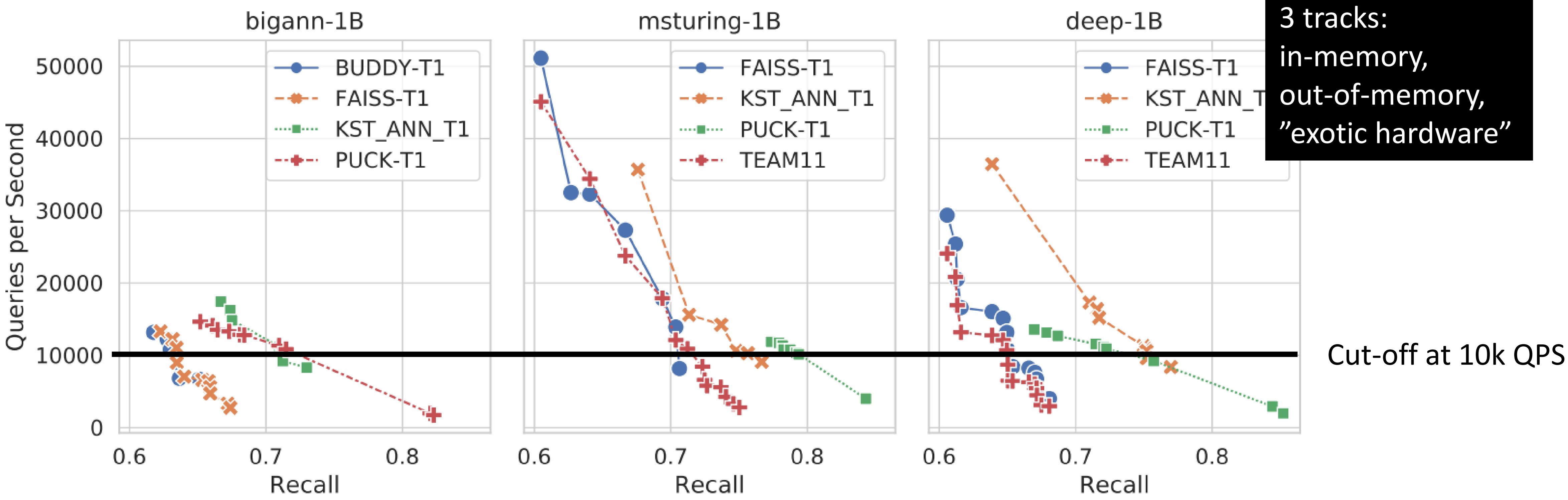




Q: Scaling up by 1000x?

2 hours → 2000 hours ~ 83 days
24 hours → 24000 hours ~ 3 years
(<u>unrealistic scaling</u>)

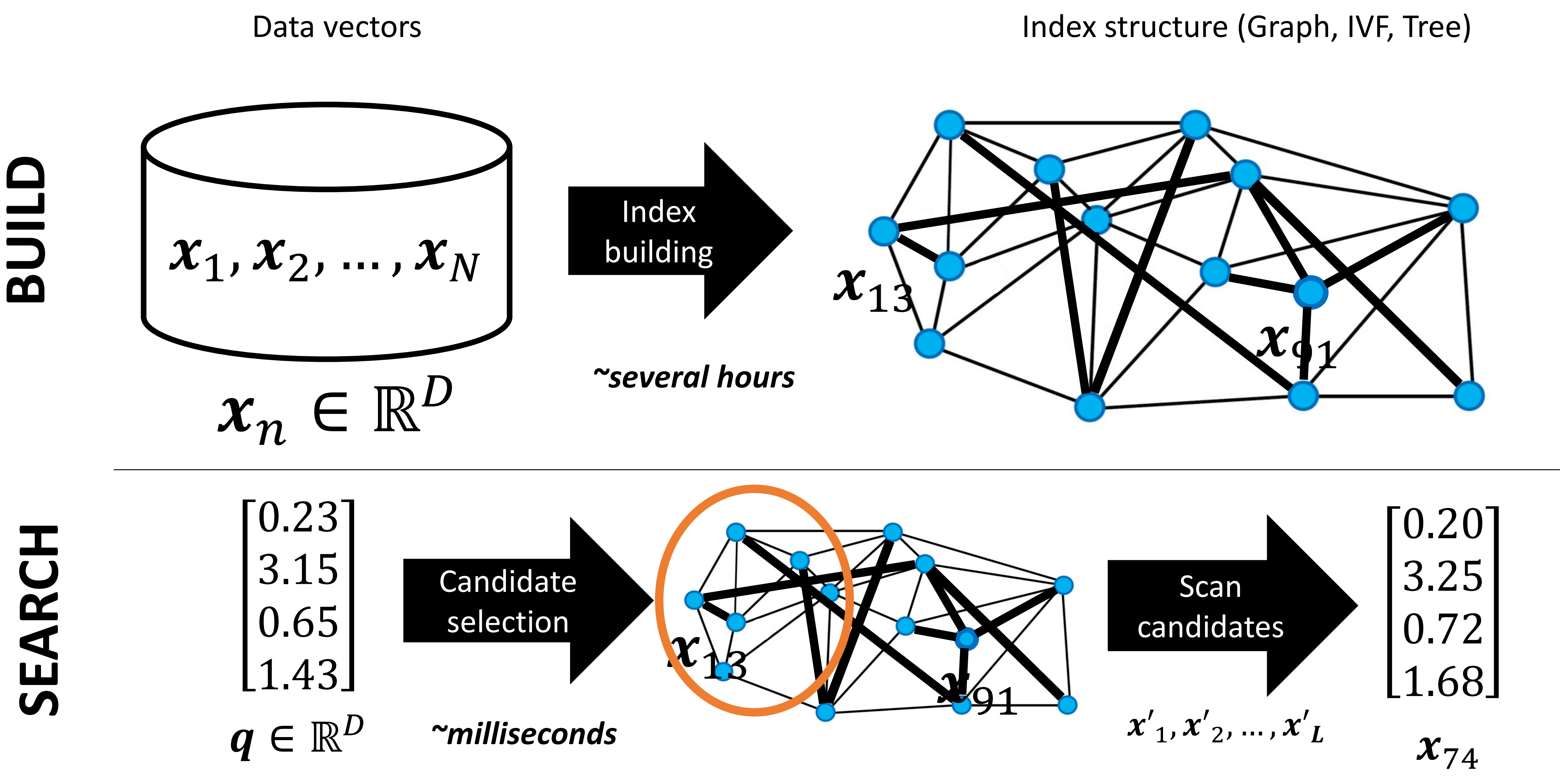
Billion-Scale ANN Challenge [Simhadri+, NeurIPS 2021]



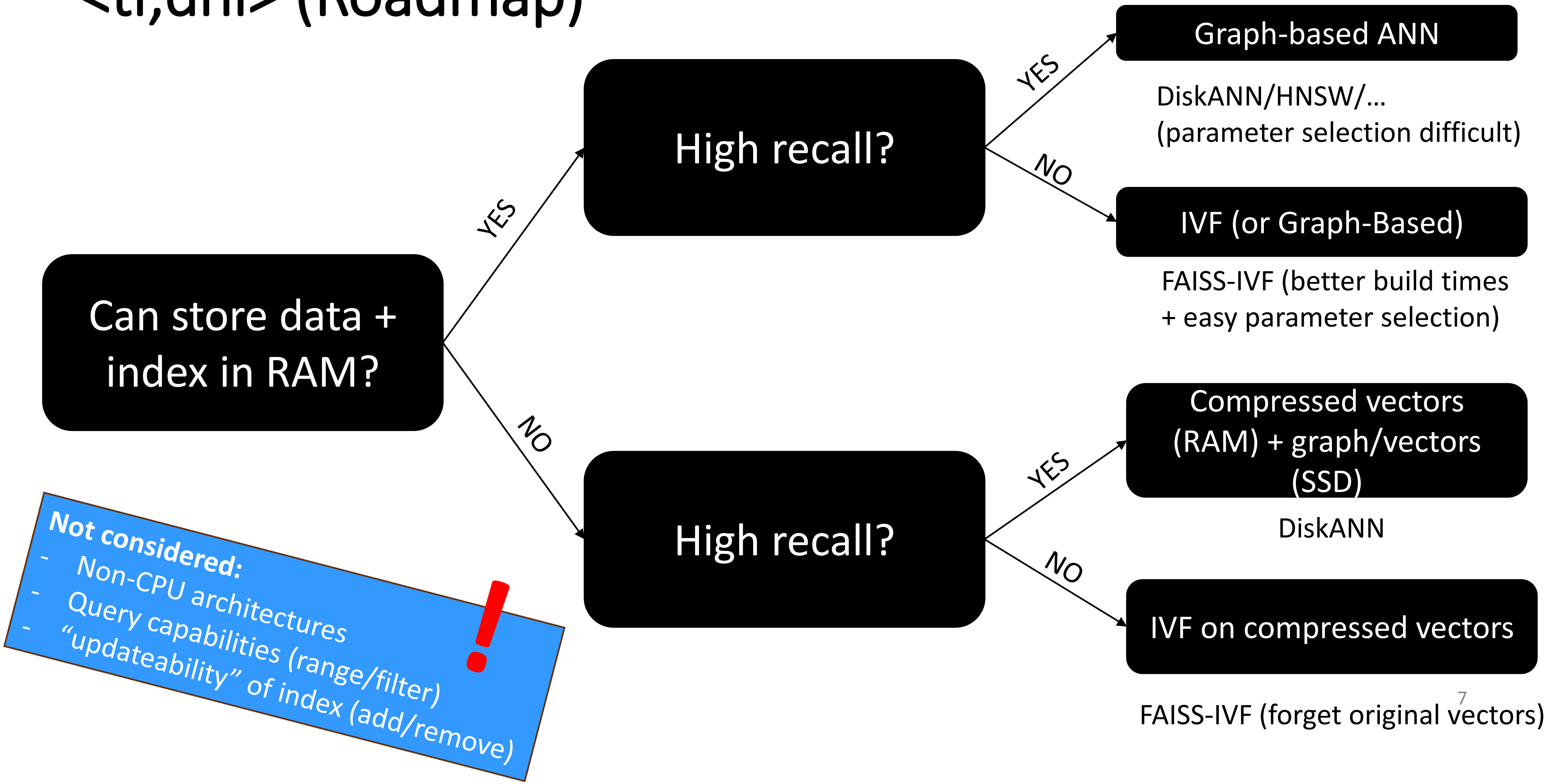
3 tracks:
in-memory,
out-of-memory,
"exotic hardware"

Many entries did not improve on baseline by much.

The ANN search pipeline



<tl;dnl> (Roadmap)



Billion-Scale Datasets

Meta AI: Image descriptors for copy detection

Dataset	Datatype	Dimensions	Distance	Range/k-NN	Base data	Sample data	Query data	Ground truth	Release terms
BIGANN	uint8	128	L2	k-NN	1B points	100M base points	10K queries	link	CC0
Facebook SimSearchNet++*	uint8	256	L2	Range	1B points	N/A	100k queries	link	CC BY-NC
Microsoft Turing-ANNS*	float32	100	L2	k-NN	1B points	N/A	100K queries	link	link to terms
Microsoft SPACEV*	int8	100	L2	k-NN	1B points	100M base points	29.3K queries	link	O-UDA
Yandex DEEP	float32	96	L2	k-NN	1B points	350M base points	10K queries	link	CC BY 4.0
Yandex Text-to-Image*	float32	200	inner-product	k-NN	1B points	50M queries	100K queries	link	CC BY 4.0

256 GB

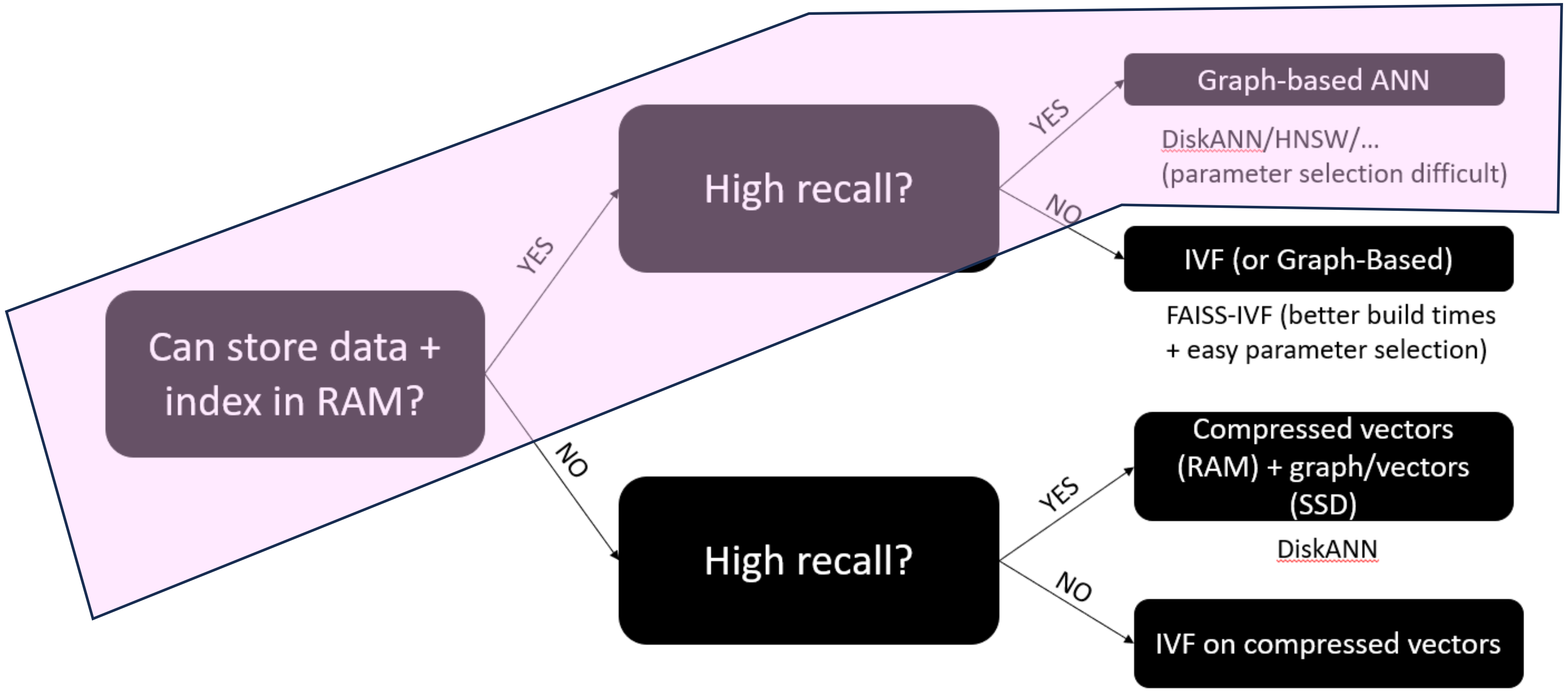
100 GB

800 GB

Microsoft Bing: Search string → Web documents

https://big-ann-benchmarks.com/

NeurIPS 2021 Challenge



High Resources, High Recall

Possible setup: Multi-Socket Xeon, 256 GB - 2TB of RAM

Scaling Graph-Based Approaches

Scaling Graph-Based ANNS Algorithms to Billion-Size Datasets: A Comparative Analysis

Magdalen Dobson
Carnegie Mellon University
mrdobson@cs.cmu.edu

Zheqi Shen
UC Riverside
zshen055@ucr.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Laxman Dhulipala
University of Maryland
laxman@umd.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Harsha Vardhan Simhadri
Microsoft Research
harshasi@microsoft.com

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

s.IR] 7 May 2023

Abstract

Algorithms for approximate nearest-neighbor search (ANNS) have been the topic of significant recent interest in the research community. However, evaluations of such algorithms are usually restricted to a small number of datasets with millions or tens of millions of points, whereas real-world applications require algorithms that work on the scale of billions of points. Furthermore, existing evaluations of ANNS algorithms are typically heavily focused on measuring and optimizing for queries-per-second (QPS) at a given accuracy, which can be hardware-dependent and ignores important metrics such as build time.

