billion-scaleの近似最近傍探索

2019/1/22

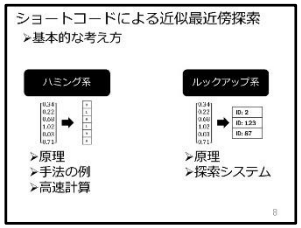
松井勇佑
東京大学　生産技術研究所

関連する資料

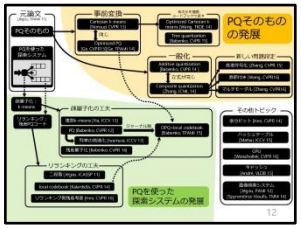
直積量子化を用いた近似最近傍探索に関するサーベイ
http://yusukematsui.me/project/survey_pq/survey_pq_jp.html

サーベイ論文：
Y. Matsui, Y. Uchida, H. Jégou, and S. Satoh
“A Survey of Product Quantization”, ITE Journal, 2018, [link]

関連するスライド・本：



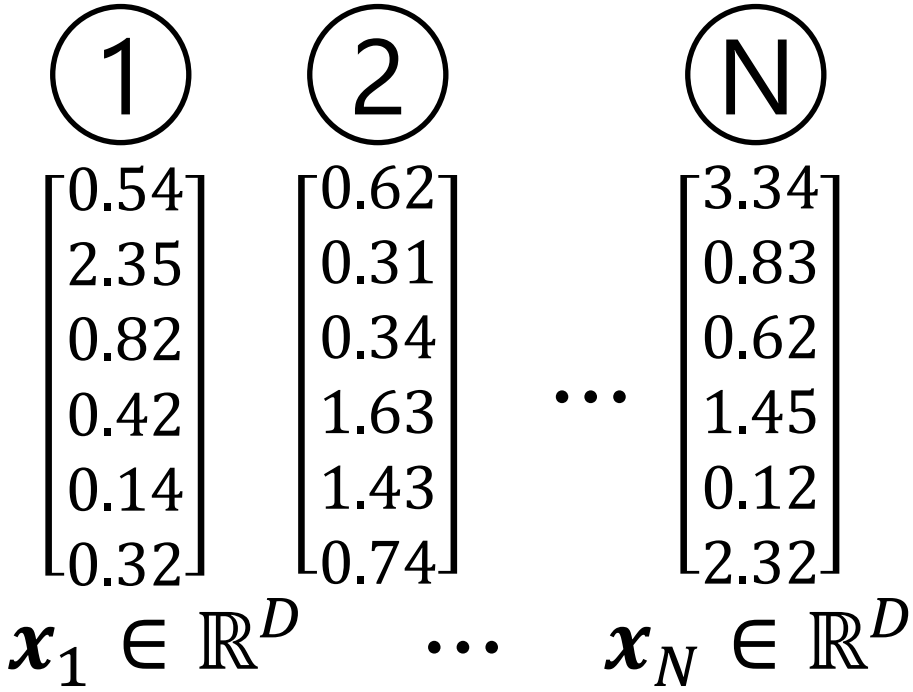
ショートコードによる
大規模近似最近傍探索
講義資料，大阪大学，2016
[link]



【招待ショートサーベイ】
直積量子化を用いた
近似最近傍探索
PRMU, 2016 [link]

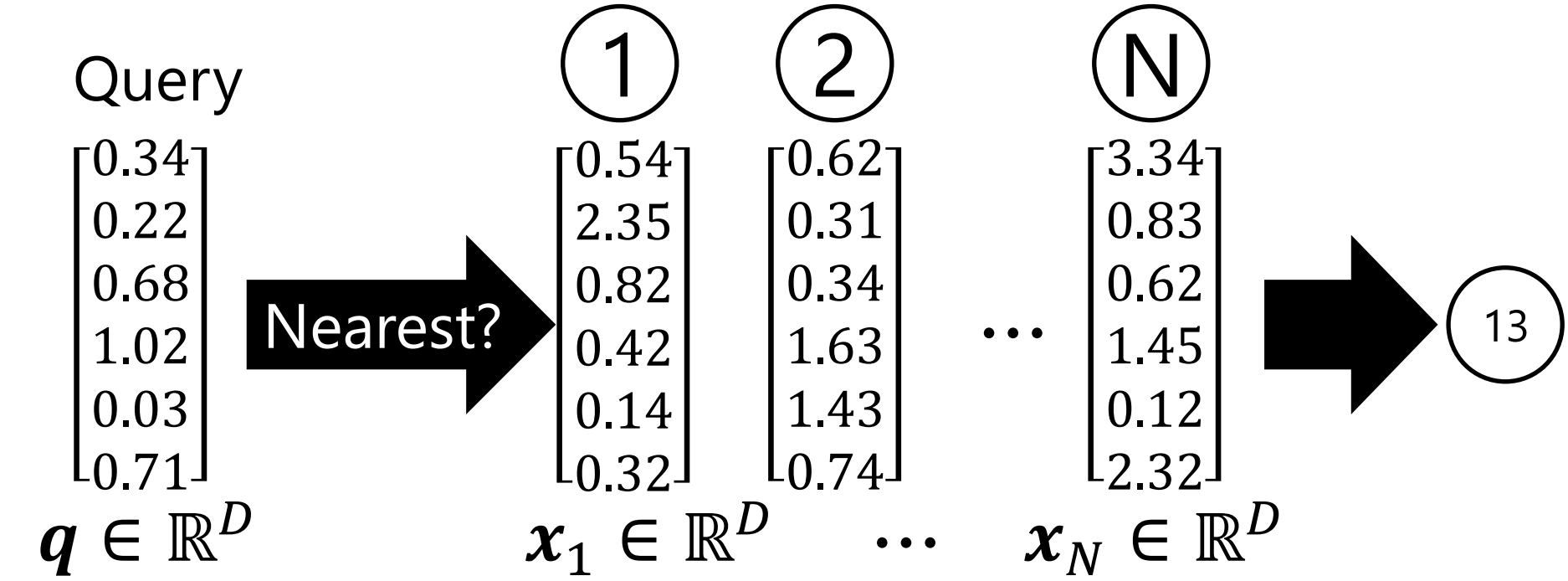
コンピュータビジョン—広がる要素技術と応用—
第6章：近似最近傍探索
共立出版，2018 [Amazon] [共立出版]
米谷 竜・斎藤 英雄・池畑 諭・牛久 祥孝・内山 英昭・内海 ゆづ子・小野 峻佑・
片岡 裕雄・金崎 朝子・川西 康友・齋藤 真樹・櫻田 健・高橋 康輔・松井 勇佑