Solving this problem is known as *k-nearest neighbor search*, and is notoriously hard to solve exactly in high-dimensional spaces [18]. Since solutions for most real-world applications can tolerate small errors, most deployments focus on the *approximate nearest neighbor search (ANNS)* problem, which has been widely applied as a core subroutine in fields such as search recommendations, machine learning, and information retrieval [68]. Modern applications are placing new demands on ANNS data structures to be scalable to billions of points [61], support streaming insertions and deletions [42, 62, 66], work on a wide variety of difficult datasets [43], and support efficient nearest neighbor queries as well as range

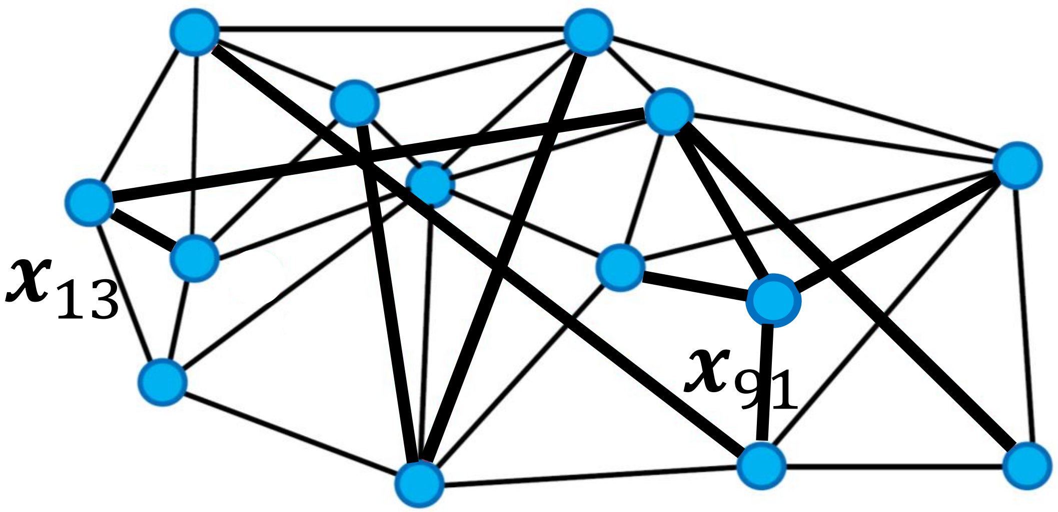
Machines

- Azure Msv2 (4 Xeon, 192 vCPUs, 2 TB RAM), $384 USD/day
- Azure Ev5 (2 Xeon, 96 vCPUs, 672 GB RAM), $144 USD/day

https://arxiv.org/pdf/2305.04359.pdf

Scaling Graph-Based Approaches

- **Recap**
  - Vectors are nodes
  - Connected to "diverse set of similar points" + long range edges
- **Incremental build**
  - Use search algorithm to find potential candidate neighbors
  - Prune these candidates



Index size?

~1B x "avg. degree of node"

Practically all algorithms enforce user-set bound!

Faster build?

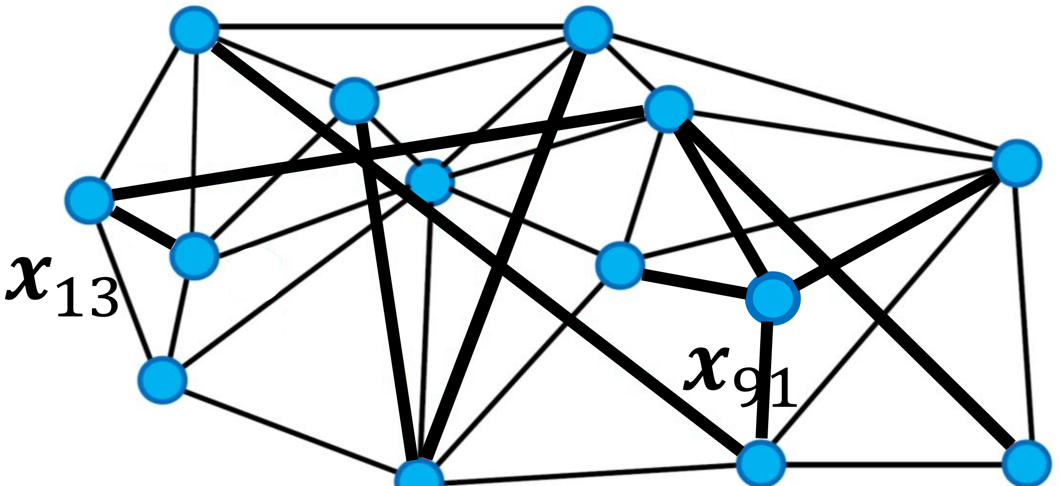
Smaller target degree + smaller beam width

Tradeoffs?

Need larger beam width to compensate for "worse build graph"

Parallelizing insertion

- Order all points arbitrarily
- For each point:
  - Carry out greedy search for nearest neighbor in “current graph”
  - Connect to *pruned* set of vertices found during the NN search



Algorithm 2: insert(p, s, R, L).

Input: Point p, starting point s, beam width L, degree bound R.
Output: Point p is inserted into the nearest neighbor graph.

$\mathcal{V}, \mathcal{K} \leftarrow$ greedySearch$(p, s, L, 1)$
$\mathrm{N_{out}}(p) \leftarrow$ prune$(\mathcal{V})$
for $q \in N_{out}(p)$ **do**
  $\mathrm{N_{out}}(q) \leftarrow \mathrm{N_{out}}(q) \cup \{p\}$
  if $|N_{out}(q)| > R$ **then**
    $\mathrm{N_{out}}(q) \leftarrow$ prune$(\mathrm{N_{out}}(q))$

Thread-safety?

Algorithm 3: batchBuild$(\mathcal{P}, s, R, L)$.

Input: Point set $\mathcal{P}$, starting point s, beam width L, degree bound R.
Output: A nearest neighbor graph consisting of all points in $\mathcal{P}$ and start point s.

$i \leftarrow 0$
while $2^i \leq |\mathcal{P}|$ **do**
  parallel for $j \in [2^i, 2^{i+1})$ **do**
    $\mathcal{V}, \mathcal{K} \leftarrow$ greedySearch$(\mathcal{P}[j], s, L)$
    $\mathrm{N_{out}}(\mathcal{P}[j]) \leftarrow$ prune$(\mathcal{V})$
  $\mathcal{B} \leftarrow \bigcup_{j=2^i}^{2^{i+1}-1} \mathrm{N_{out}}(P[j])$
  parallel for $b \in \mathcal{B}$ **do**
    // Find $\mathcal{N}$ as all points in the current batch that added b as their neighbors
    $\mathcal{N} \leftarrow \{\mathcal{P}[j] \mid j \in [2^i, 2^{i+1}) \wedge b \in \mathrm{N_{out}}(\mathcal{P}[j])\}$
    $\mathrm{N_{out}}(b) \leftarrow \mathrm{N_{out}}(b) \cup \mathcal{N}$
    if $|N_{out}(b)| > R$ **then** $\mathrm{N_{out}}(b) \leftarrow$ prune$(\mathrm{N_{out}}(b))$
  $i \leftarrow i + 1$

“prefix doubling”

Understanding parameters

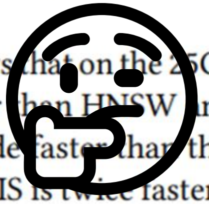
- **Index building**
  - Degree bound R
    - upper limit on index size
  - Beam width L (building)
    - better neighbors
  - Pruning factor (α)
    - ”diversified neighbors”
- **Searching**
  - Beam width R_{search}

Sensitive to parameter choices & they are difficult to choose!

DiskANN The main parameters for the DiskANN index build are (1) the degree bound R, (2) the beam width L used during insertion, and (3) the pruning parameter α. In our experiments, we found that no single parameter setting was optimal for all recall regimes, and that there were significant tradeoffs in other recall values when maximizing for recall above .99; thus we chose to use parameters optimized for the .94-.97 range. Note that for TEXT2IMAGE, which minimizes negative inner product, the α value must be less than one in order to select for a denser graph.

1-million experiments. Due to scalability issues, we could not report results on the 25GB experiments for HCNNG (indexing time exceeded 24 hours) and KGRAPH/DPG (could not reach an acceptable accuracy, i.e., recall > 0.8). Due to the low performance on the 25GB experiments of VAMANA and EFANNA (indexing a 25GB dataset required over 300GB RAM and indexing a 100GB dataset needed more than the 1.4TB of available memory) and NSG (since it uses EFANNA as a base graph), we excluded them from experiments with larger datasets.

Indexing Time. Figure 1 shows that on the 25GB dataset, ELPIS can build its index 2x and 5x faster than HNSW and NSG, respectively, and over an order of magnitude faster than the other competitors. On the other dataset sizes, ELPIS is twice faster than its second best competitor, HNSW. Since NSG [50] is built on top of EFANNA [48], we include the time to build both indexing structures. Although VAMANA [111] builds the graph based on a random initial graph, it spends more than 7 hours to create the Deep25GB index. This is



https://www.vldb.org/pvldb/vol16/p1548-azizi.pdf

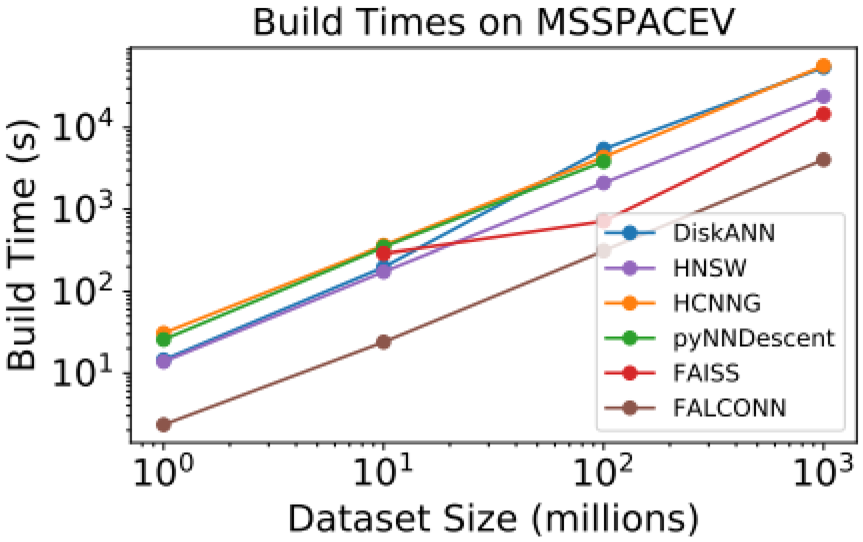
Build times & scaling

	BIGANN	MSSPACEV	TEXT2IMAGE	SSNPP
DiskANN	$R = 64, L = 128, \alpha = 1.2$	$R = 64, L = 128, \alpha = 1.2$	$R = 64, L = 128, \alpha = .9$	$R = 150, L = 400, \alpha = 1.2$

Degree bound

Beam width

Billion-scale: Index size not more than 4R GB (e.g., 256GB, 600GB)



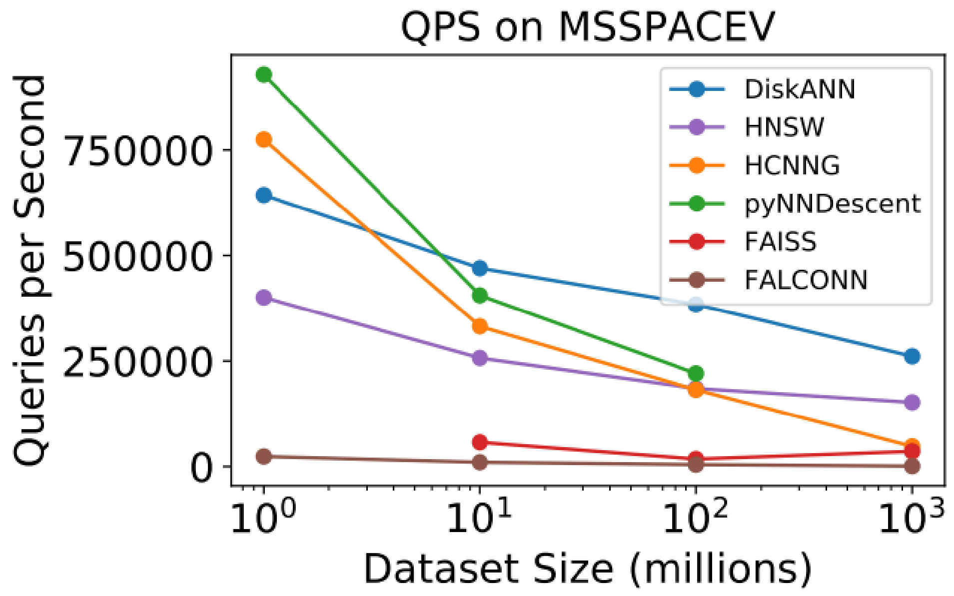
	BIGANN	MSSPACEV	TEXT2IMAGE	SSNPP
DiskANN	11.0	15.1	61.6	83.1
HNSW	9.2	6.7	14.9	91.6
HCNNG	8.6	15.8	21.4	19.0
FAISS	5.2	4.1	4.5	4.5
FALCONN	1.75	1.12	1.45	1.42

Table 1: Build times (hours) on billion-scale datasets.

10x increase → 11-12x build time increase

Parallelizing search

- Usually parallelization over queries (inter-query parallelism)
- Not so much in focus
- Beam width selection: “trial-and-error”

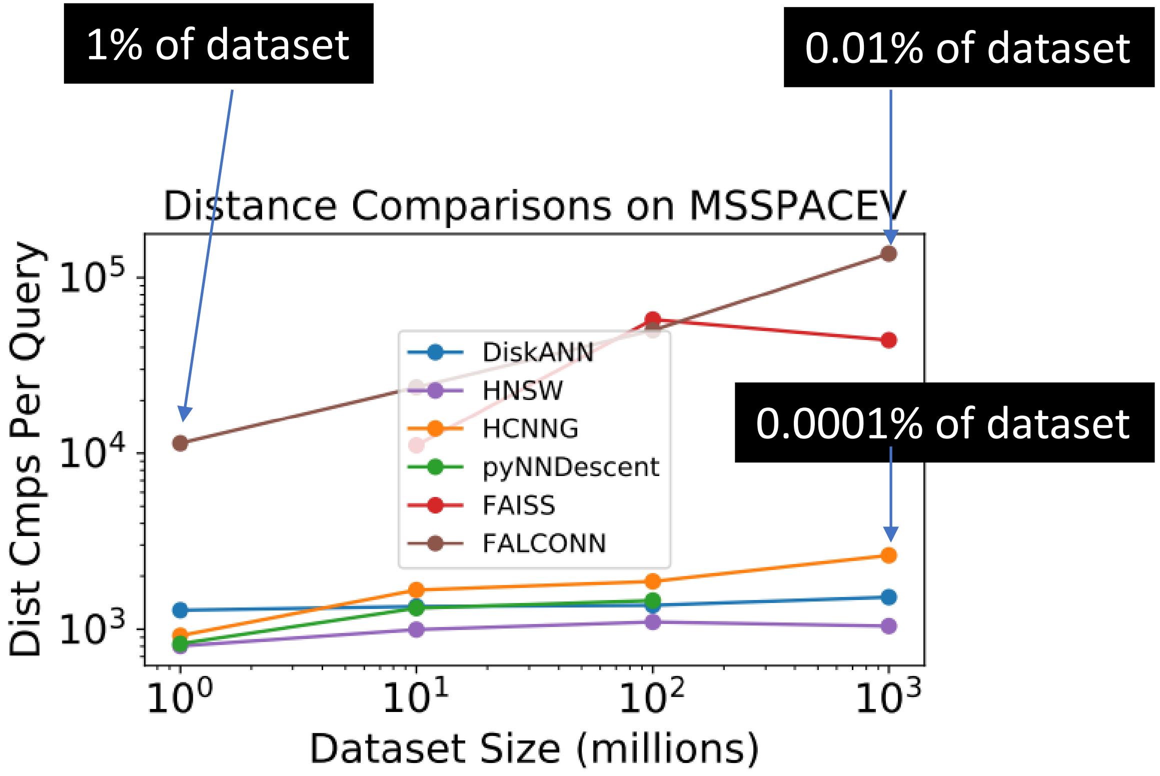


(b) QPS for fixed recall (.8) as dataset size increases.

Scaling: dataset 1000x larger → queries 2x slower

Summary

- **Advantages**
  - Good scaling of *#candidates*
  - Unparalleled performance in high-recall regime
- **Disadvantages**
  - Influence of parameter choices difficult to predict
  - High index building times (but “almost out-of-box”)



(c) Distance comparisons per query for fixed recall (.8) as the dataset size increases.