最近傍探索



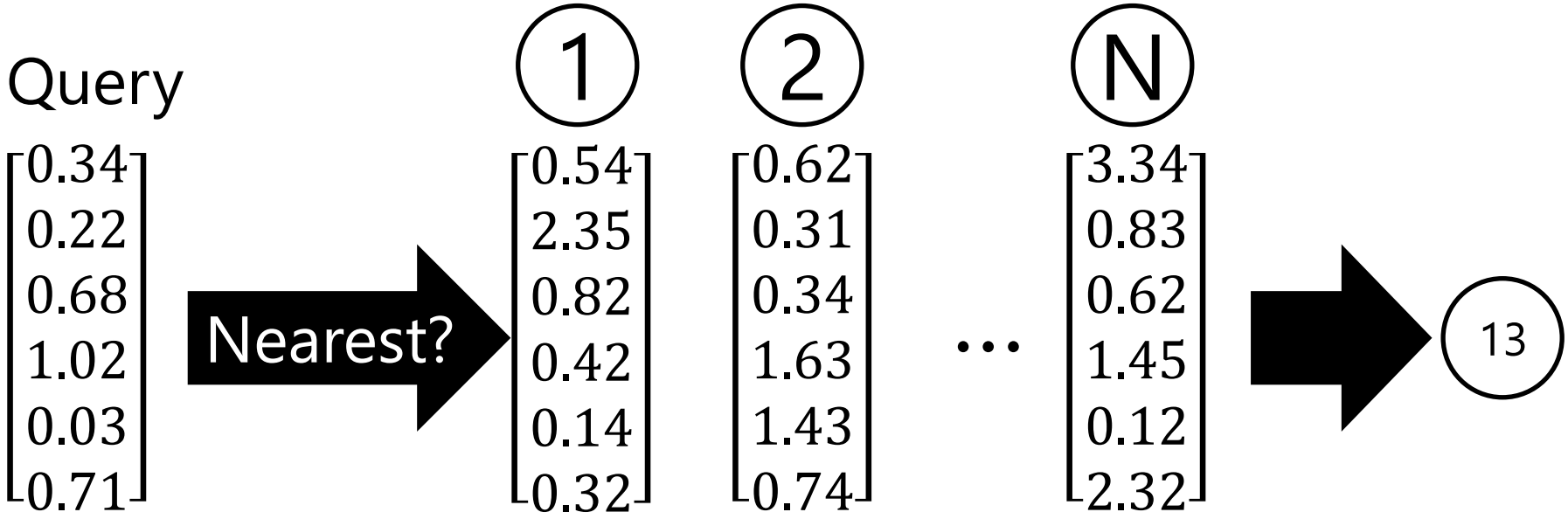
- N個のベクトルがある

最近傍探索



- N個のベクトルがある
- クエリベクトルが与えられたとき，
  一番近いベクトルを探す
- 計算機科学における基本的な問題

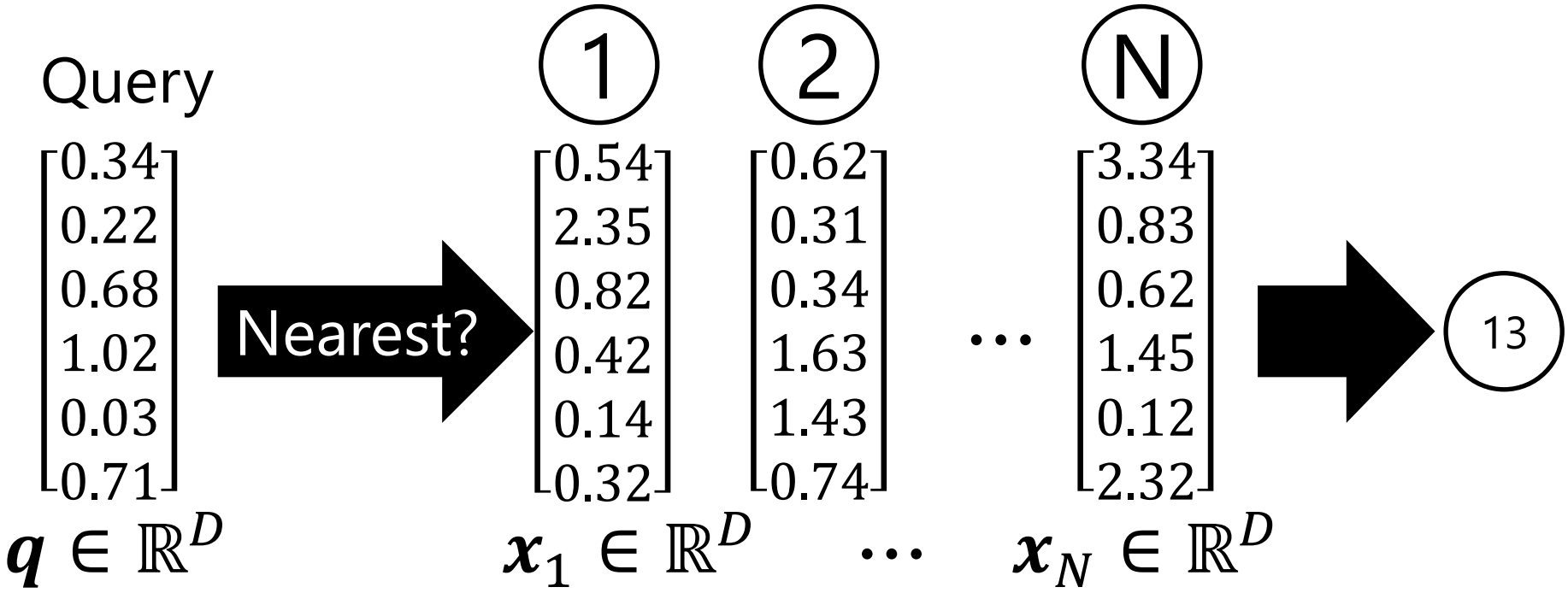
最近傍探索



$$n^* = \underset{n \in \{1,\ldots,N\}}{\operatorname{argmin}} \|\boldsymbol{q} - \boldsymbol{x}_n\|_2^2$$

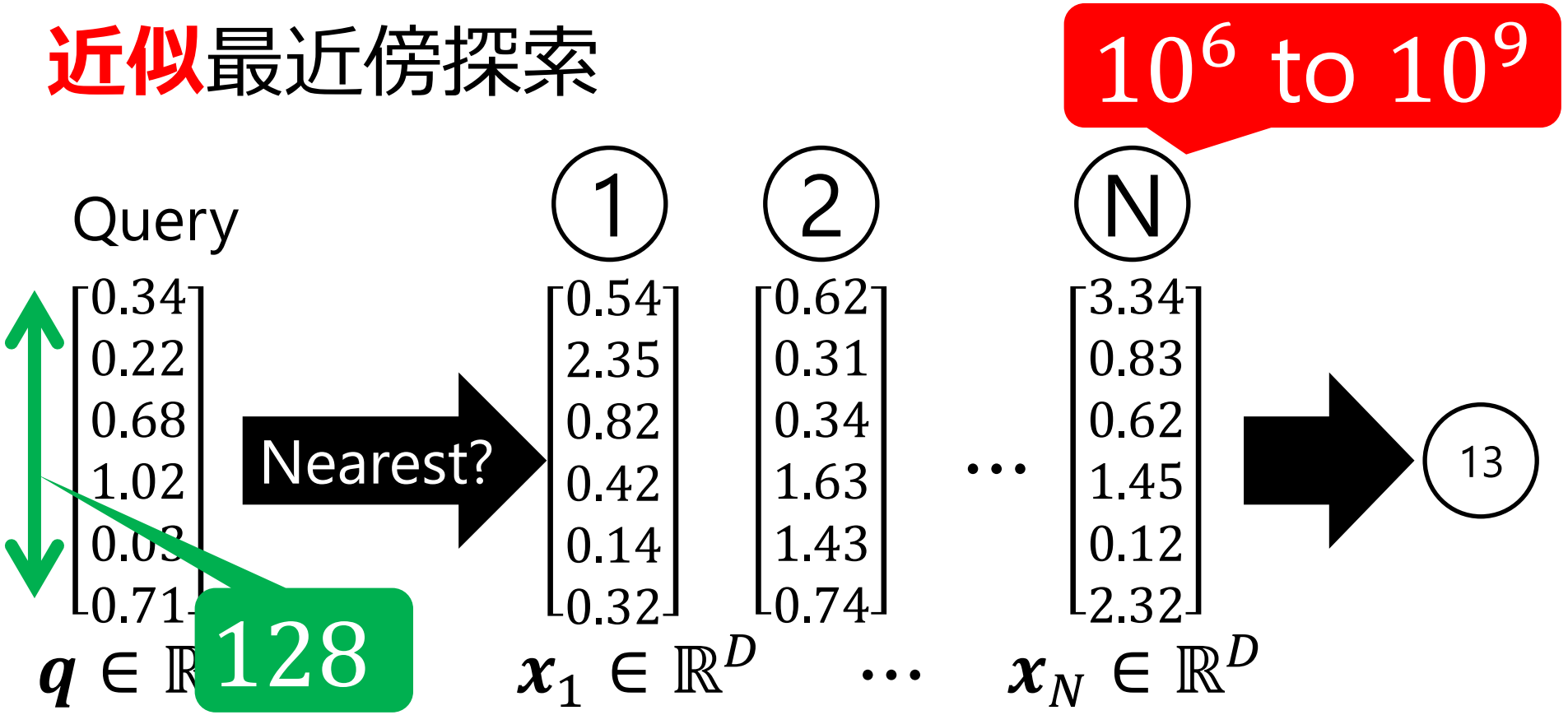
- 真面目に線形に全探索：$O(DN)$
- 遅い

近似最近傍探索



- **近似**最近傍探索
- 厳密に最近傍でなくてもよいので，
  高速に解を求める
- 速度，メモリ，精度のトレードオフ

近似最近傍探索



10 ms

32GB RAM

- 今日紹介する手法の規模感
- **大規模**な近似最近傍探索を扱う

N

10^9

billion-scale

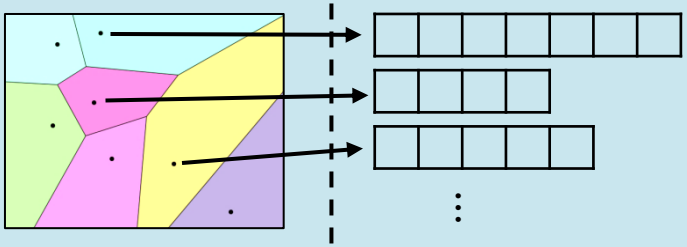
10^6

million-scale

転置インデクス+データ圧縮

ざっくり空間分割
- k-means
- 複数k-means
- PQ/OPQ
- etc…

データ圧縮
- 生データのまま
- Scalar quantization
- PQ/OPQ
- etc…



圧縮データを直接探索

PQTable

Multi hash table

生データを直接扱う：
精度○，メモリ効率×

Locality Sensitive Hashing (LSH)系

Tree系 / Space Partitioning系

Graph探索系

データを圧縮する：
精度△，メモリ効率○

ルックアップ系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → ID: 2 / ID: 123 → ADC 線形探索