How to get started (DiskANN)

[DiskANN: Simhadri+, NeurIPS19]

```
FROM ubuntu:jammy


RUN apt update
RUN apt install -y software-properties-common
RUN add-apt-repository -y ppa:git-core/ppa
RUN apt update
RUN DEBIAN_FRONTEND=noninteractive apt install -y git
make cmake g++ libaio-dev libgoogle-perftools-dev
libunwind-dev clang-format libboost-dev
libboost-program-options-dev libmkl-full-dev
libcpprest-dev python3.10

RUN git clone https://github.com/microsoft/DiskANN.git
WORKDIR /home/app/DiskANN
RUN pip3 install virtualenv build
RUN python3 -m build
RUN pip install dist/diskannpy-0.5.
0-cp310-cp310-linux_x86_64.whl
WORKDIR /home/app
```

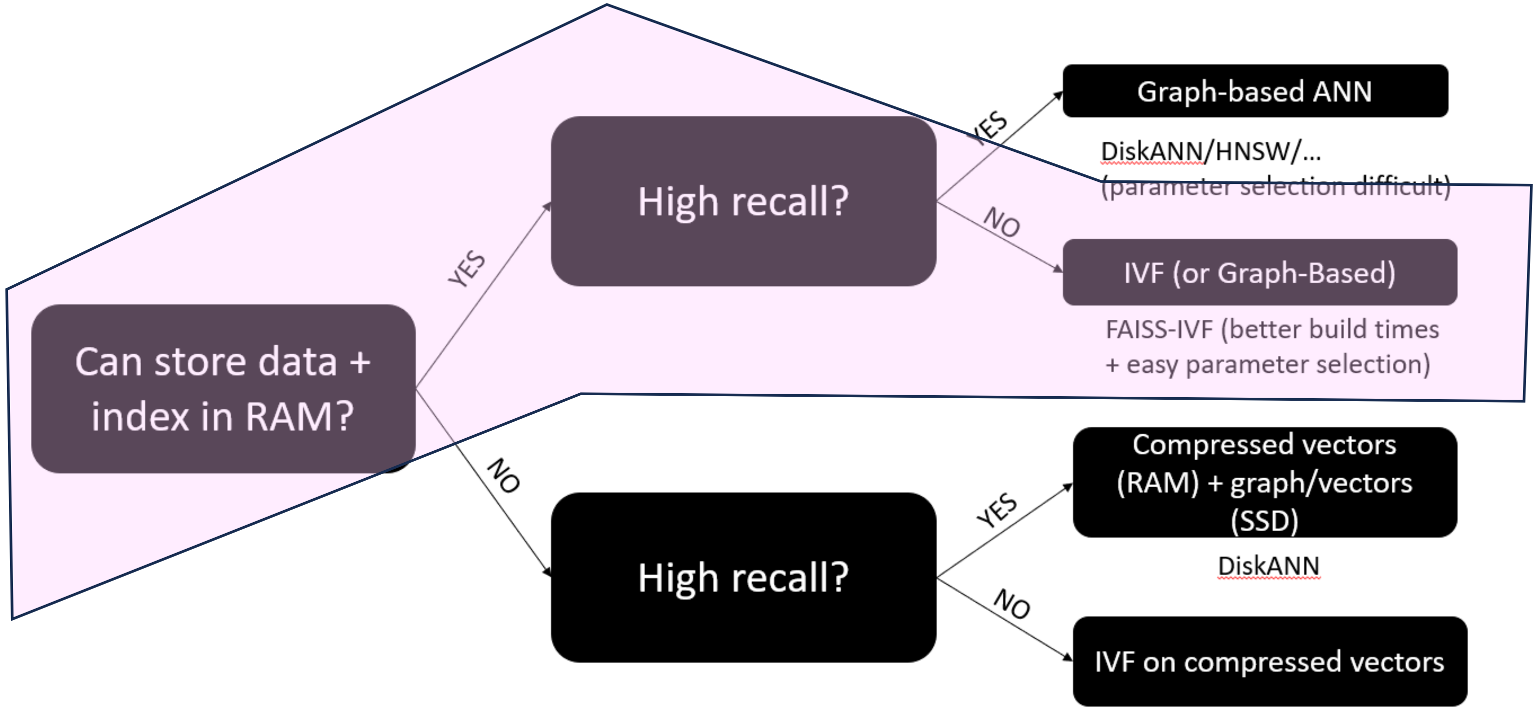
```python
import numpy as np
import diskannpy

class diskann:
    def fit(self, ds, L, R):
        """Build index for dataset `ds` with `R` degree, `L` beam width."""
        diskannpy.build_memory_index(
            data = ds.get_dataset_fn(),
            distance_metric = 'l2',
            vector_dtype = np.int8,
            complexity=L,
            graph_degree=R,
            num_threads = 64,
            alpha=1.2,
            use_pq_build=False,
            num_pq_bytes=0, #irrelevant given use_pq_build=False
            use_opq=False
        )

        print('Loading index..')
        self.index = diskannpy.StaticMemoryIndex(
            distance_metric = 'l2',
            vector_dtype = np.int8,
            num_threads = 64, #to allocate scratch space for up to 64 search threads
            initial_search_complexity = 100
        )
        print('Index ready for search')

    def query(self, X, k, Ls):
        """Carry out a batch query for k-NN of query set X."""
        self.res, self.query_dists = self.index.batch_search(X, k, Ls, 64)
```

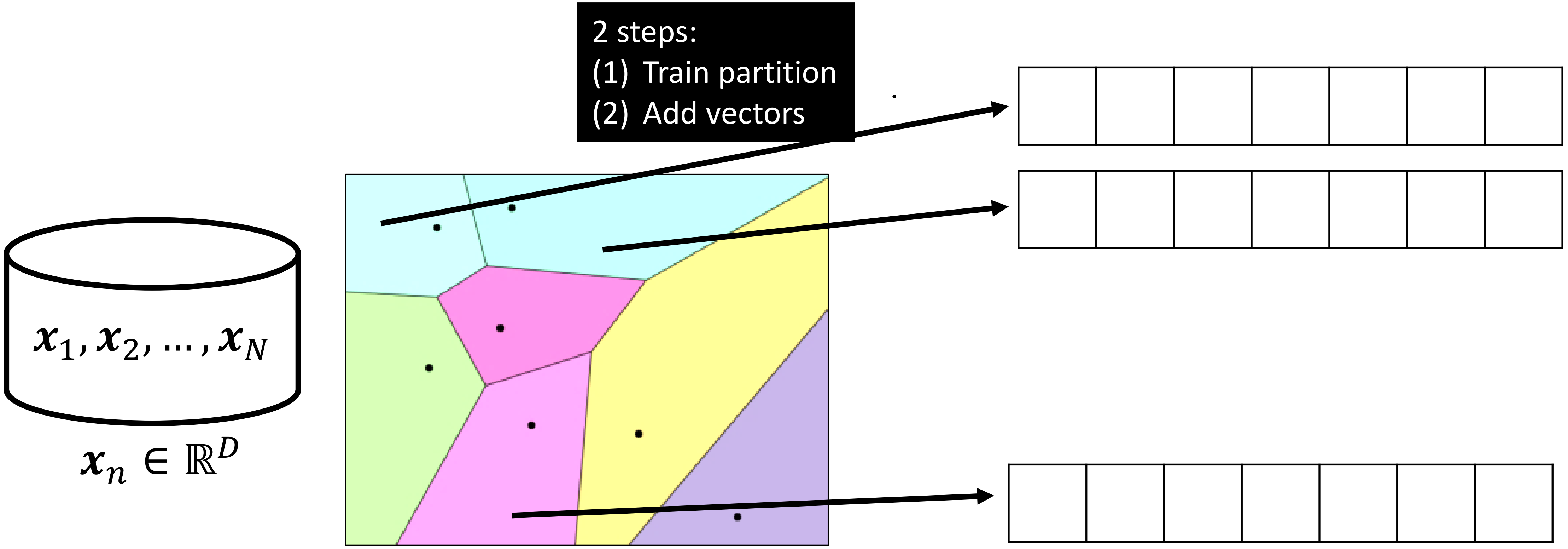
Official Documentation: https://github.com/Microsoft/DiskANN
Python examples: https://github.com/harsha-simhadri/big-ann-benchmarks,



High Resources, Low Recall

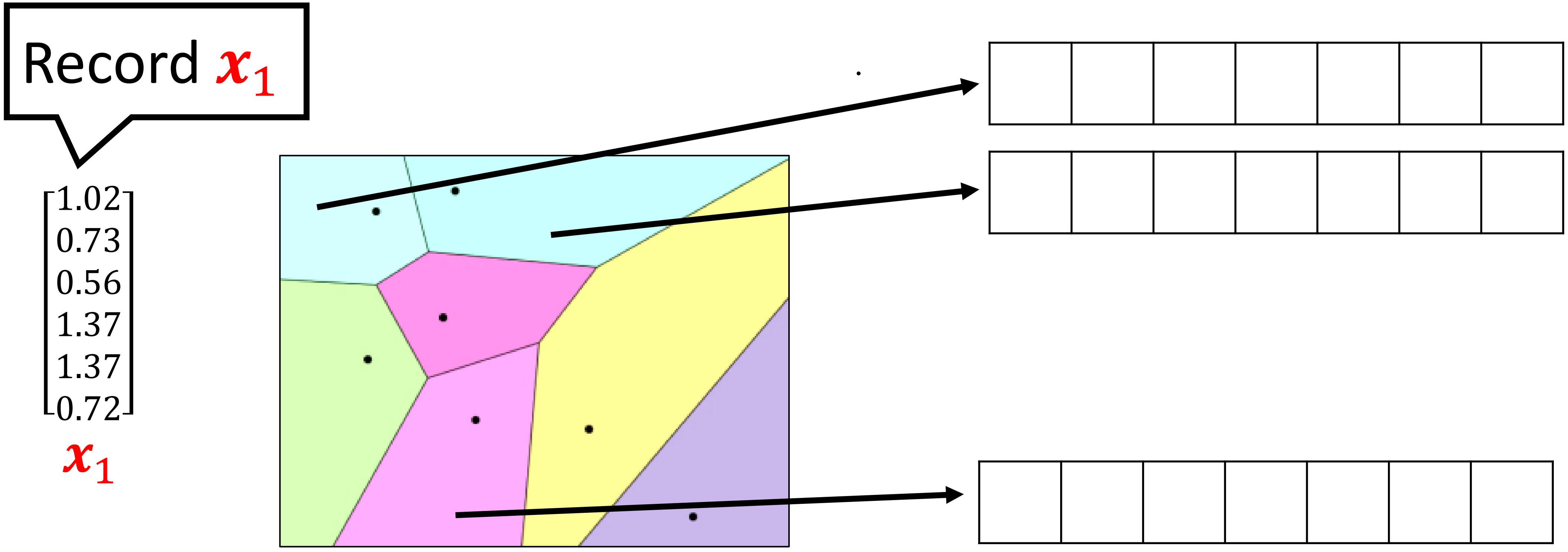
Possible setup: Multi-Socket Xeon, 256 GB - 2TB of RAM

IVF-based solutions ("inverted file index")



Finding a space partition: <u>Clustering-based (k-means)</u>, LSH-based, …

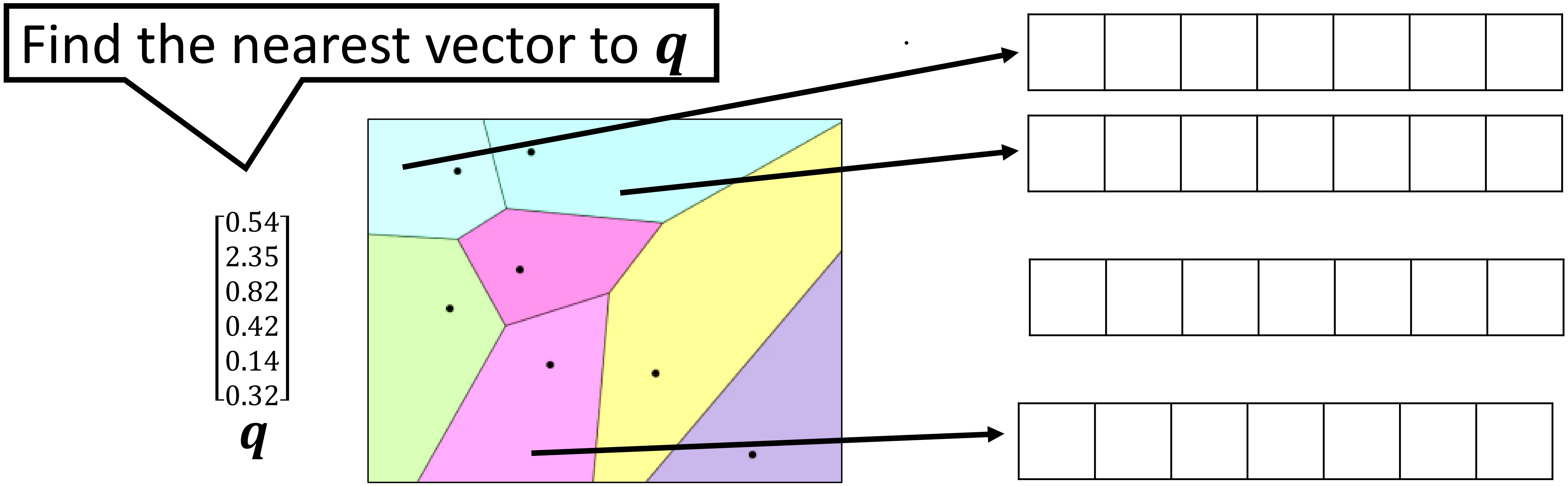
IVF: insert a vector



Cells: all points closest to given centroid ("Voronoi cells")

Build parameter: #clusters

IVF: search

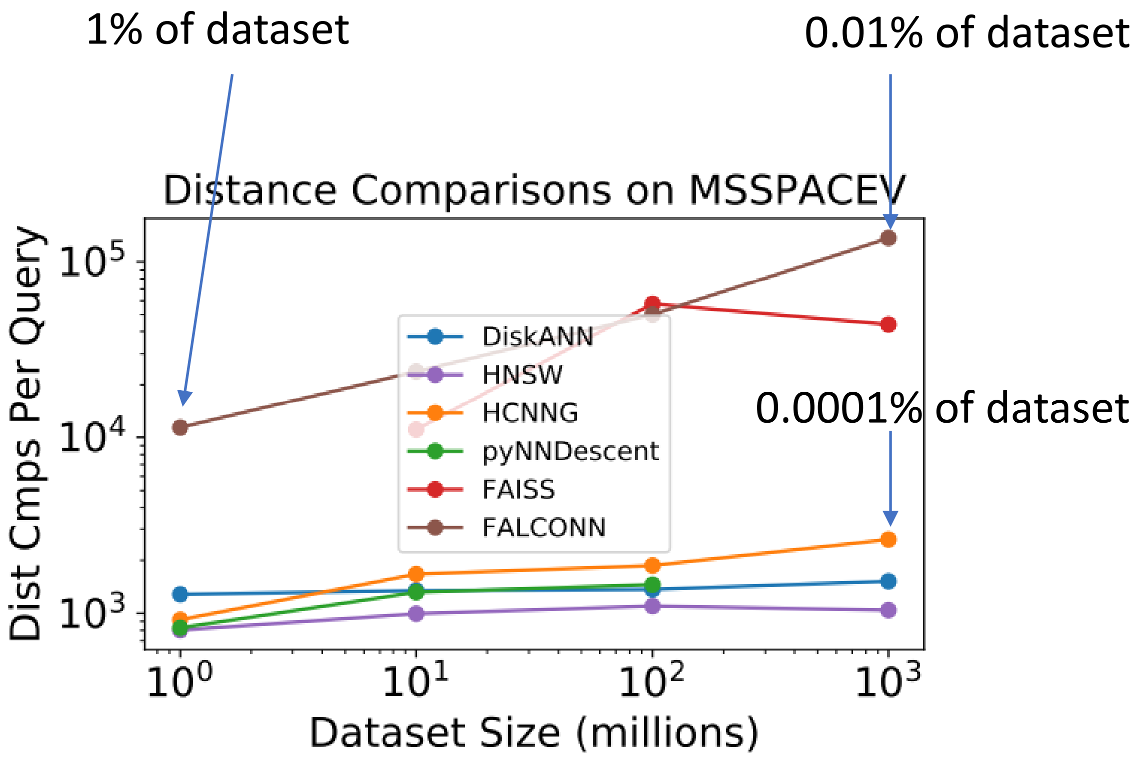


Search parameter: #clusters to inspect

Candidates: #clusters inspected * avg. cluster size

How to choose parameters?

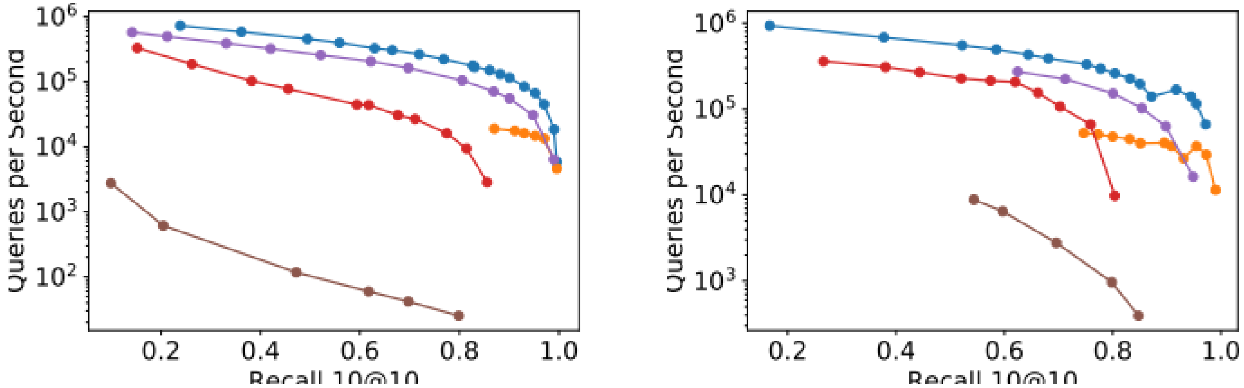
- **Goal**: inspect 0.0001% of dataset for 1B vectors ➔ 1000 points
- **Back-of-the-envelope calculation:**
  - ~1000 points per cluster
  - → need a million clusters 🤔
- **Making this practical**
  - Build an index on centroids
- **Standard solution**
  - Build a graph on top of the centroids
  - Alternatives: hierarchical k-means



(c) Distance comparisons per query for fixed recall (.8) as the dataset size increases.

IVF-based approaches

- **Advantages**
  - Predictable index size and relatively easy to understand parameters
  - Strong implementations available
  - GPU-based solutions
- **Disadvantages**
  - Many candidates necessary in the high-recall regime
  - Quantization necessary to limit impact of these distance computations



(a) BIGANN-1B (b) MSSPACEV-1B

Great documentation with code examples!
https://github.com/facebookresearch/faiss/wiki

How to get started?

- Install via `conda install -c pytorch faiss-cpu`

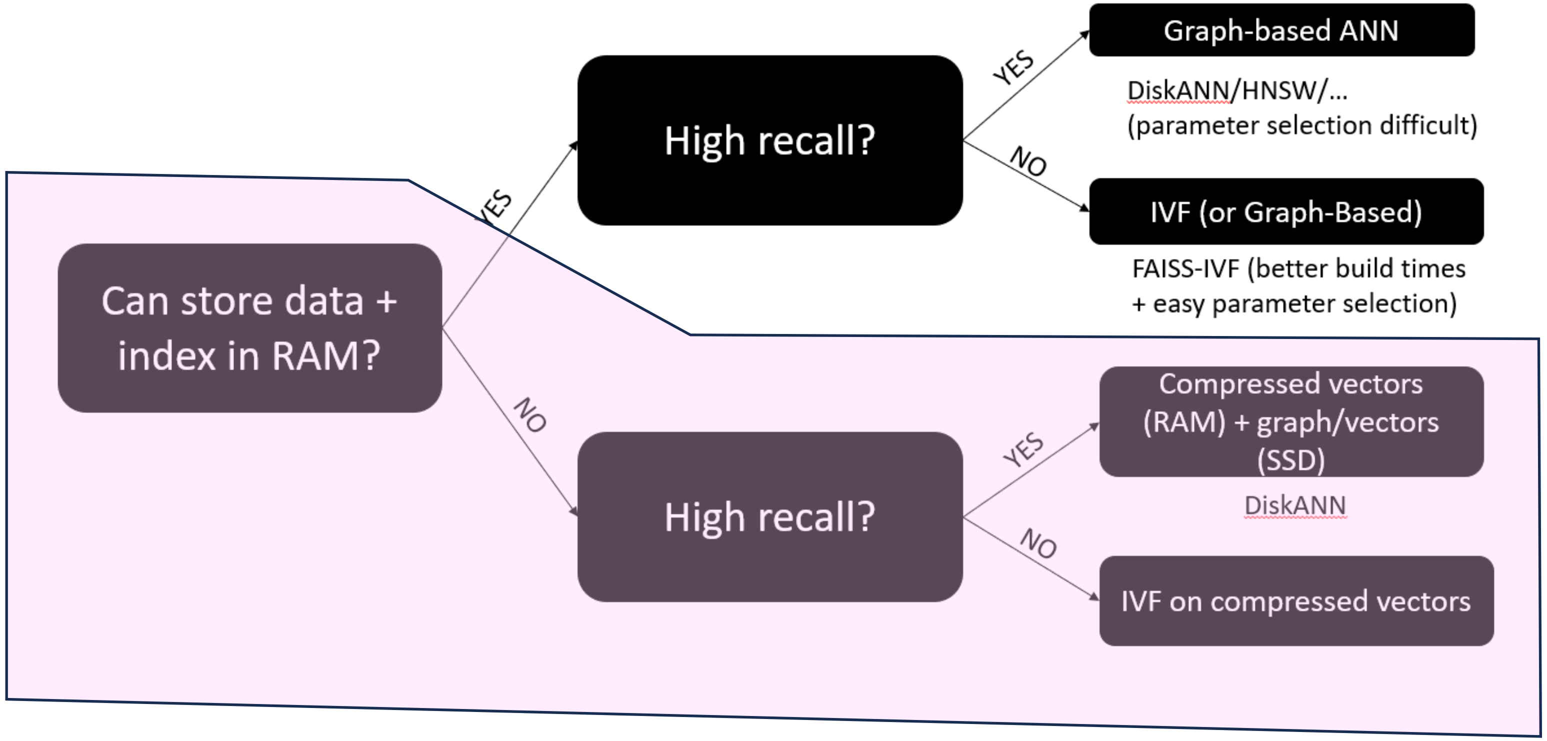
```
nlist = 100
k = 4
quantizer = faiss.IndexFlatL2(d)  # the other index
index = faiss.IndexIVFFlat(quantizer, d, nlist)
assert not index.is_trained
index.train(xb)
assert index.is_trained

index.add(xb)                  # add may be a bit slower as well
D, I = index.search(xq, k)     # actual search
print(I[-5:])                  # neighbors of the 5 last queries
index.nprobe = 10              # default nprobe is 1, try a few more
D, I = index.search(xq, k)
print(I[-5:])                  # neighbors of the 5 last queries
```

`index = faiss.index_factory(128, "PCA64,IVF16384_HNSW32,Flat")`

Index factories available!

https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index



Billion-Scale ANN with limited resources

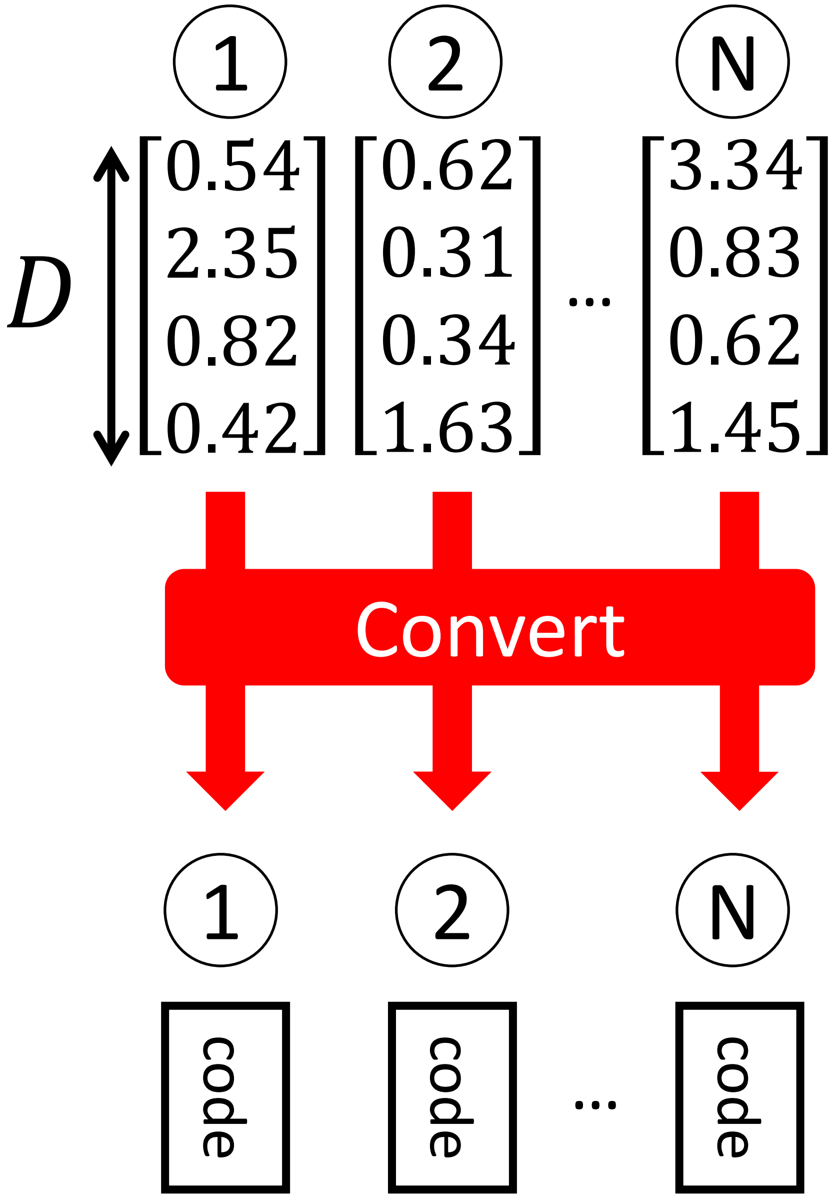
Interlude: Vector Quantization

Quantization techniques

	BIGANN	MSSPACEV	TEXT2IMAGE	SSNPP
DiskANN	$R = 64, L = 128, \alpha = 1.2$	$R = 64, L = 128, \alpha = 1.2$	$R = 64, L = 128, \alpha = .9$	$R = 150, L = 400, \alpha = 1.2$
HNSW	$m = 32, efc = 128, \alpha = .82$	$m = 32, efc = 128, \alpha = .83$	$m = 32, efc = 128, \alpha = 1.1$	$m = 75, efc = 400, \alpha = .82$
HCNNG	$T = 30, Ls = 1000, s = 3$	$T = 50, Ls = 1000, s = 3$	$T = 30, Ls = 1000, s = 3$	$T = 50, Ls = 1000, s = 3$
pyNNDescent	$K = 40, Ls = 100,$ $T = 10, \alpha = 1.2$	$K = 60, Ls = 100,$ $T = 10, \alpha = 1.2$	$K = 60, Ls = 100,$ $T = 10, \alpha = .9$	$K = 60, Ls = 1000,$ $T = 10, \alpha = 1.4$
FAISS	OPQ64_128, IVF1048576_HNSW32, PQ128x4fsr	OPQ64_128, IVF1048576_HNSW32, PQ64x4fsr	OPQ64_128, IVF1048576_HNSW32, PQ128x4fsr	OPQ64_128, IVF1048576_HNSW32, PQ64

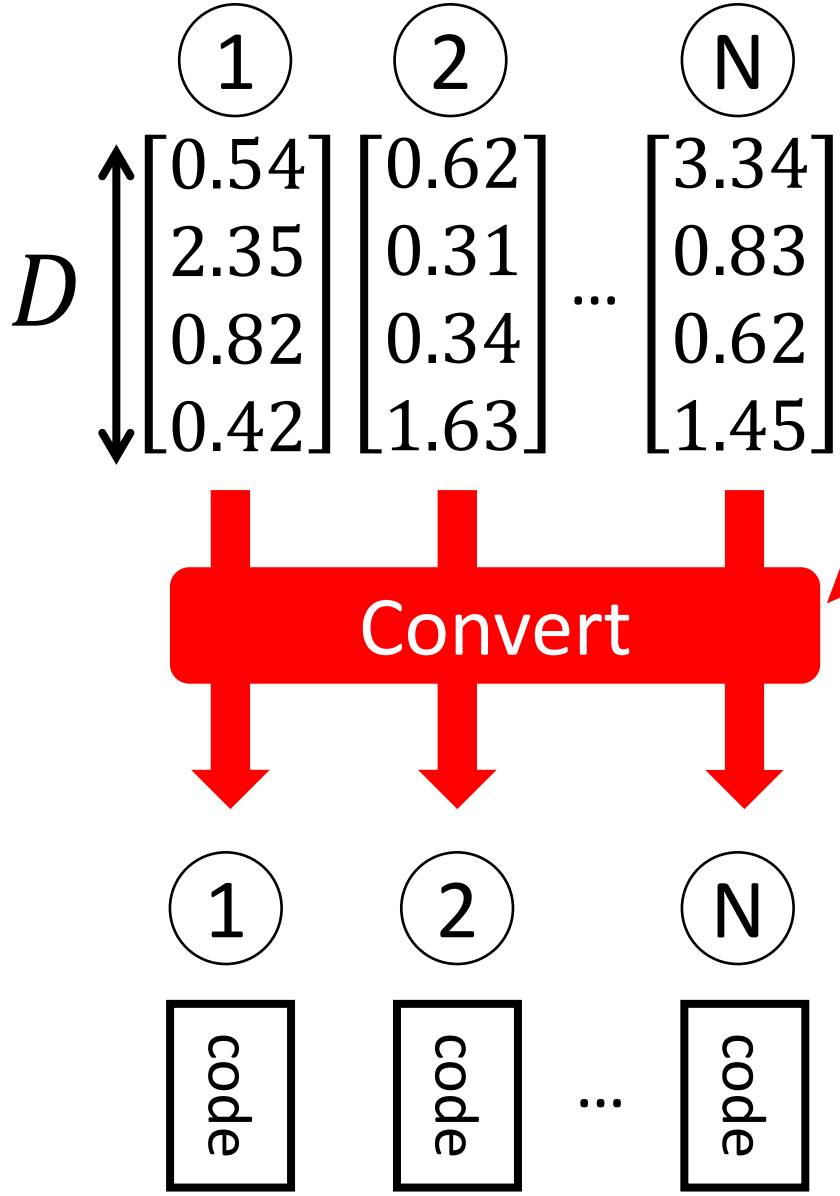
Cluster with 1M centroids, using HNSW to index the centroids

Basic idea



- Need $4ND$ byte to represent N real-valued vectors using floats
- If N or D is too large, we cannot read the data on memory
  - ✓ E.g., 512 GB for $D = 128, N = 10^9$
- Convert each vector to a **short-code**
- Short-code is designed as memory-efficient
  - ✓ E.g., 4 GB for the above example, with 32-bit code
- Run search for short-codes

Basic idea

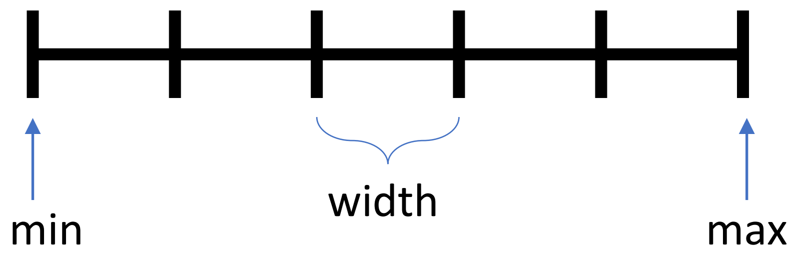


What kind of conversion is preferred?

1. The “distance” between two codes can be calculated
2. The distance can be computed quickly
3. That distance approximates the distance between the original vectors (e.g., L_2)
4. Sufficiently small length of codes can achieve the above three criteria

Quantization Techniques

- **Low precision**
  - work with fp16 instead of 32/64 bit floats
- **Scalar quantization**
  - split up $[\min, \max]$ into K equidistant parts



Interval $[0,3]$ split up into 6 parts

$$\begin{bmatrix} 0.54 \\ 2.35 \\ 0.82 \\ 0.42 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 \\ 5 \\ 1 \\ 0 \end{bmatrix}$$

- **(binary/locality-sensitive) Hashing**
  - Apply hashing to embed into lower dimensional space
- **Product quantization**

Product Quantization; PQ [Jégou+, TPAMI 2011]

- Split a vector into sub-vectors, and quantize each sub-vector

vector; x

$$D \quad \begin{bmatrix} 0.34 \\ 0.22 \\ 0.68 \\ 1.02 \\ 0.03 \\ 0.71 \end{bmatrix}$$

Codebook

Trained beforehand by k-means on training data

ID: 1 $\begin{bmatrix} 0.13 \\ 0.98 \end{bmatrix}$ ID: 2 $\begin{bmatrix} 0.32 \\ 0.27 \end{bmatrix}$ … ID: 256 $\begin{bmatrix} 1.03 \\ 0.08 \end{bmatrix}$

ID: 1 $\begin{bmatrix} 0.3 \\ 1.28 \end{bmatrix}$ ID: 2 $\begin{bmatrix} 0.35 \\ 0.12 \end{bmatrix}$ … ID: 256 $\begin{bmatrix} 0.99 \\ 1.13 \end{bmatrix}$

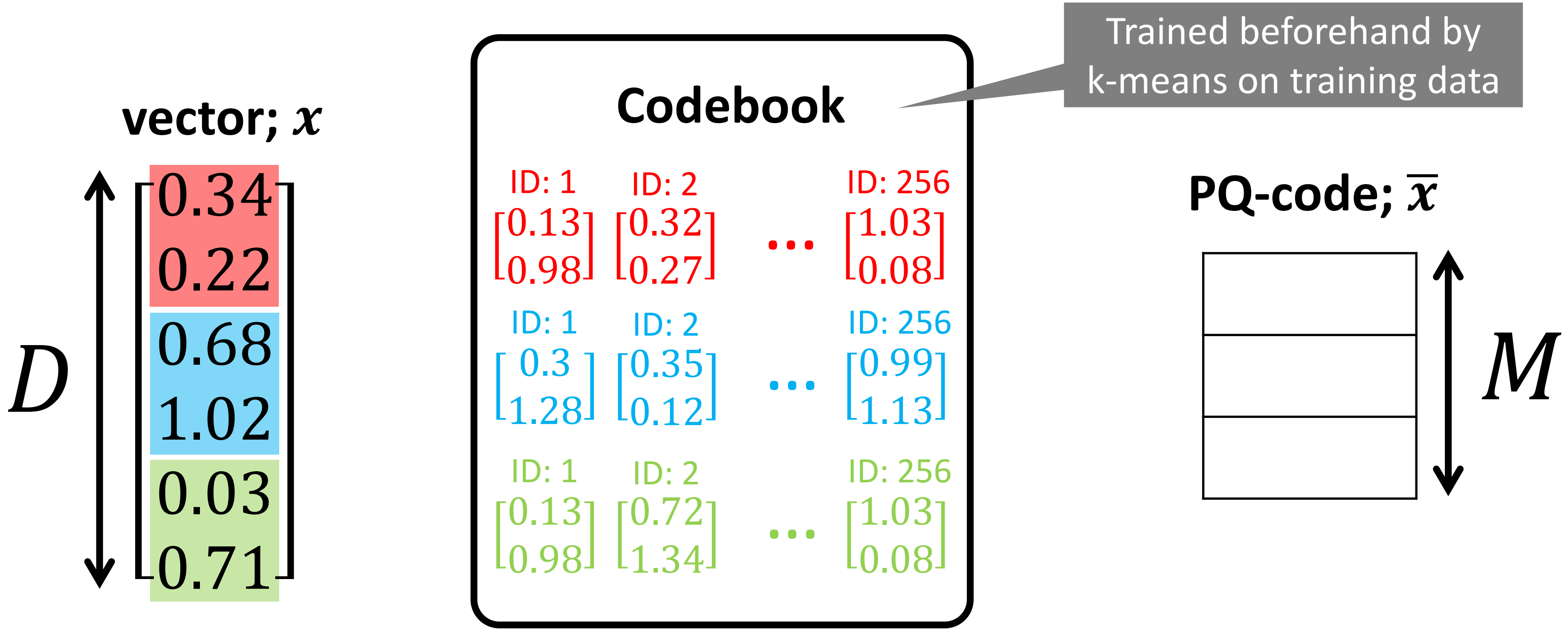
ID: 1 $\begin{bmatrix} 0.13 \\ 0.98 \end{bmatrix}$ ID: 2 $\begin{bmatrix} 0.72 \\ 1.34 \end{bmatrix}$ … ID: 256 $\begin{bmatrix} 1.03 \\ 0.08 \end{bmatrix}$