ハミング系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → 0 1 0 0 → ハミング 線形探索

N

10^9

billion-scale

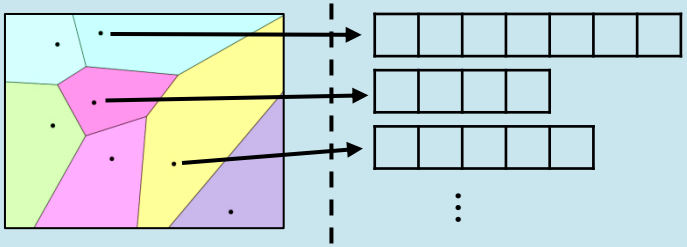
10^6

million-scale

転置インデクス+データ圧縮

ざっくり空間分割
- k-means
- 複数k-means
- PQ/OPQ
- etc…

データ圧縮
- 生データのまま
- Scalar quantization
- PQ/OPQ
- etc…



圧縮データを直接探索

PQTable

Multi hash table

生データを直接扱う：
精度○，メモリ効率×

Locality Sensitive Hashing (LSH)系

Tree系 / Space Partitioning系

Graph探索系

データを圧縮する：
精度△，メモリ効率○

ルックアップ系

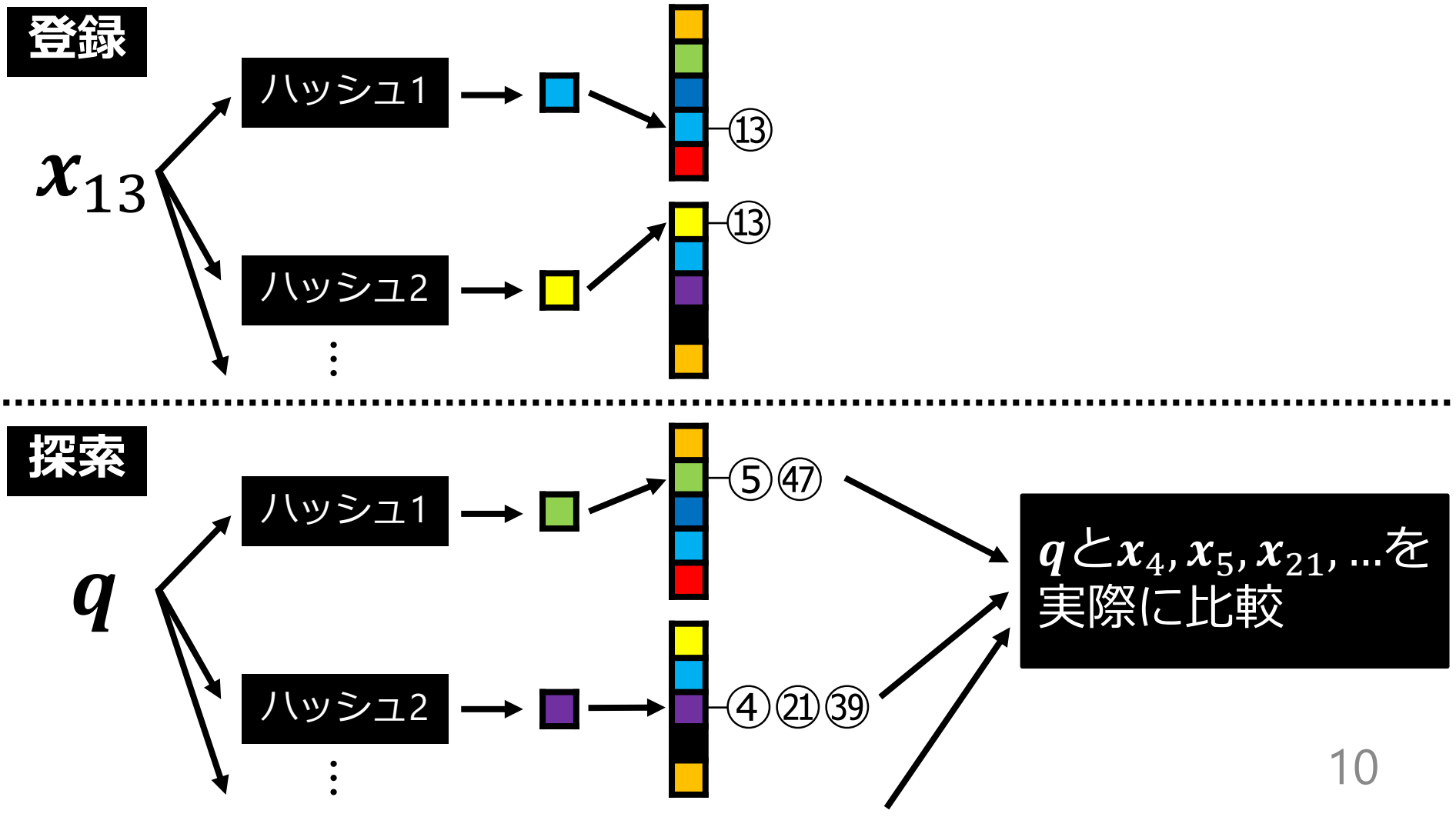
$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → ID: 2 / ID: 123 → ADC 線形探索

ハミング系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → 0 1 0 0 → ハミング 線形探索

Locality Sensitive Hashing (LSH)

➢近いベクトルを高い確率で同じ値に変換するような「ハッシュ関数」と，問い合わせの「データ構造」



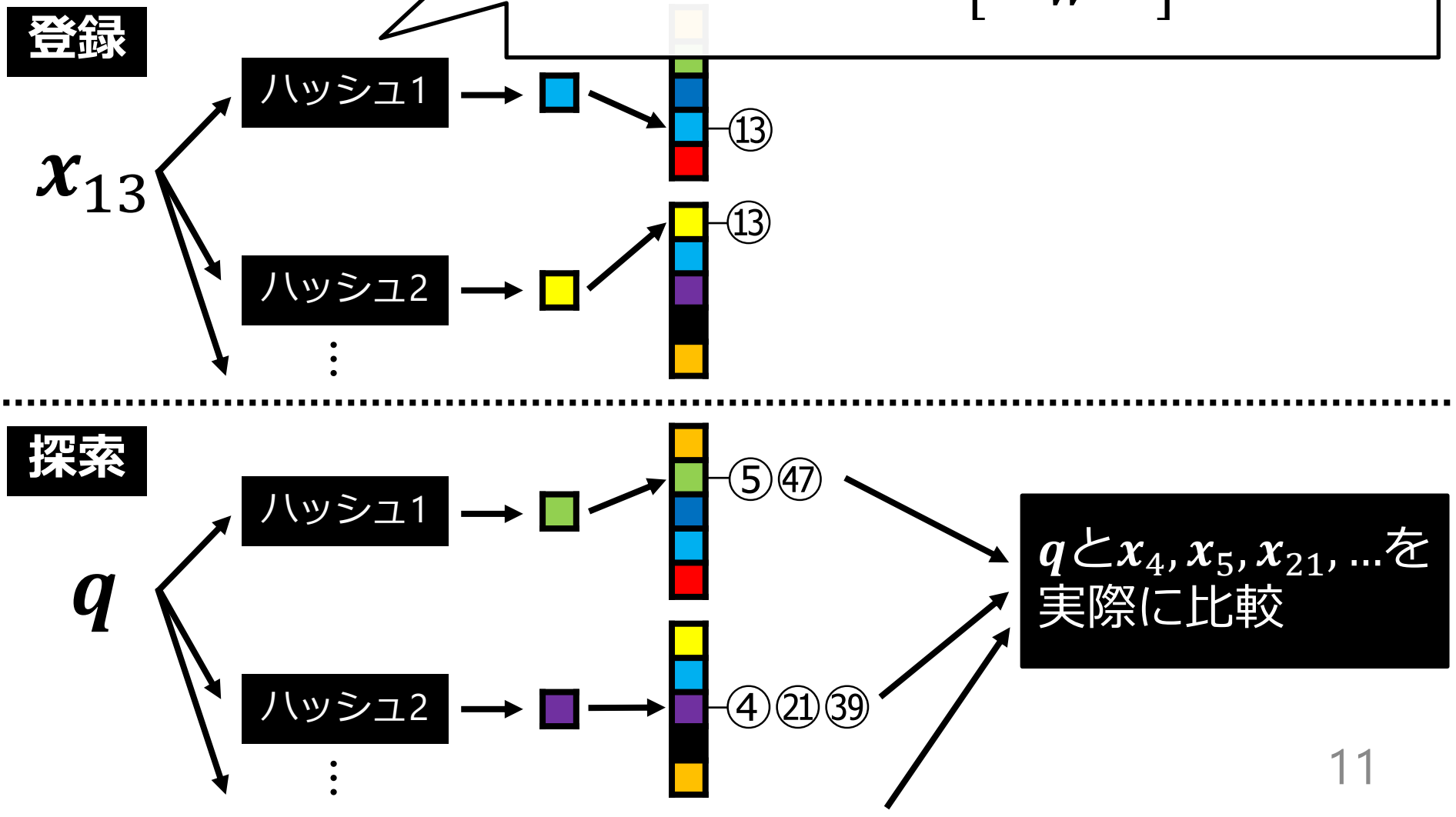
Locality Sensitive Hashing (LSH)

➢近いベクトルを高い確率で同じ値に変換するような「ハッシュ関数」と，問い合わせのデータ構造」

例えばランダムな射影 [Dater+, SCG 04]

$$H(\boldsymbol{x}) = [h_1(\boldsymbol{x}), \ldots, h_M(\boldsymbol{x})]^T$$

$$h_m(\boldsymbol{x}) = \left\lfloor \frac{\boldsymbol{a}^T\boldsymbol{x} + b}{W} \right\rfloor$$

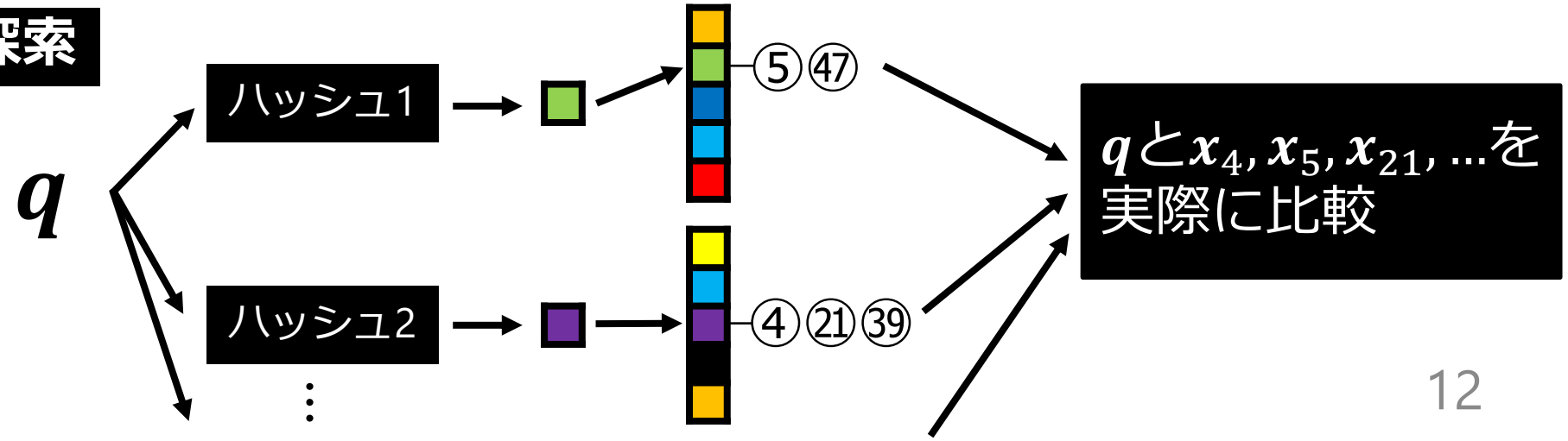


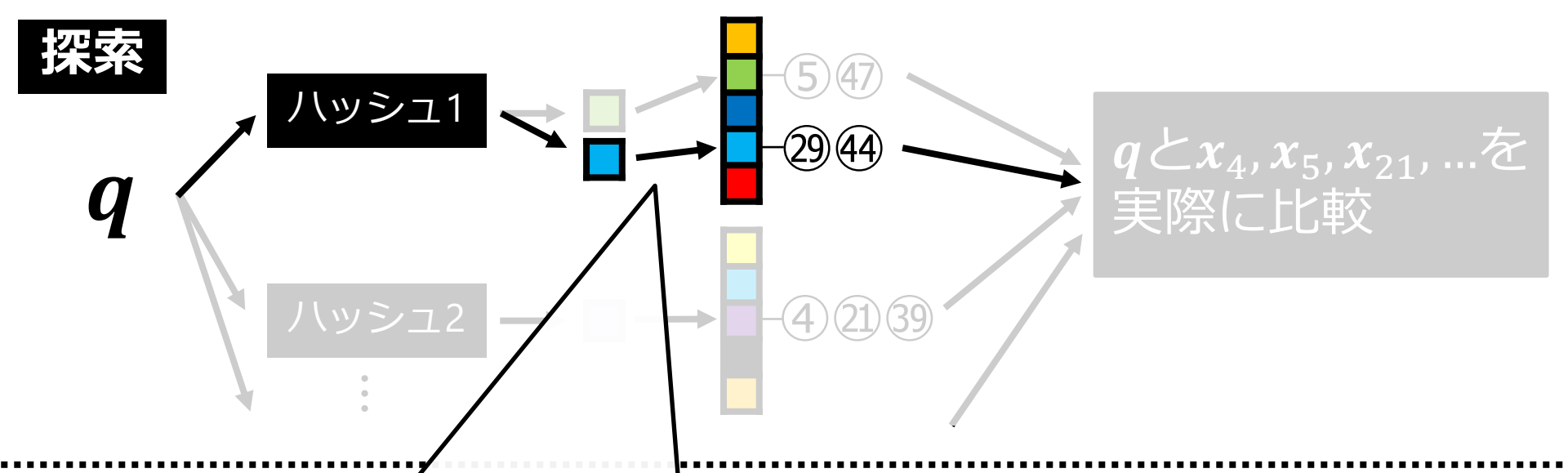
☺:
- 数学的な解析が容易
- 理論分野では今でも解析が盛ん

☹:
- 精度を上げるにはテーブルがたくさん必要
- また、元のデータ（$\boldsymbol{x}_n$）を保持する必要有り
- なのでメモリ消費が多い
- 実データではデータ依存手法(PQ等)のほうが精度良い

・・・なので、近年のCVの論文では関連研究として昔の手法扱いされるだけの場合が多かった☹☹☹

探索

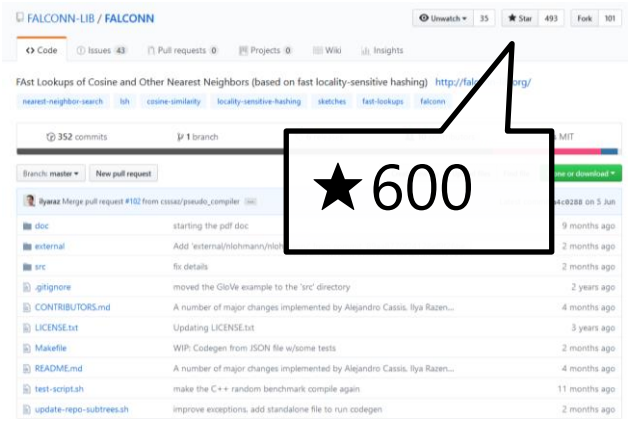




- 「次の候補」も考慮すると実用的なメモリ消費でいける（Multi-Probe [Lv+, VLDB 07]）
  - 豆知識：この考え方はInverted Multi-Index(後述)におけるMulti-Sequence Algorithmと同じ
- これを元に作られたライブラリ：**FALCONN**

Falconn

https://github.com/falconn-lib/falconn




```
$> pip install FALCONN

table = falconn.LSHIndex(params_cp)
table.setup(X-center)
query_object = table.construct_query_object()
# query parameter config here
query_object.find_nearest_neighbor(Q-center, topk)
```

- パラメータ設定がややめんどくさい
- **データ追加が高速**（後述のannoyやnmslibに比べ）
- なのでその場でインデックスを構築する場合などに有利？

参考資料

- わかりやすいまとめスライド：CVPR 2014 Tutorial on Large-Scale Visual Recognition, Part I: Efficient matching, H. Jégou [link]
- 実用的なQ&A：FAQ in Wiki of FALCONN [link]
- 本発表で用いたハッシュ関数：M. Datar et al., “Locality-sensitive hashing scheme based on p-stable distributions,” SCG 2004.
- Multi-Probe：Q. Lv et al., “Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search”, VLDB 2007
- まとめ記事：A. Andoni and P. Indyk, “Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions,” Comm. ACM 2008

N

10^9

billion-scale

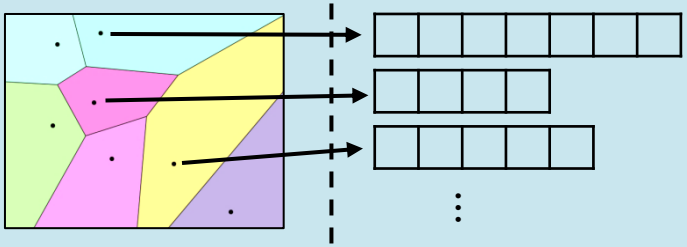
10^6

million-scale

転置インデクス+データ圧縮

ざっくり空間分割
- k-means
- 複数k-means
- PQ/OPQ
- etc…

データ圧縮
- 生データのまま
- Scalar quantization
- PQ/OPQ
- etc…



圧縮データを直接探索

PQTable

Multi hash table

生データを直接扱う：
精度○，メモリ効率×

Locality Sensitive Hashing (LSH)系

Tree系 / Space Partitioning系

Graph探索系

データを圧縮する：
精度△，メモリ効率○

ルックアップ系

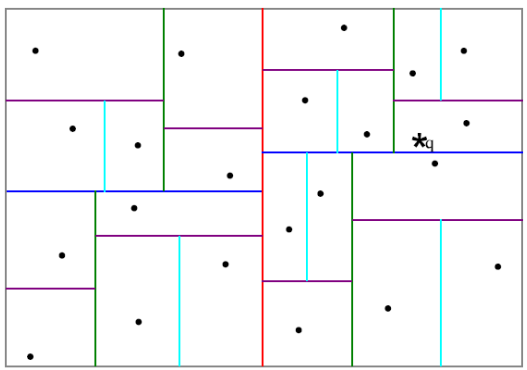
$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → ID: 2 / ID: 123 → ADC 線形探索