PQ-code; $\bar{x}$

M

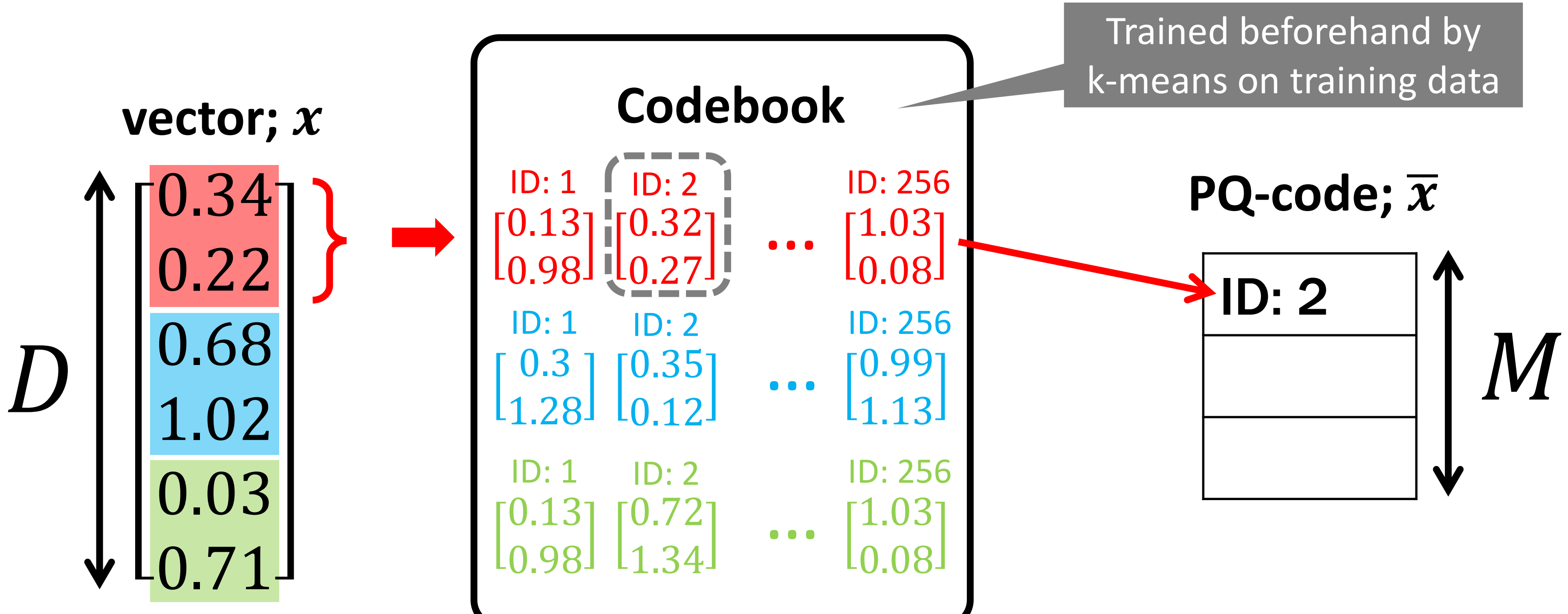
Product Quantization; PQ [Jégou+, TPAMI 2011]

➢ Split a vector into sub-vectors, and quantize each sub-vector



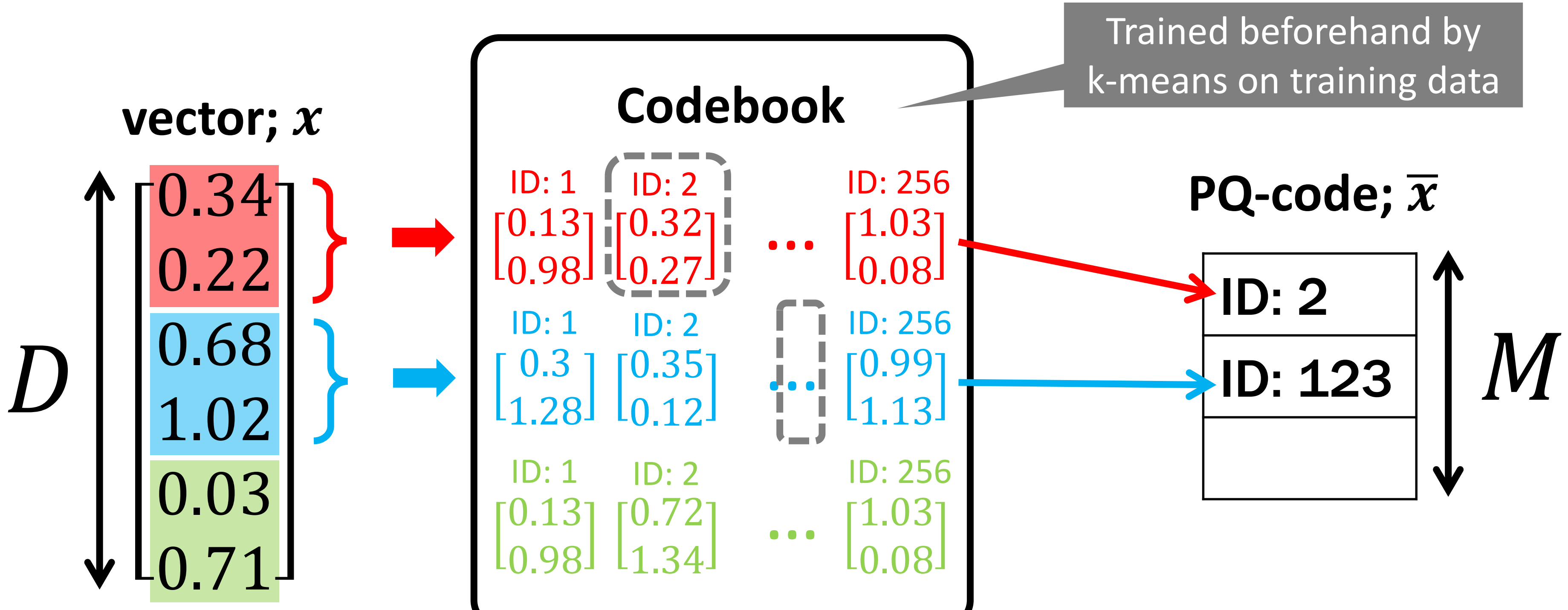
Product Quantization; PQ [Jégou+, TPAMI 2011]

➢ Split a vector into sub-vectors, and quantize each sub-vector



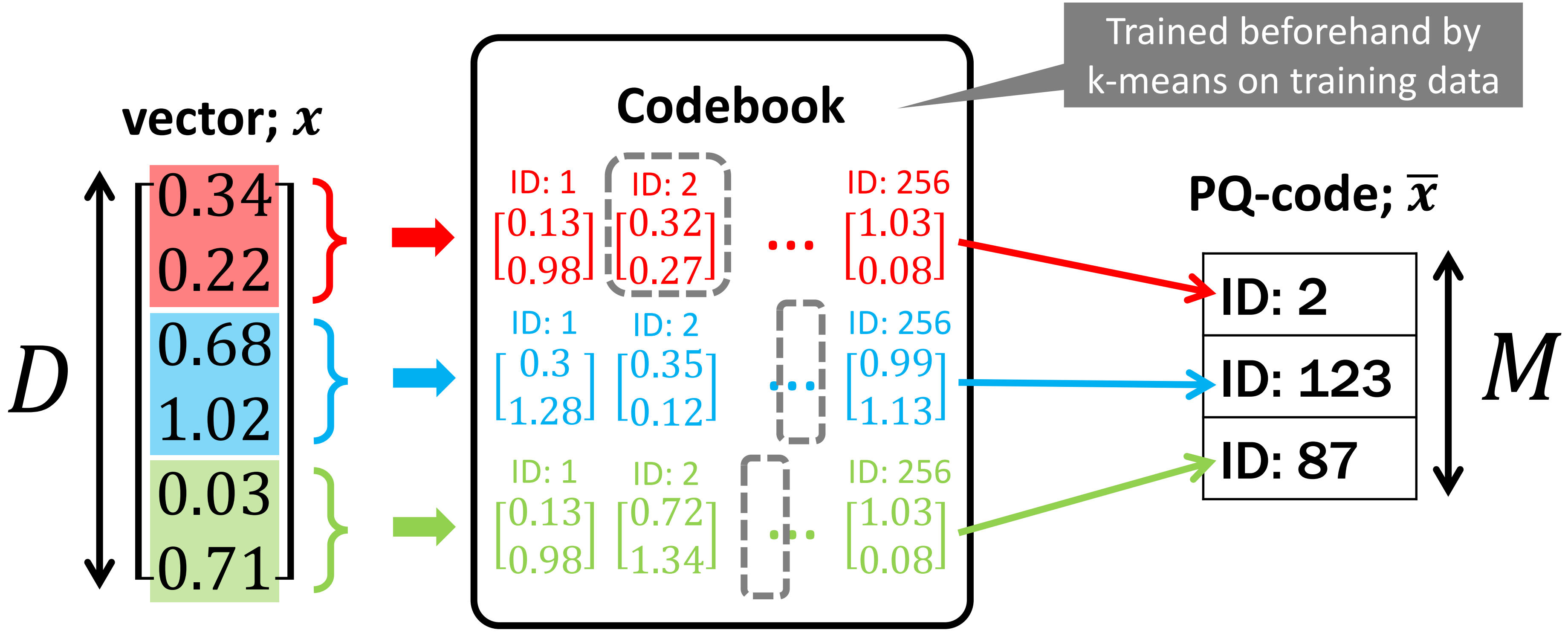
Product Quantization; PQ [Jégou, TPAMI 2011]

- Split a vector into sub-vectors, and quantize each sub-vector



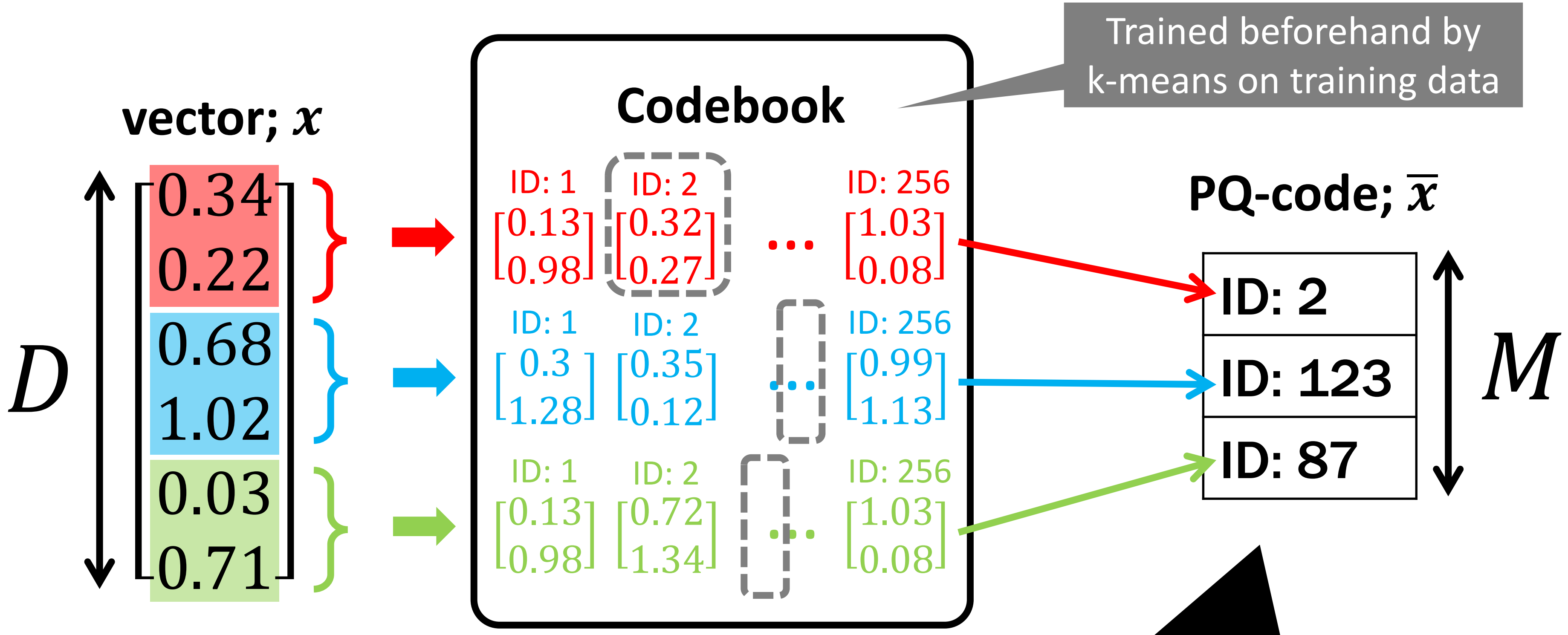
Product Quantization; PQ [Jégou, TPAMI 2011]

- Split a vector into sub-vectors, and quantize each sub-vector



Product Quantization; PQ [Jégou, TPAMI 2011]

➢ Split a vector into sub-vectors, and quantize each sub-vector



➢ Simple

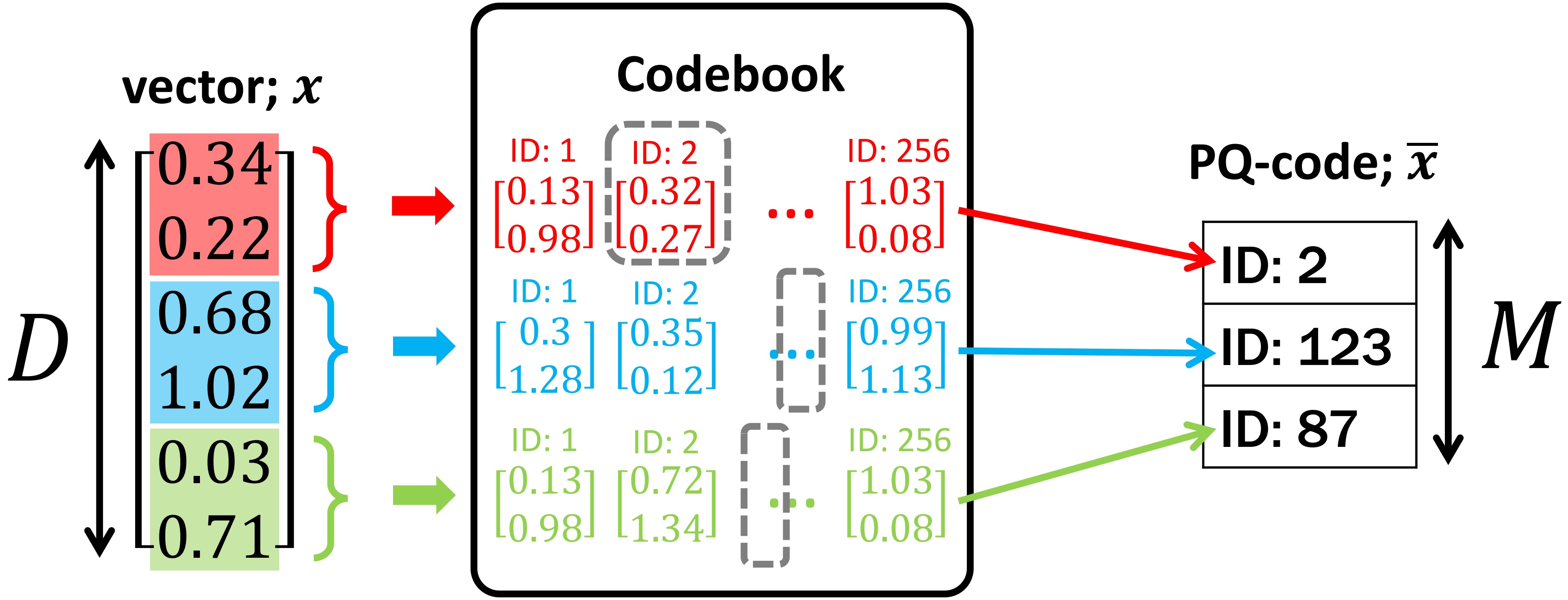
➢ Memory efficient

➢ Distance can be estimated

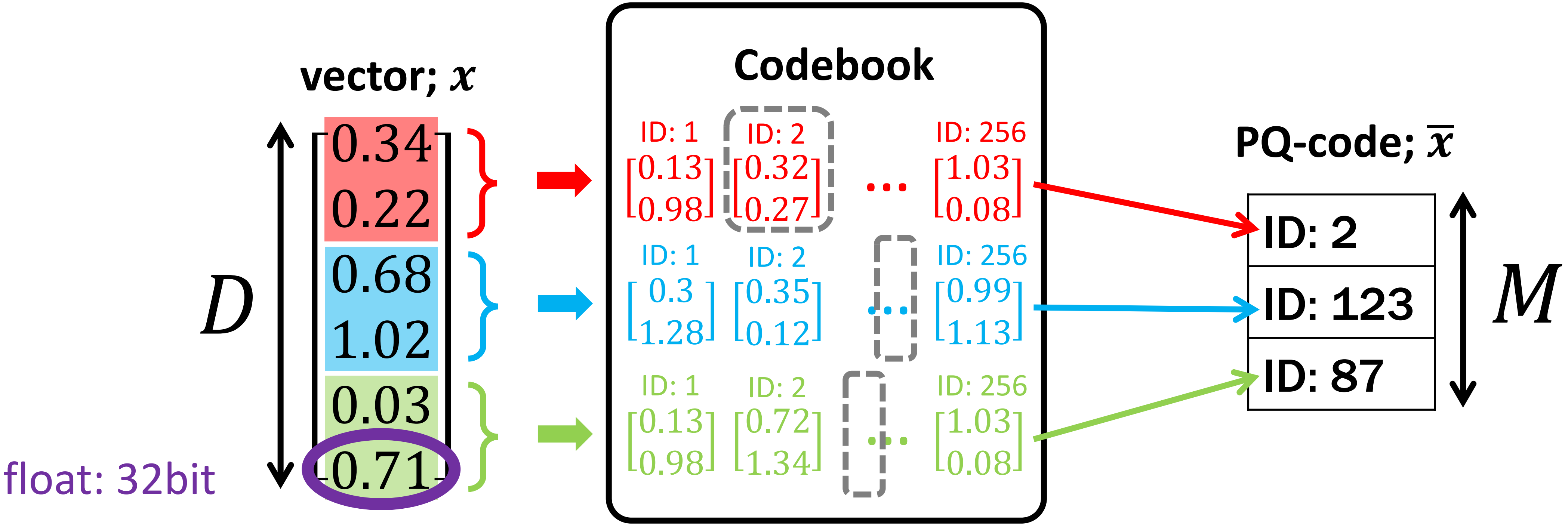
Bar notation for PQ-code:

$$\boldsymbol{x} \in \mathbb{R}^D \quad \mapsto \quad \bar{\boldsymbol{x}} \in \{1, \ldots, 256\}^M$$