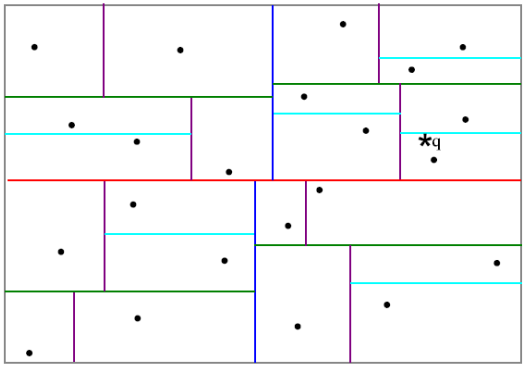
ハミング系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → 0 1 0 0 → ハミング 線形探索

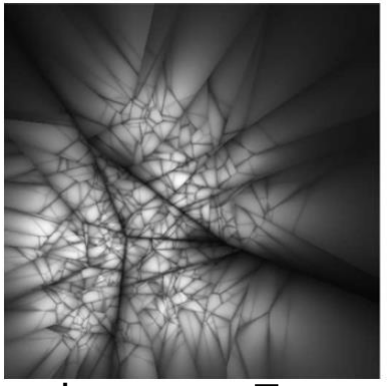
FLANN: Fast Library for Approximate Nearest Neighbors

画像は[Muja and Lowe, TPAMI 2014]から引用





Randomized KD Tree



k-means Tree

- Tree系の代表格。「Randomized KD Tree」と「k-means Tree」から良い方のアルゴリズムが自動的に選択される
- https://github.com/mariusmuja/flann

☺
  - 実装が使いやすく、00年代後半～10年代前半に非常に人気
  - OpenCVやPCLに含まれている

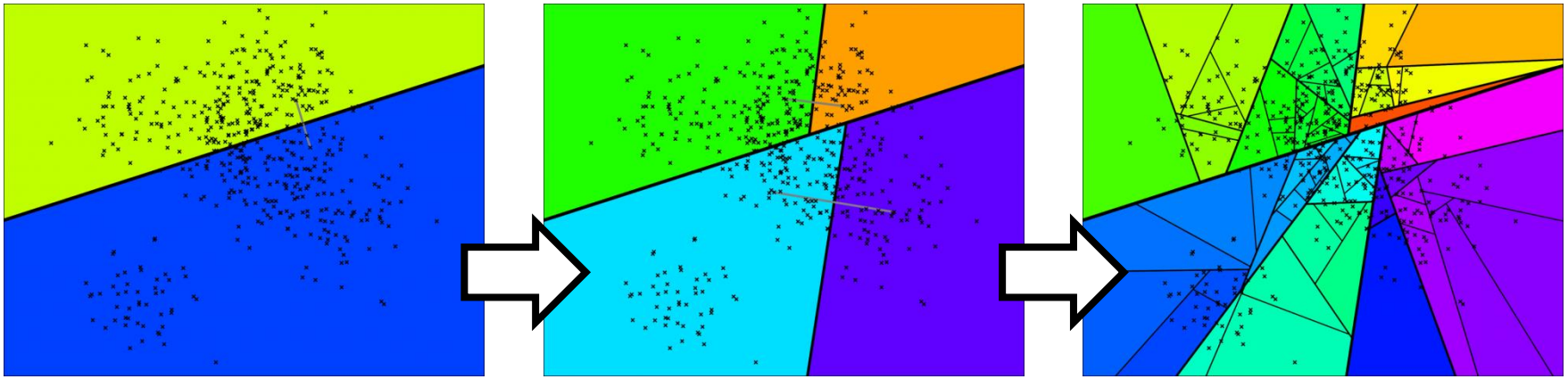
☹
  - 元のデータを保持するのでメモリ消費大
  - 2018年現在、コードのメンテナンスが止まっている

Annoy

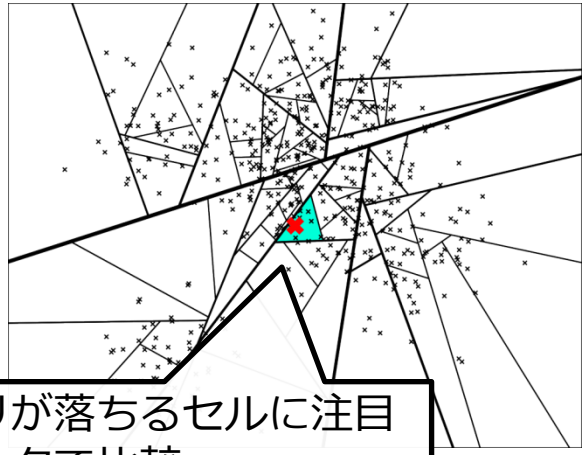
以下の画像は全て著者ブログポスト(https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html)より引用

「2-means tree」＋「複数trees」＋「優先度付きキューを共用」

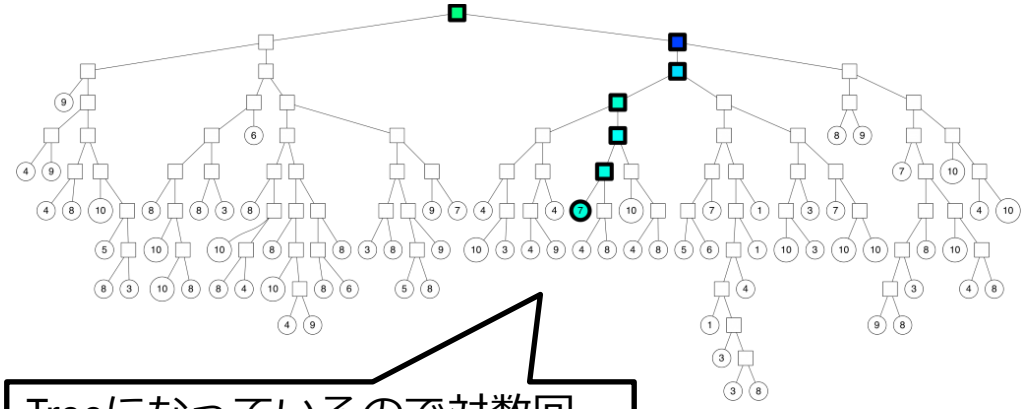
登録 ランダム二点選択→空間分割→階層的に



探索



- クエリが落ちるセルに注目
- 実データで比較



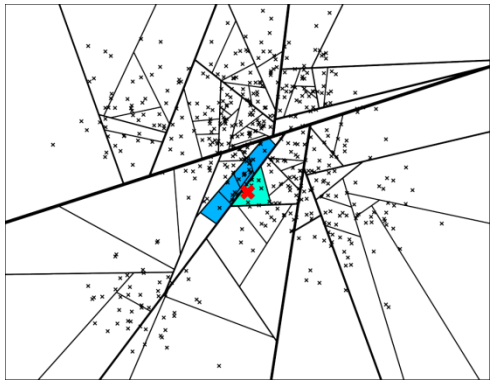
Treeになっているので対数回の比較でたどり着ける

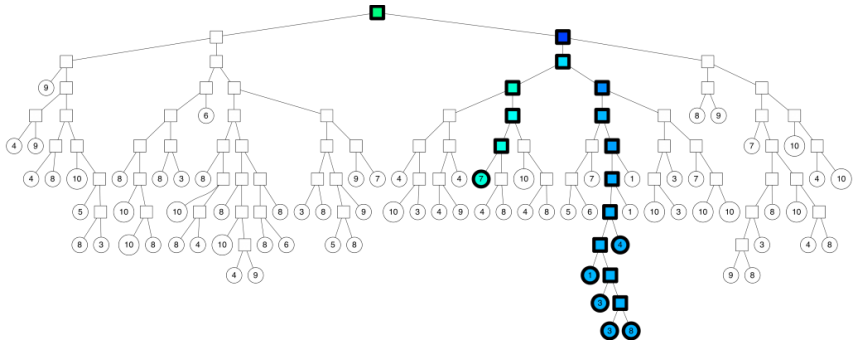
Annoy

以下の画像は全て著者ブログポスト(https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html)より引用

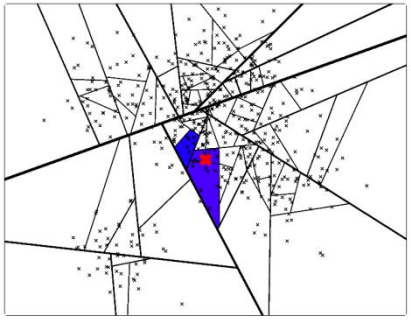
「2-means tree」＋「複数trees」＋「優先度付きキューを共用」

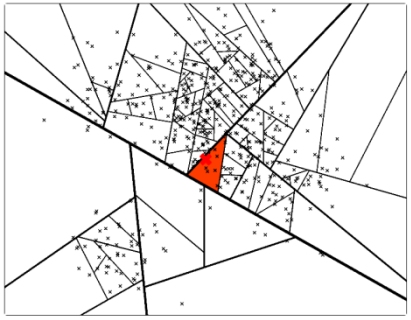
工夫1 より点数が欲しい場合は距離の優先度付きキュー

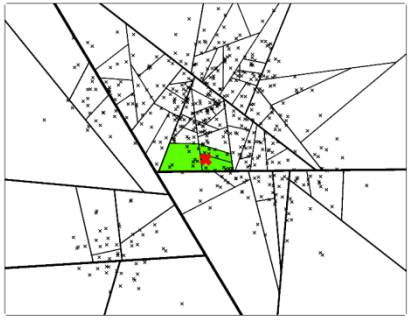


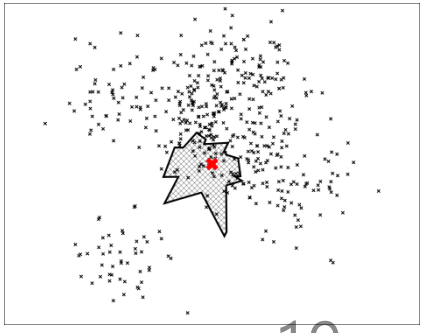


工夫2 複数treeを用意して精度上げる（優先度付きキューは共用）







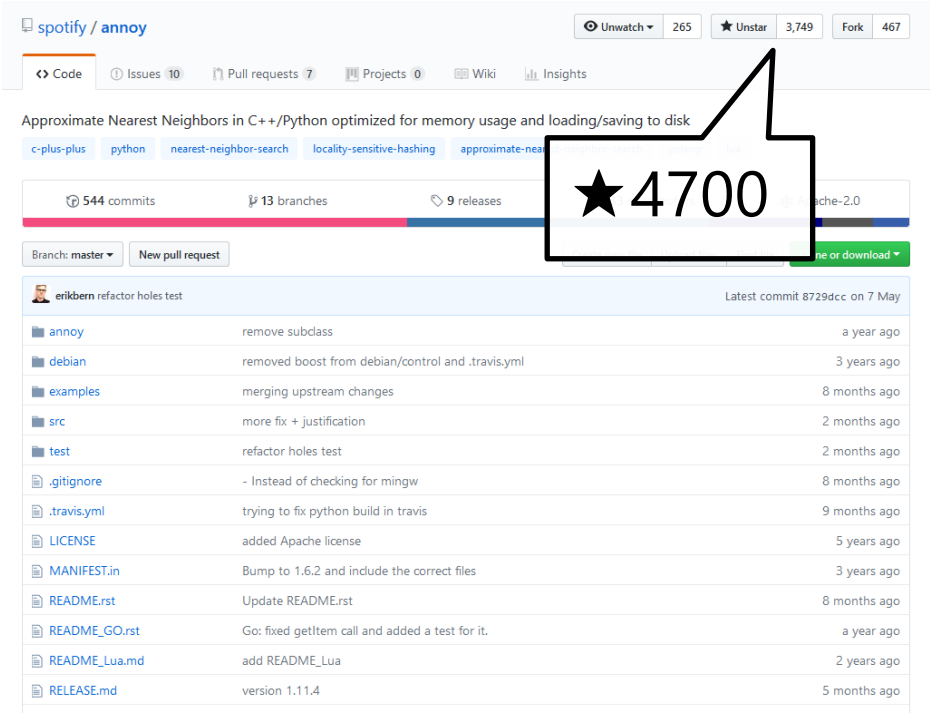


Annoy

https://github.com/erikbern/annoy

```
$> pip install annoy
```

```
t = AnnoyIndex(D)
for n, x in enumerate(X):
    t.add_item(n, x)
t.build(n_trees)

t.get_nns_by_vector(q, topk)
```



☺

- Spotify技術者が開発。Spotifyで実際に使われている
- よく整備されていて安定している印象
- パラメータが少なく直感的で、インタフェースがシンプル
- 近年のmillion-scaleのANNのベースライン的な扱い
- 保存されたデータをmmapで読むので、たくさんのプロセスから読むときに適しているらしい

☹

- 実データを保持するのでメモリ消費大
- ちゃんとした技術詳細資料がない

コメント：Annoyもflannもlshも後述するIVFADCも、最初にざっくり空間分割し、そこで注目したものだけ詳しく調べるという意味では似ている

N

10^9

billion-scale

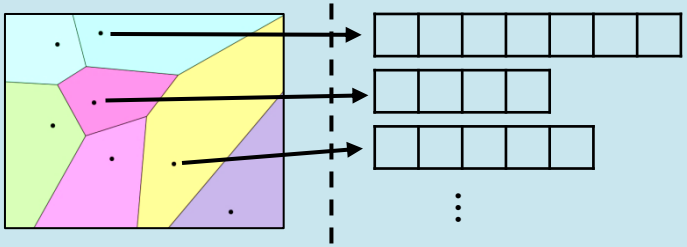
10^6

million-scale

転置インデクス+データ圧縮

ざっくり空間分割
- k-means
- 複数k-means
- PQ/OPQ
- etc…

データ圧縮
- 生データのまま
- Scalar quantization
- PQ/OPQ
- etc…



圧縮データを直接探索

PQTable

Multi hash table

生データを直接扱う：
精度○，メモリ効率×

Locality Sensitive Hashing (LSH)系

Tree系 / Space Partitioning系

Graph探索系

データを圧縮する：
精度△，メモリ効率○

ルックアップ系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → ID: 2 / ID: 123 → ADC 線形探索

ハミング系

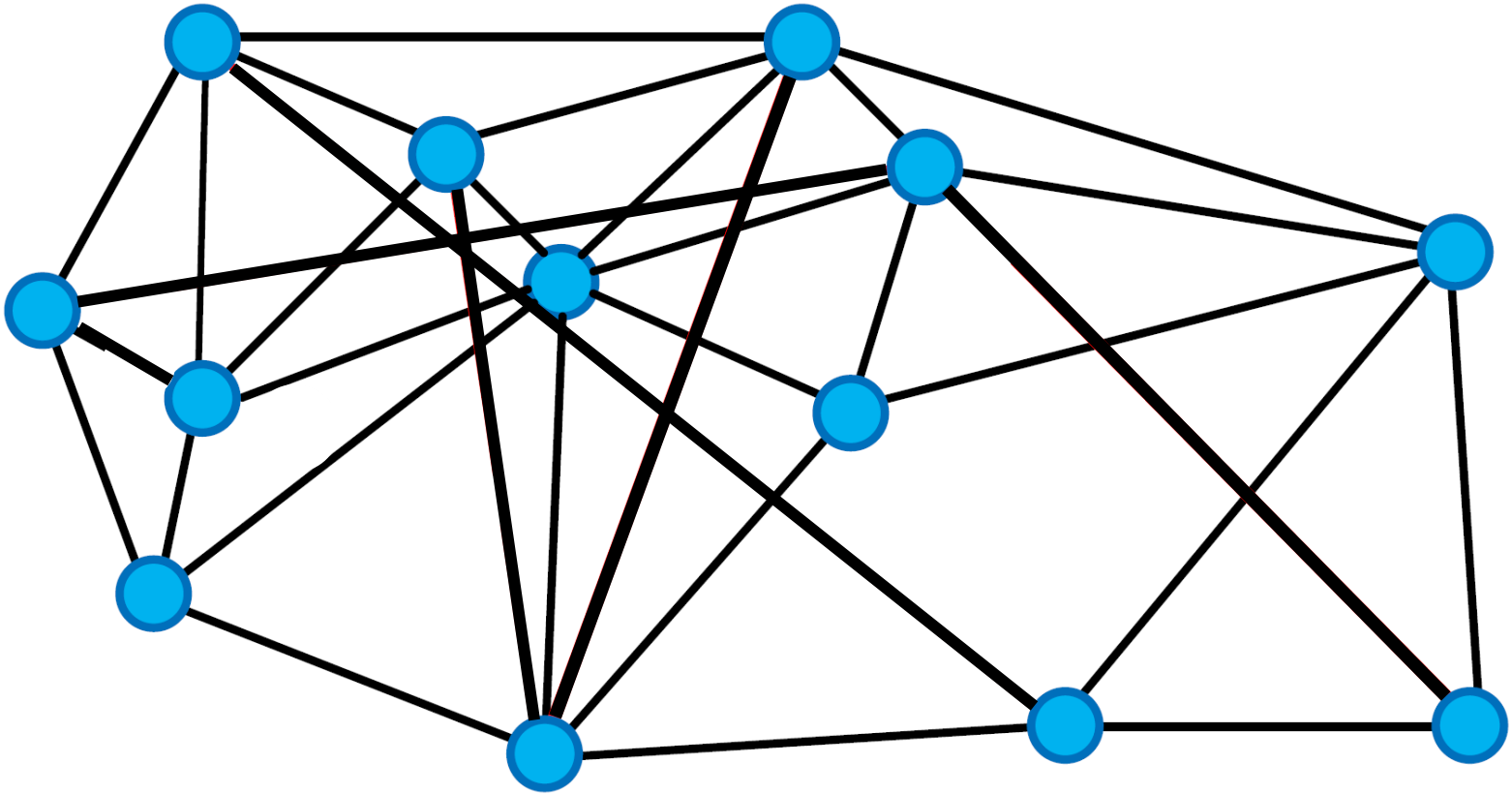
$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → 0 1 0 0 → ハミング 線形探索