Product Quantization: Memory efficient



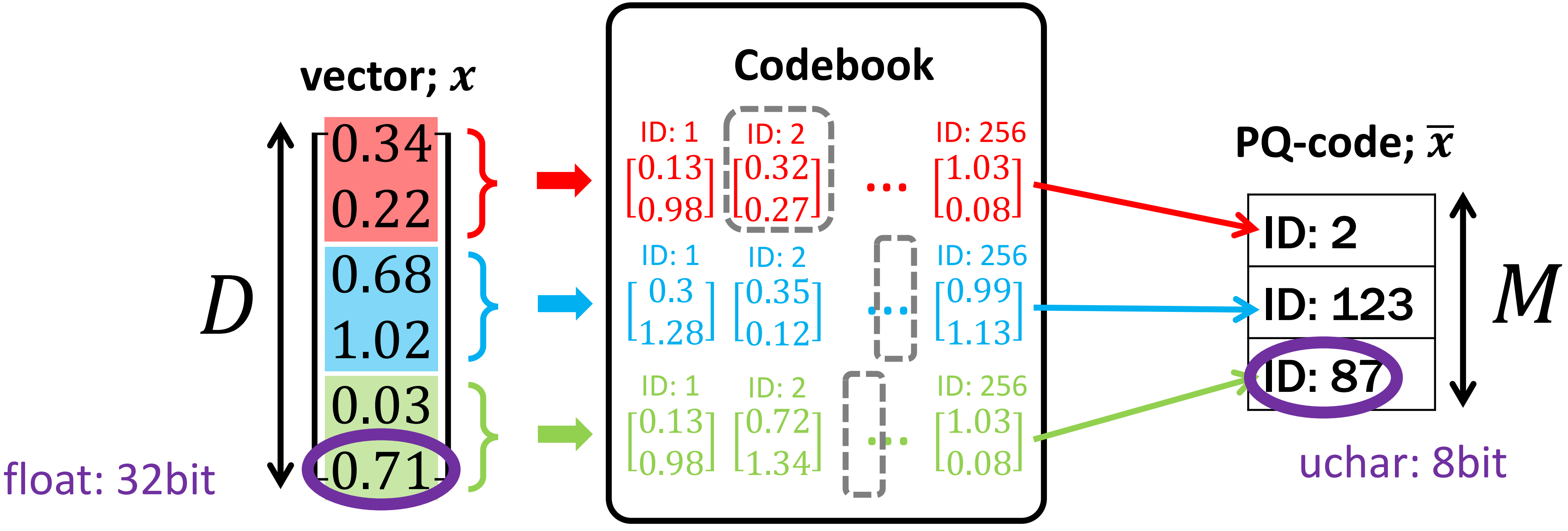
Product Quantization: Memory efficient



e.g., $D = 128$

$128 \times 32 = 4096$ [bit]

Product Quantization: Memory efficient



e.g., $D = 128$

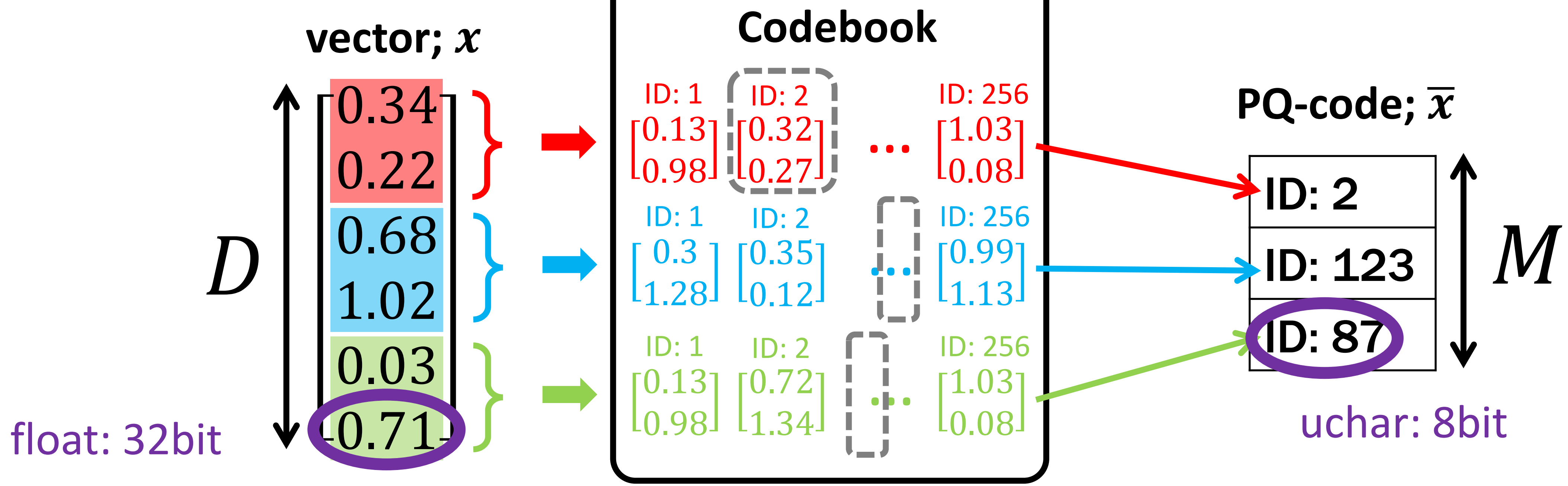
$128 \times 32 = 4096$ [bit]

e.g., $M = 8$

$8 \times 8 = 64$ [bit]

Product Quantization: Memory efficient

Can store all 1B vectors with 8 GB of RAM!



e.g., $D = 128$

$128 \times 32 = 4096$ [bit]

e.g., $M = 8$

$8 \times 8 = 64$ [bit]

1/64

Product Quantization: Distance estimation

Query; $\boldsymbol{q} \in \mathbb{R}^D$

$$\begin{bmatrix} 0.34 \\ 0.22 \\ 0.68 \\ 1.02 \\ 0.03 \\ 0.71 \end{bmatrix}$$

Database vectors

$$\begin{array}{cccc} \boldsymbol{x}_1 & \boldsymbol{x}_2 & & \boldsymbol{x}_N \\ \begin{bmatrix} 0.54 \\ 2.35 \\ 0.82 \\ 0.42 \\ 0.14 \\ 0.32 \end{bmatrix} & \begin{bmatrix} 0.62 \\ 0.31 \\ 0.34 \\ 1.63 \\ 1.43 \\ 0.74 \end{bmatrix} & \cdots & \begin{bmatrix} 3.34 \\ 0.83 \\ 0.62 \\ 1.45 \\ 0.12 \\ 2.32 \end{bmatrix} \end{array}$$

Product Quantization: Distance estimation

Query; $\boldsymbol{q} \in \mathbb{R}^D$

$$\begin{bmatrix} 0.34 \\ 0.22 \\ 0.68 \\ 1.02 \\ 0.03 \\ 0.71 \end{bmatrix}$$

Database vectors

$$\begin{array}{cccc} \boldsymbol{x}_1 & \boldsymbol{x}_2 & & \boldsymbol{x}_N \\ \begin{bmatrix} 0.54 \\ 2.35 \\ 0.82 \\ 0.42 \\ 0.14 \\ 0.32 \end{bmatrix} & \begin{bmatrix} 0.62 \\ 0.31 \\ 0.34 \\ 1.63 \\ 1.43 \\ 0.74 \end{bmatrix} & \cdots & \begin{bmatrix} 3.34 \\ 0.83 \\ 0.62 \\ 1.45 \\ 0.12 \\ 2.32 \end{bmatrix} \end{array}$$

Product quantization

Product Quantization: Distance estimation

Query; $\boldsymbol{q} \in \mathbb{R}^D$

$$\begin{bmatrix} 0.34 \\ 0.22 \\ 0.68 \\ 1.02 \\ 0.03 \\ 0.71 \end{bmatrix}$$

$\overline{\boldsymbol{x}_1} \in \{1, \dots, 256\}^M$

$\overline{\boldsymbol{x}_1}$	$\overline{\boldsymbol{x}_2}$	…	$\overline{\boldsymbol{x}_N}$
ID: 42	ID: 221		ID: 99
ID: 67	ID: 143	…	ID: 234
ID: 92	ID: 34		ID: 3

Product Quantization: Distance estimation

Query; $\boldsymbol{q} \in \mathbb{R}^D$

$$\begin{bmatrix} 0.34 \\ 0.22 \\ 0.68 \\ 1.02 \\ 0.03 \\ 0.71 \end{bmatrix}$$

Linear Scan Through Candidates

$\overline{\boldsymbol{x}_1} \in \{1, \dots, 256\}^M$

$\overline{\boldsymbol{x}_1}$	$\overline{\boldsymbol{x}_2}$	…	$\overline{\boldsymbol{x}_N}$
ID: 42	ID: 221		ID: 99
ID: 67	ID: 143		ID: 234
ID: 92	ID: 34		ID: 3

Asymmetric distance

- $d(\boldsymbol{q}, \boldsymbol{x})^2$ can be efficiently approximated by $d_A(\boldsymbol{q}, \overline{\boldsymbol{x}})^2$
- Lookup-trick: Looking up pre-computed distance-tables
- Candidate selection by d_A

Not pseudo codes

```python
import numpy as np
from scipy.cluster.vq import vq, kmeans2
from scipy.spatial.distance import cdist

def train(vec, M):
    Ds = int(vec.shape[1] / M) # Ds = D / M
    # codeword[m][k] = c_k^m
    codeword = np.empty((M, 256, Ds), np.float32)

    for m in range(M):
        vec_sub = vec[:, m * Ds : (m + 1) * Ds]
        codeword[m], label = kmeans2(vec_sub, 256)

    return codeword

def encode(codeword, vec): # vec = {x_n}_{n=1}^N
    M, _K, Ds = codeword.shape
    # pqcode[n] = i(x_n),  pqcode[n][m] = i^m(x_n)
    pqcode = np.empty((vec.shape[0], M), np.uint8)

    for m in range(M): # Eq. (2) and Eq. (3)
        vec_sub = vec[:, m * Ds: (m + 1) * Ds]
        pqcode[:, m], dist = vq(vec_sub, codeword[m])

    return pqcode

def search(codeword, pqcode, query):
    M, _K, Ds = codeword.shape
    # dist_table = D(m, k)
    dist_table = np.empty((M, 256), np.float32)

    for m in range(M):
        query_sub = query[m * Ds: (m + 1) * Ds]
        dist_table[m, :] = cdist([query_sub],
     ↪  codeword[m], 'sqeuclidean')[0] # Eq. (5)

    # Eq. (6)
    dist = np.sum(dist_table[range(M), pqcode], axis=1)

    return dist

if __name__ == "__main__":
    # Read vec_train, vec ({x_n}_{n=1}^N), and query (y)
    M = 4
    codeword = train(vec_train, M)
    pqcode = encode(codeword, vec)
    dist = search(codeword, pqcode, query)
    print(dist)
```

- Only tens of lines in Python
- Pure Python library: nanopq https://github.com/matsui528/nanopq
- `pip install nanopq`

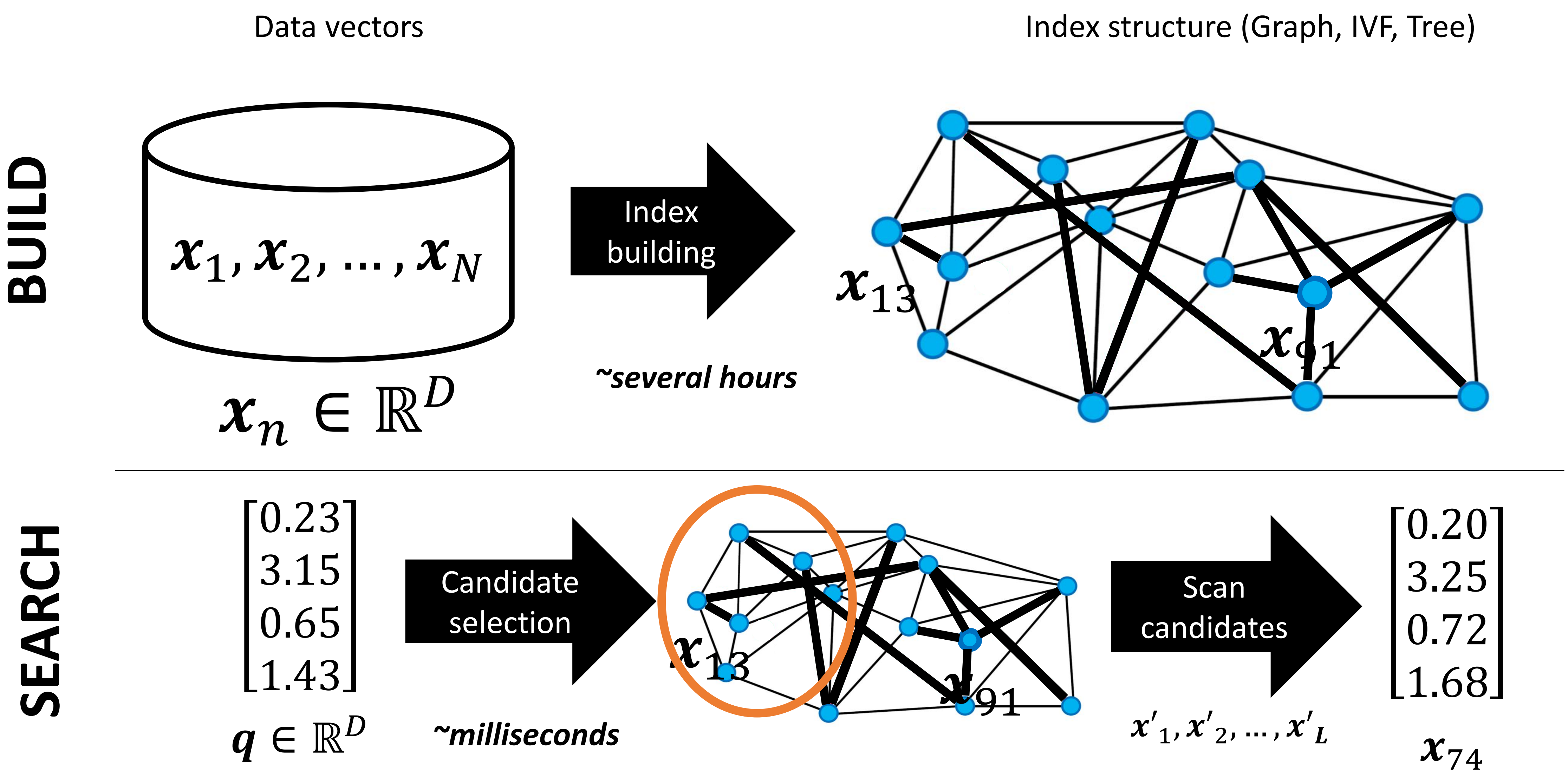
Rotate vectors to allow for better product quantization [Ge+14]