Graph探索系

- **Navigable Small World Graphs (NSW)**
- データ点をノードとするグラフを作っておく
- ランダム点からスタートし、「繋がっているノードのうちクエリに近いものへ移動」を繰返す（greedyなtraverse）

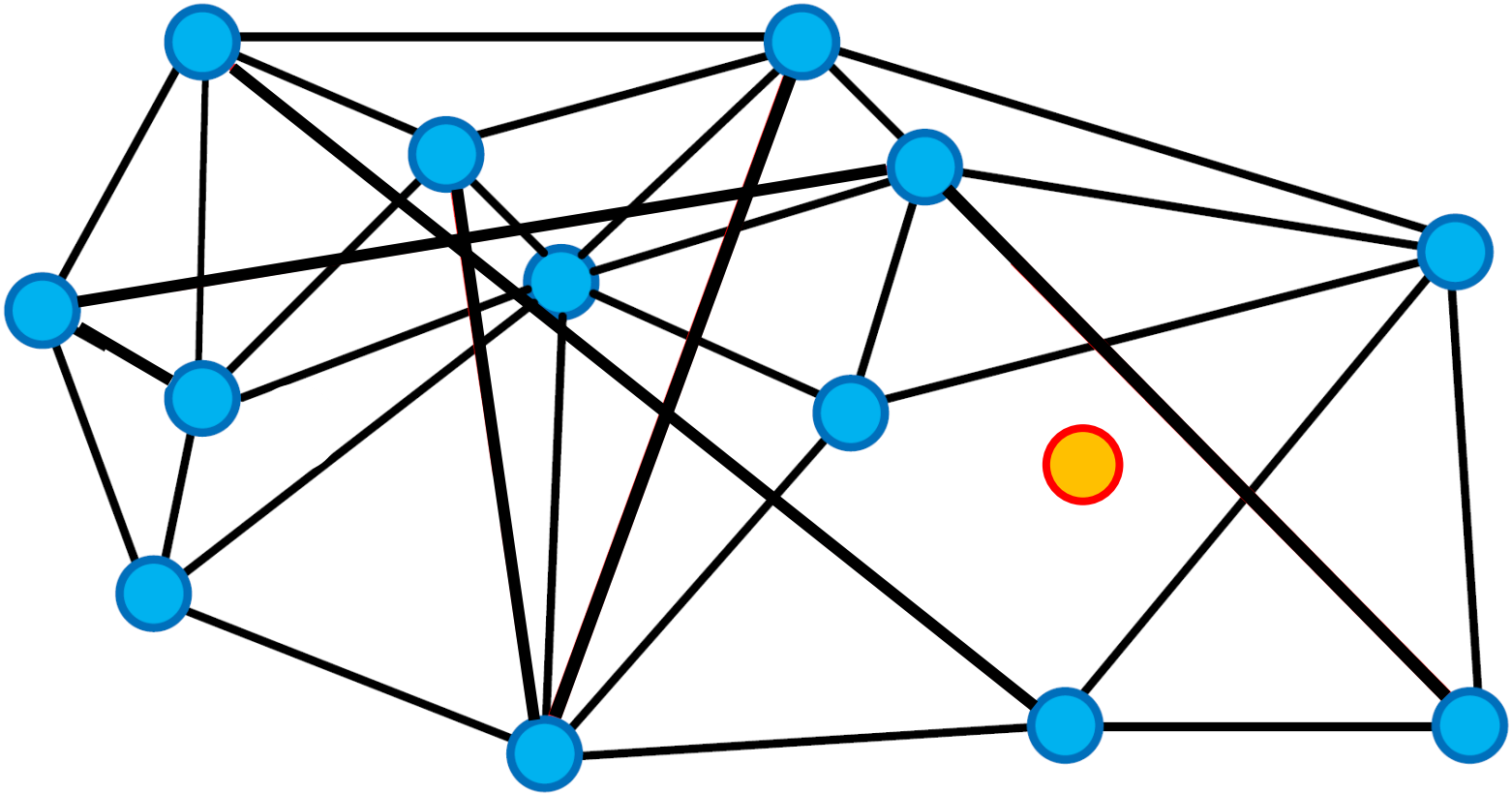
- この階層バージョン（Hierarchical NSW: HNSW）が、million-scaleのデータに対し極めて高速・高精度であることが2017年ごろにわかってきた
- 特に、nmslibというライブラリに実装されているものが2018年現在のmillion-scaleの決定版
- Faiss (後述)にも実装された

登録 画像は[Malkov+, Information Systems, 2013]から引用



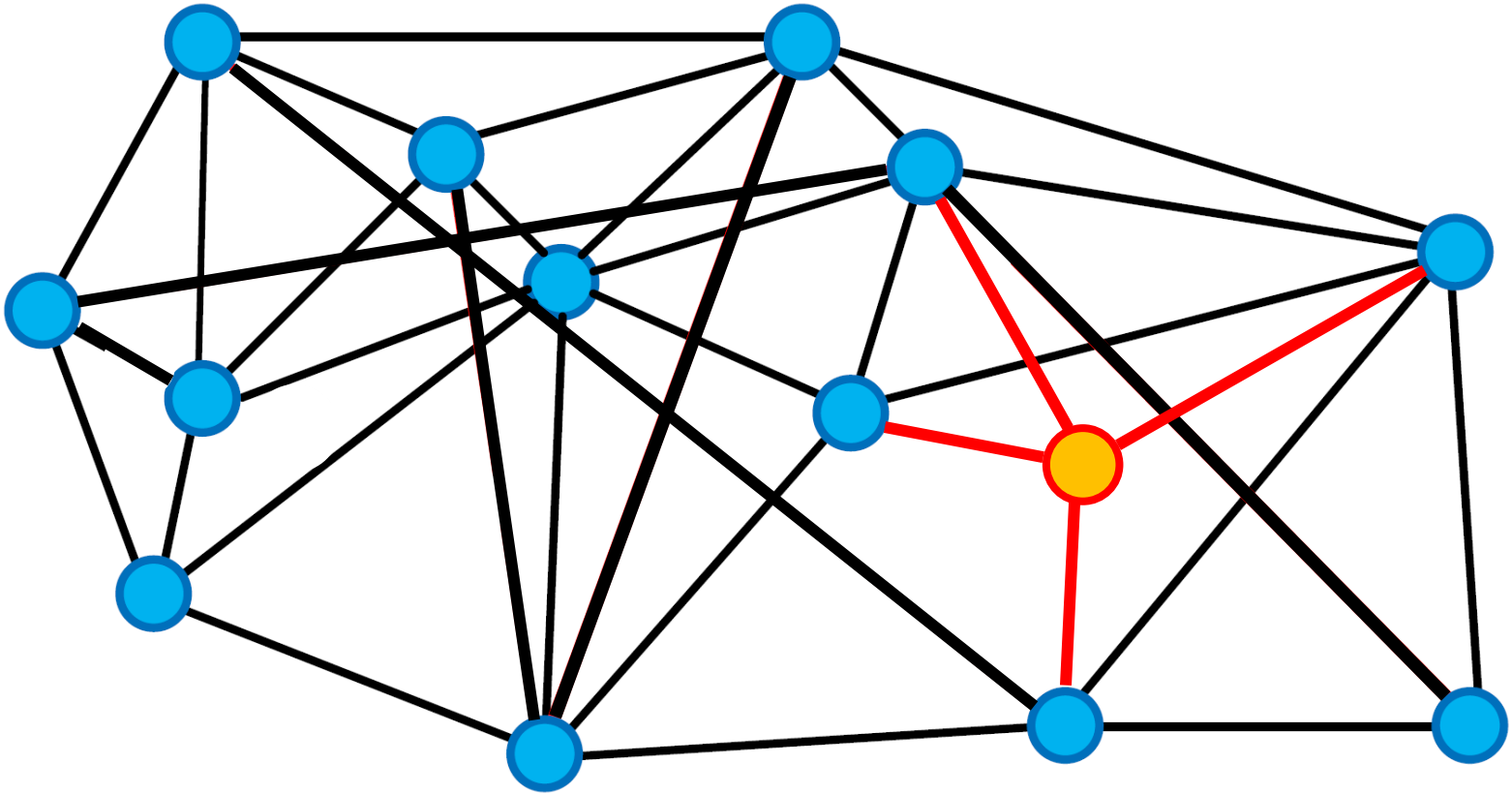
- データ点がノード
- 新しい入力データ点に対し、近傍のものにリンクを張る、としてグラフを作っていく