	BIGANN	MSSPACEV	TEXT2IMAGE	SSNPP
DiskANN	$R = 64, L = 128, \alpha = 1.2$	$R = 64, L = 128, \alpha = 1.2$	$R = 64, L = 128, \alpha = .9$	$R = 150, L = 400, \alpha = 1.2$
HNSW	$m = 32, efc = 128, \alpha = .82$	$m = 32, efc = 128, \alpha = .83$	$m = 32, efc = 128, \alpha = 1.1$	$m = 75, efc = 400, \alpha = .82$
HCNNG	$T = 30, Ls = 1000, s = 3$	$T = 50, Ls = 1000, s = 3$	$T = 30, Ls = 1000, s = 3$	$T = 50, Ls = 1000, s = 3$
pyNNDescent	$K = 40, Ls = 100,$ $T = 10, \alpha = 1.2$	$K = 60, Ls = 100,$ $T = 10, \alpha = 1.2$	$K = 60, Ls = 100,$ $T = 10, \alpha = .9$	$K = 60, Ls = 1000,$ $T = 10, \alpha = 1.4$
FAISS	OPQ64_128, IVF1048576_HNSW32, PQ128x4fsr	OPQ64_128, IVF1048576_HNSW32, PQ64x4fsr	OPQ64_128, IVF1048576_HNSW32, PQ128x4fsr	OPQ64_128, IVF1048576_HNSW32, PQ64

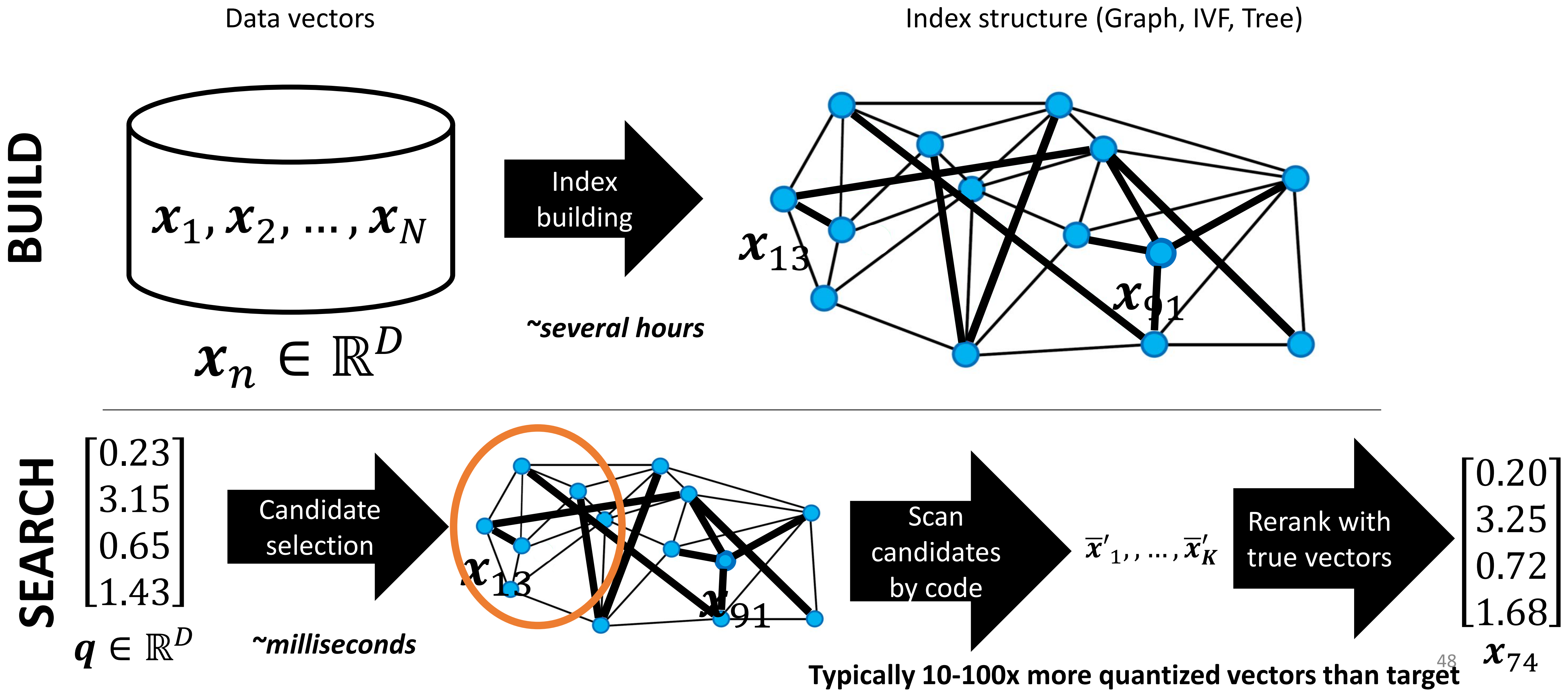
- Compress vector into 128 blocks,
- each with 2^4 = 16 codewords,
- use SIMD-based asymmetric distance computation [Andre+17]

Cluster with 1M centroids, using HNSW to index the centroids

The ANN search pipeline



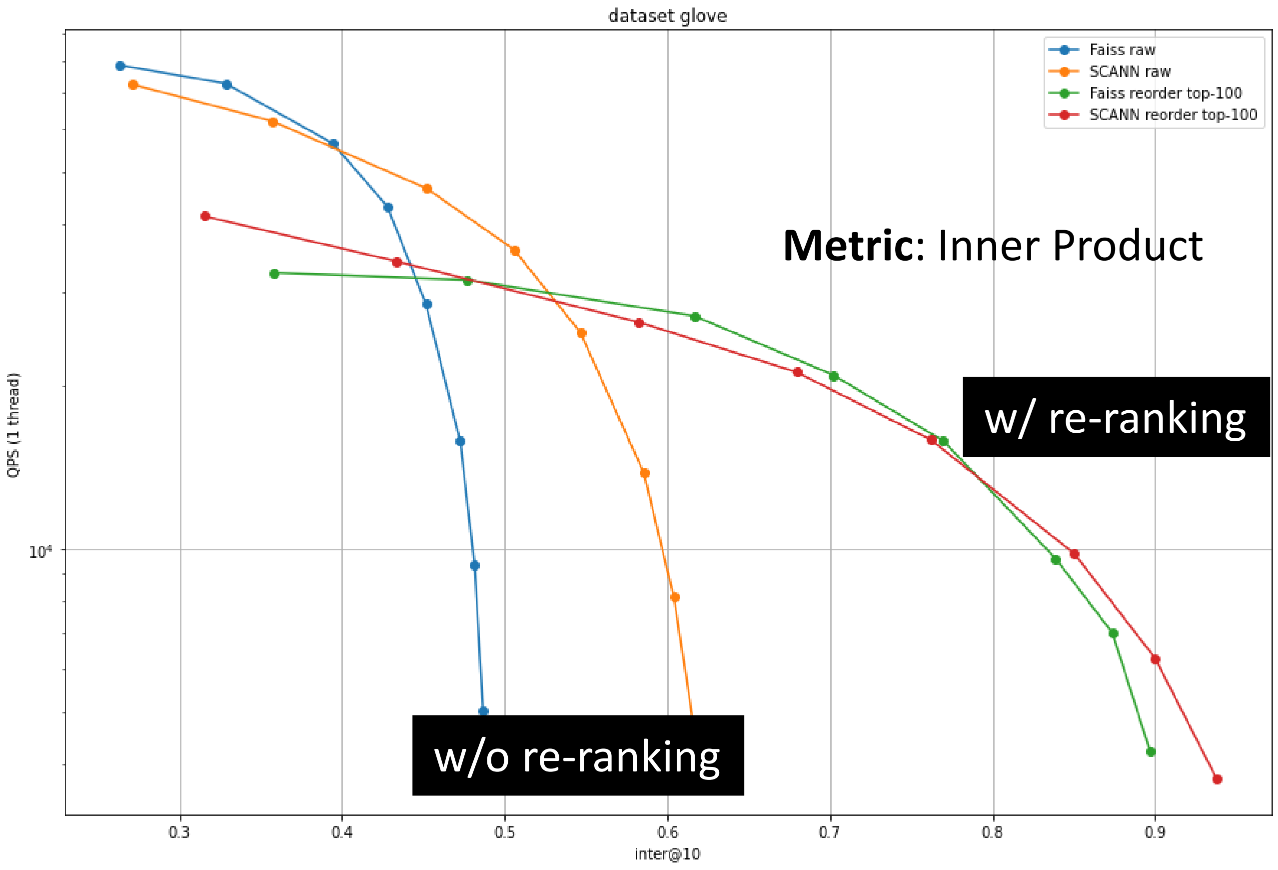
The ANN search pipeline (with quantization)



Index on Quantized Vectors

SCANN: Guo+, ICML 2020.

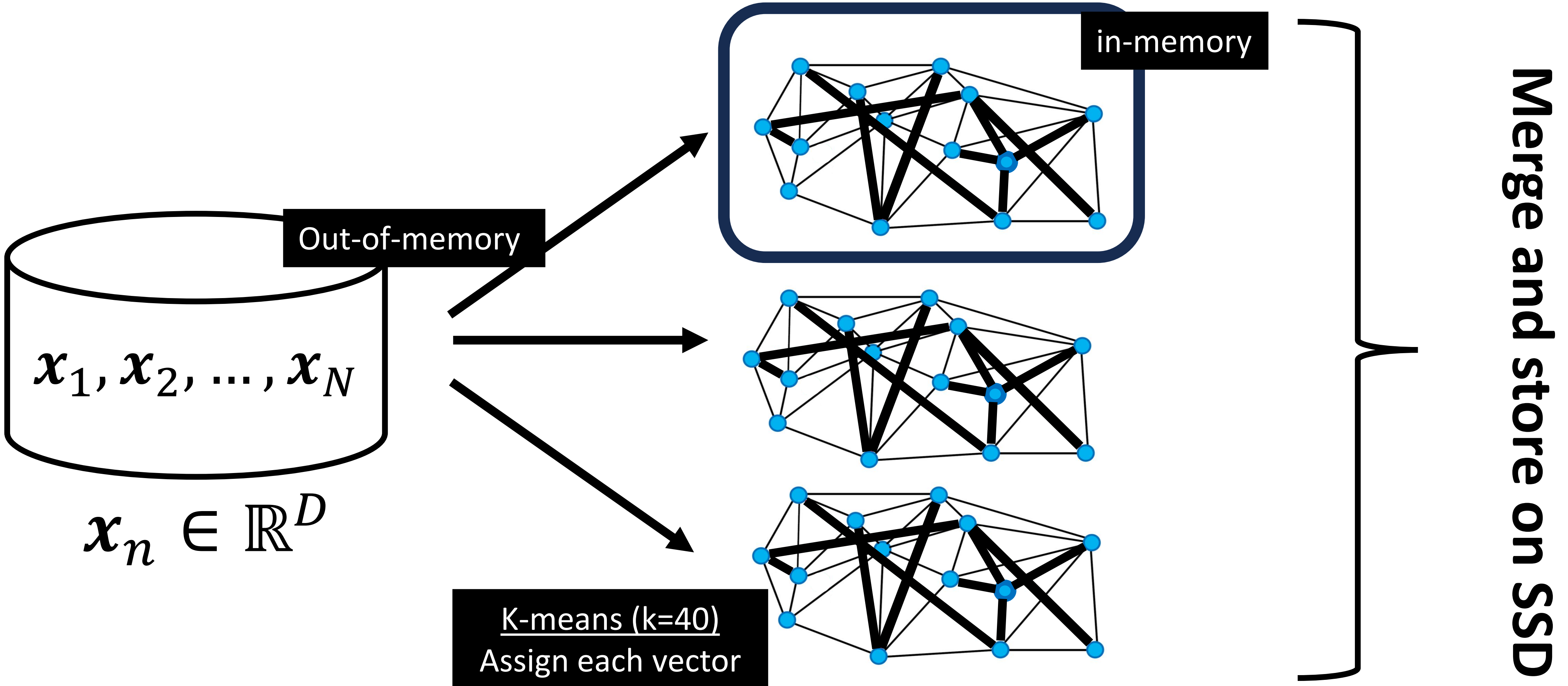
- Learn codes, represent each vector by its PQ code
- Code size: 32-64 byte
  - Can store the compressed vectors in memory
  - Lookup tables in cache/avx registers
- Index cost on top
  - **Graph**: 1G * degree_bound
    - Typically requires small degree_bounds (not well studied?)
  - **IVF**: 1M centroids + index on centroids on top of vectors
    - Usually works well



Recall quality very data dependent!

https://github.com/facebookresearch/faiss/wiki/Indexing-1M-vectors

Out-of-Memory index + High-Recall (DiskANN)



K-means (k=40)
Assign each vector to its closest 2 centroids

DiskANN out-of-memory

RAM **32+ GB**

$\bar{x}_i$

ID: 42
ID: 67
ID: 92

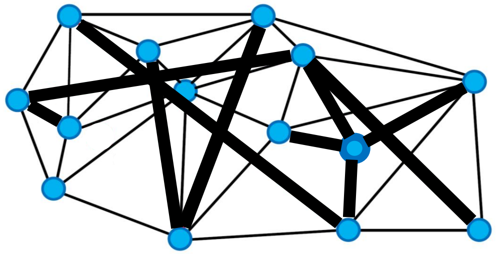
~32byte /vector

Expanding a node:
1. Read adjacent nodes from SSD (+ fetch original vector "for free")
2. Compute distances of query to neighbors (using PQ codes)

Still serves 1k+ queries per second

SSD **dataset size + 512 GB graph**

$R = 128$ degree



Memory layout for vector x_i

x_i	y_1		y_R

Original vector

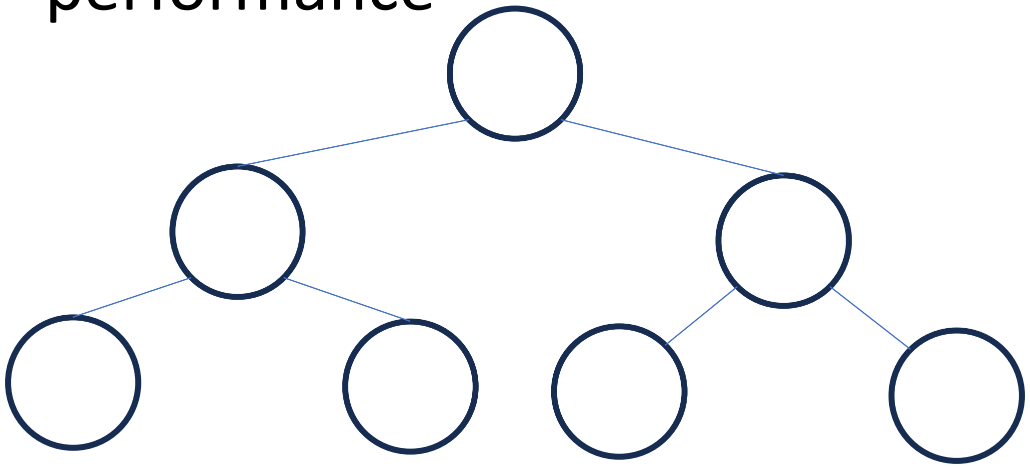
Neighbor indices, Padded if $< R$ neighbors

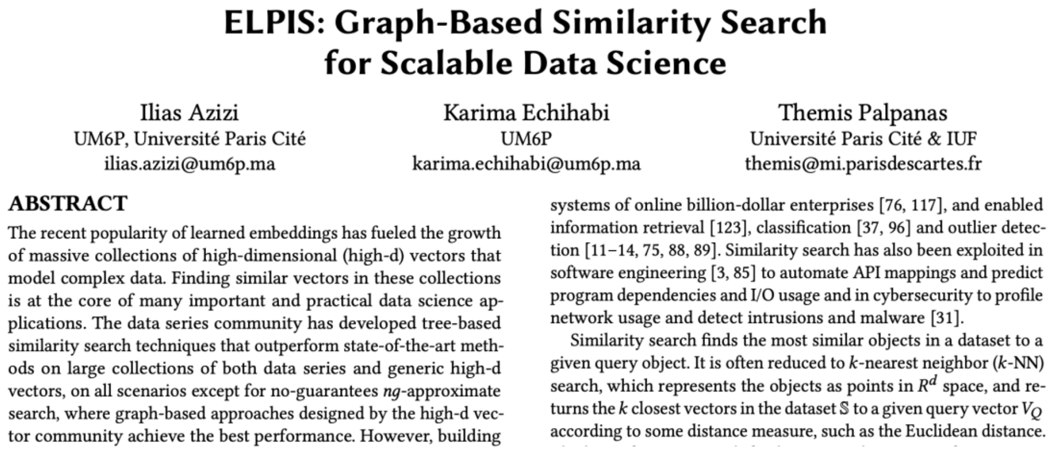
1 block read from SSD

(Very) recent developments

A new graph approach?

- Hierarchical tree, leaves are HNSW graphs
- Interesting quantization technique motivated by time series
- Better build times, good query performance





ELPIS: Graph-Based Similarity Search for Scalable Data Science

Ilias Azizi
UM6P, Université Paris Cité
ilias.azizi@um6p.ma

Karima Echihabi
UM6P
karima.echihabi@um6p.ma

Themis Palpanas
Université Paris Cité & IUF
themis@mi.parisdescartes.fr

ABSTRACT

The recent popularity of learned embeddings has fueled the growth of massive collections of high-dimensional (high-d) vectors that model complex data. Finding similar vectors in these collections is at the core of many important and practical data science applications. The data series community has developed tree-based similarity search techniques that outperform state-of-the-art methods on large collections of both data series and generic high-d vectors, on all scenarios except for no-guarantees *ng*-approximate search, where graph-based approaches designed by the high-d vector community achieve the best performance. However, building

systems of online billion-dollar enterprises [76, 117], and enabled information retrieval [123], classification [37, 96] and outlier detection [11–14, 75, 88, 89]. Similarity search has also been exploited in software engineering [3, 85] to automate API mappings and predict program dependencies and I/O usage and in cybersecurity to profile network usage and detect intrusions and malware [31].

Similarity search finds the most similar objects in a dataset to a given query object. It is often reduced to k-nearest neighbor (k-NN) search, which represents the objects as points in R^d space, and returns the k closest vectors in the dataset $\mathbb{S}$ to a given query vector V_Q according to some distance measure, such as the Euclidean distance.

To appear at VLDB 2023,
https://www.vldb.org/pvldb/vol16/p1548-azizi.pdf

Automated Parameter tuning

- Finding build/search parameters by constrained optimization
- Build on top of ScaNN

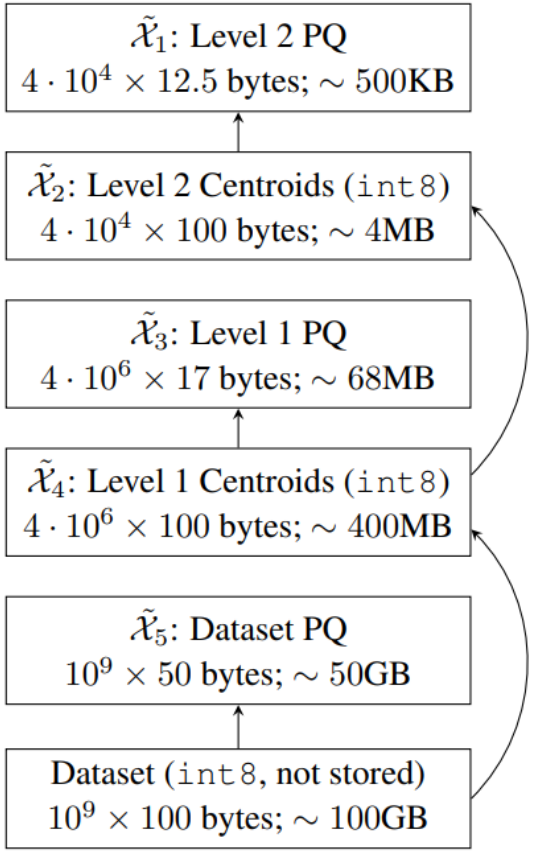
AUTOMATING NEAREST NEIGHBOR SEARCH CONFIGURATION WITH CONSTRAINED OPTIMIZATION

Philip Sun, Ruiqi Guo & Sanjiv Kumar
Google Research
New York, NY
{sunphil,guorq,sanjivk}@google.com

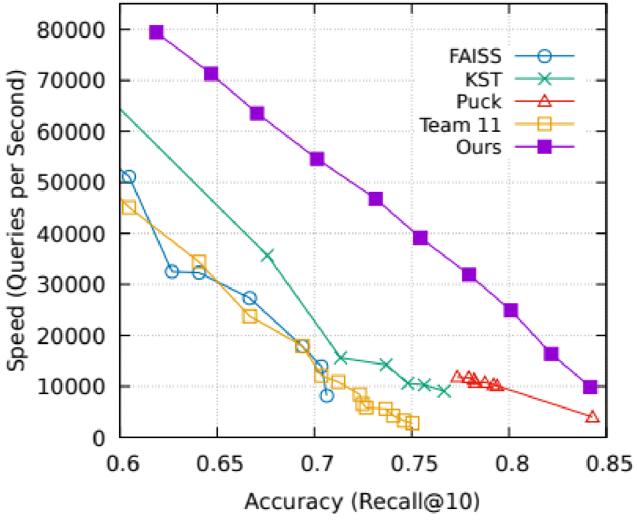
ICLR23

ABSTRACT

The approximate nearest neighbor (ANN) search problem is fundamental to efficiently serving many real-world machine learning applications. A number of techniques have been developed for ANN search that are efficient, accurate, and scalable. However, such techniques typically have a number of parameters that affect the speed-recall tradeoff, and exhibit poor performance when such parameters aren't properly set. Tuning these parameters has traditionally been a manual process, demanding in-depth knowledge of the underlying search algorithm. This is becoming an increasingly unrealistic demand as ANN search grows in popu-



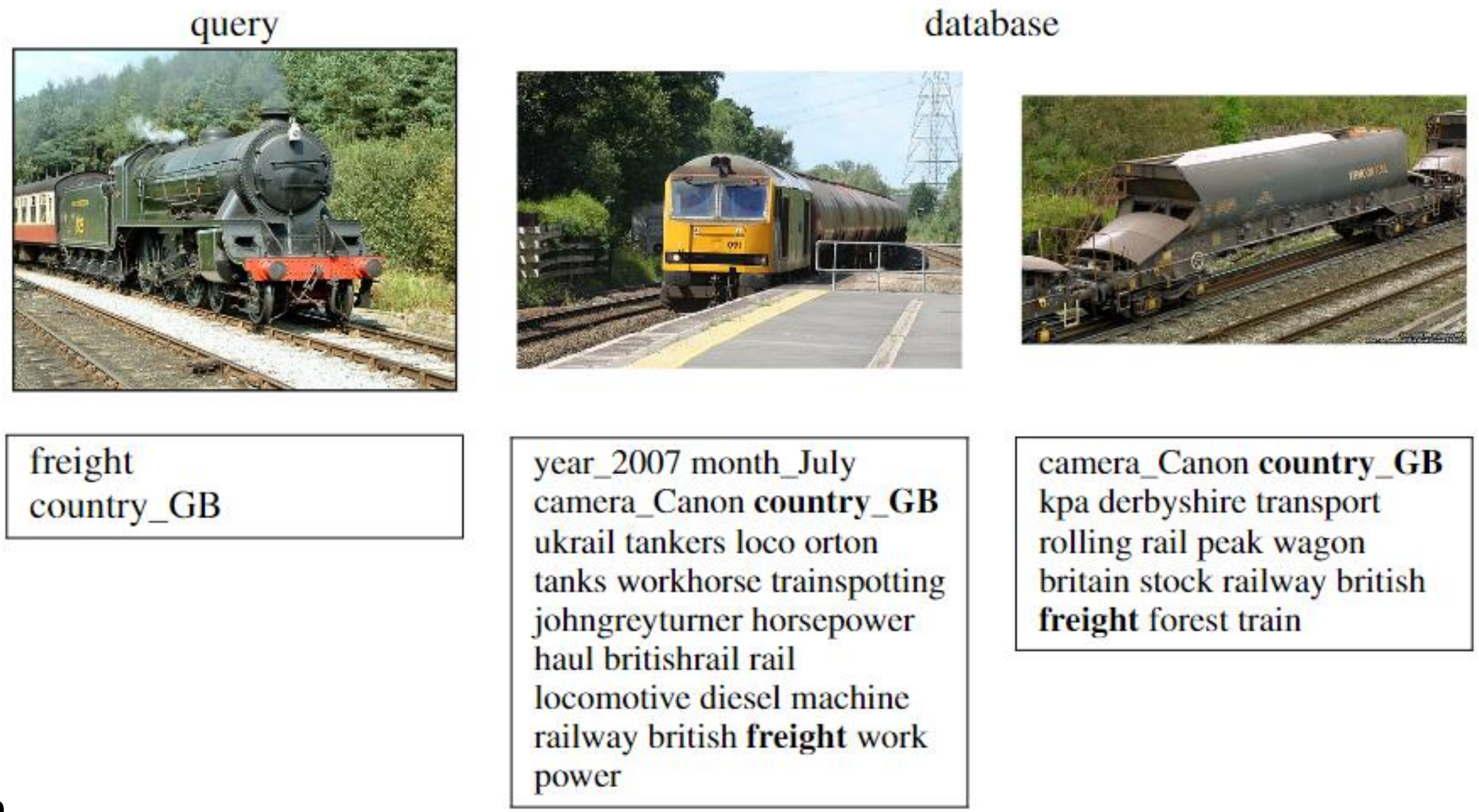
(b) Microsoft Turing-ANNS



(b) Microsoft Turing-ANNS

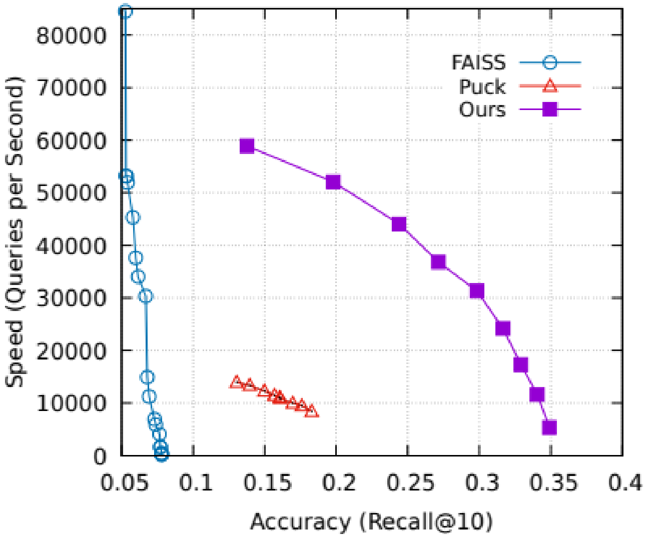
Filtered search

- **Setting**
  - Vectors have associated metadata
  - Example, YFCC: tags, gps, date
- **Query**
  - Find the most similar images to this images that were taken with a Sony Camera in 2017 in Vancouver



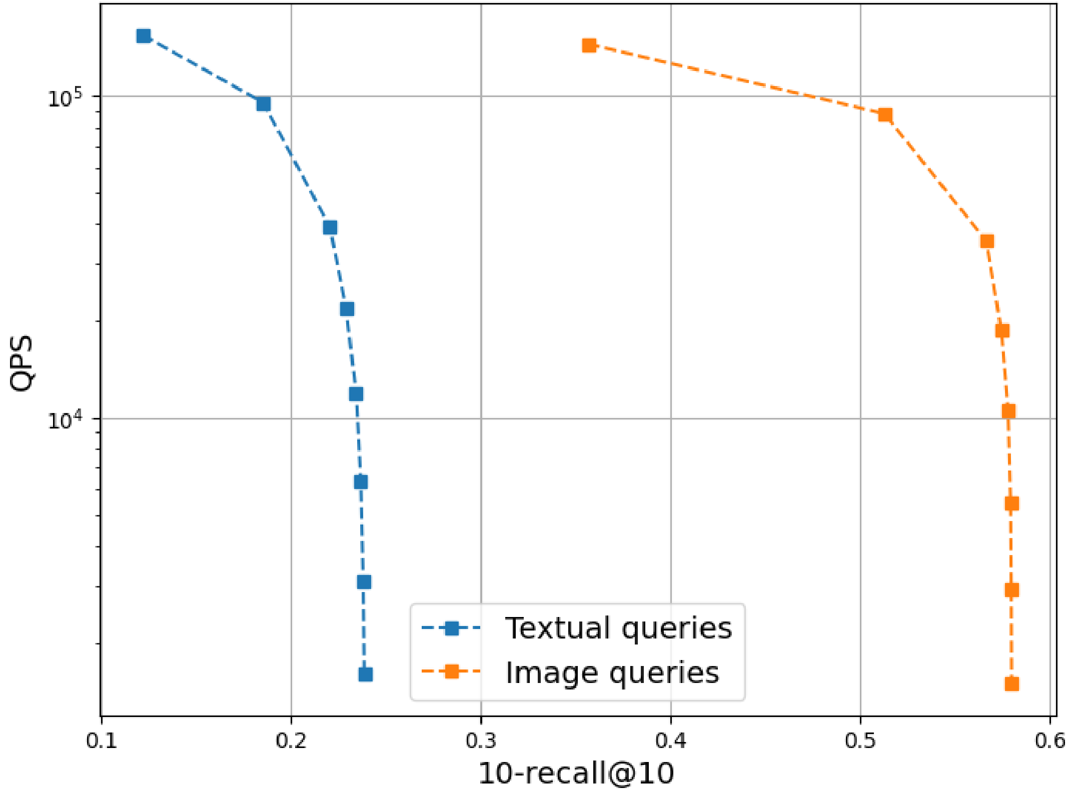
Out-of-distribution queries

- **Setting**
  - Vectors are image embeddings
  - Queries are text embeddings



(c) Yandex Text-to-Image

https://arxiv.org/pdf/2301.01702.pdf



OPQ64_128,IVF16384,PQ64

Yandex, Text-2-Image dataset

Streaming settings

- **Setting**
  - Many applications (search engine, recommender system) need to handle updates
- Daily rebuilds often too expensive
- **Question**: Clever update strategies?

	Web Search & Reco	Email Search	Enterprise search
Index Size	~1 trillion pages	100s of trillions of sentences	Trillions of paragraphs across documents
Update Rate (latency <1s)	Billions of updates/day	Ingest new email, Purge deletes	Handle >1% change/day
Search latency/QPS	<10ms 10-100K+ Queries/sec	100s of ms	10-100ms

https://harsha-simhadri.org/pubs/ANNS-talk-Sep22.pptx

NeurIPS 2023 Challenge: Practical Vector Search

- **4 Tasks (10M vectors)**
  - Filtered ANN
  - Streaming ANN
  - Out-of-distribution ANN
  - ANN on sparse data
- Strong baselines based on IVF (faiss) and graphs (DiskANN)
- Cloud credits available for testing (screening process)

Practical Vector Search Challenge 2023

Harsha Vardhan Simhadri*
Microsoft Research India
harshasi@microsoft.com

Martin Aumüller
IT University of Copenhagen
maau@itu.dk

Dmitry Baranchuk
Yandex
dbaranchuk@yandex-team.ru

Matthijs Douze
Meta AI Research
matthijs@meta.com

Edo Liberty
Pinecone.io
edo@pinecone.io

Amir Ingber
Pinecone.io
ingber@pinecone.io

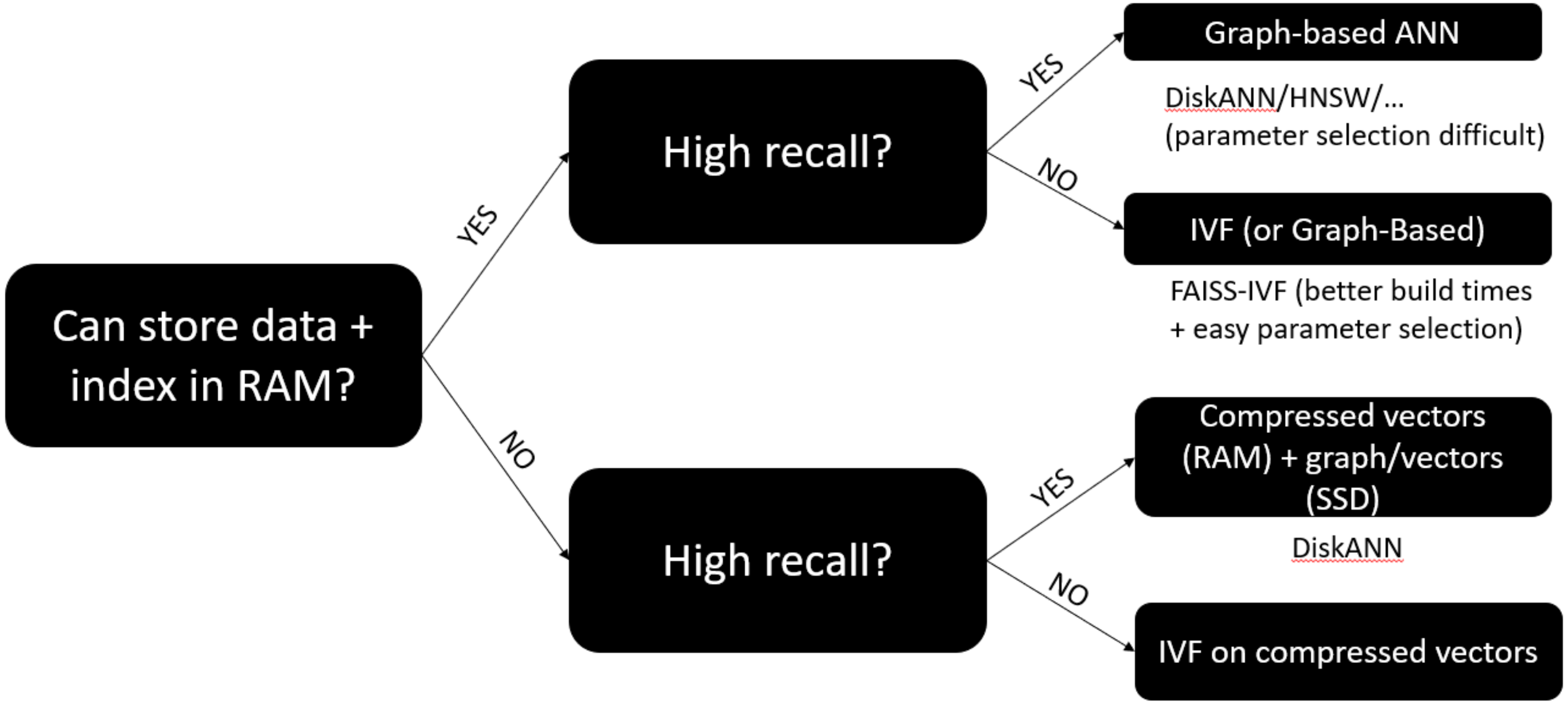
Frank Liu
Zilliz
frank.liu@zilliz.com

George Williams
Independent Researcher
gwilliams@ieee.org

Official announcement soon!

https://big-ann-benchmarks.com

Timeframe: July-November 2023



Thanks!

https://matsui528.github.io/cvpr2023_tutorial_neural_search/

https://big-ann-benchmarks.com