登録 画像は[Malkov+, Information Systems, 2013]から引用



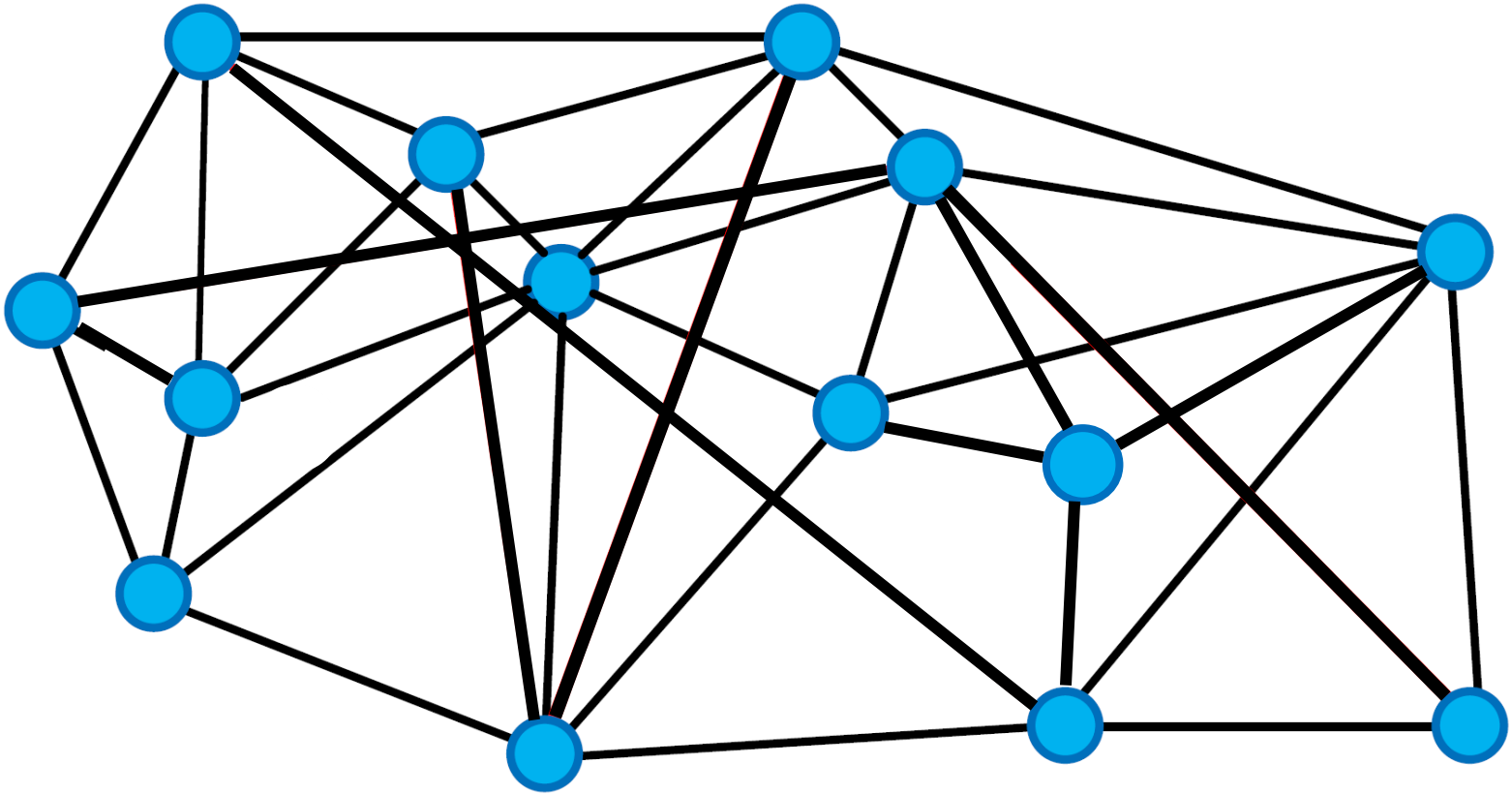
- データ点がノード
- 新しい入力データ点に対し、近傍のものにリンクを張る、としてグラフを作っていく

登録 画像は[Malkov+, Information Systems, 2013]から引用



- データ点がノード
- 新しい入力データ点に対し、近傍のものにリンクを張る、としてグラフを作っていく

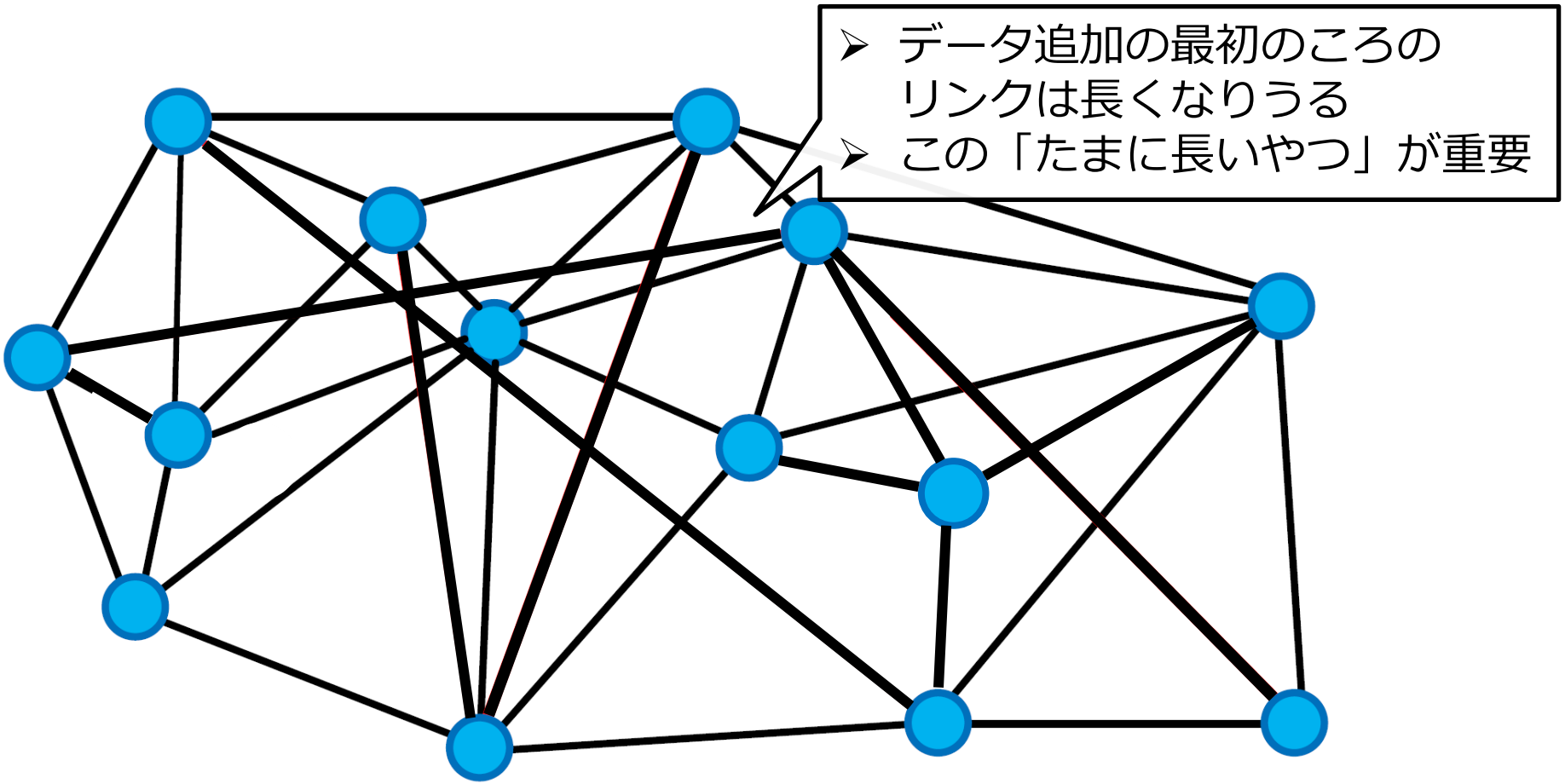
登録 画像は[Malkov+, Information Systems, 2013]から引用



- データ点がノード
- 新しい入力データ点に対し、近傍のものにリンクを張る、としてグラフを作っていく

登録

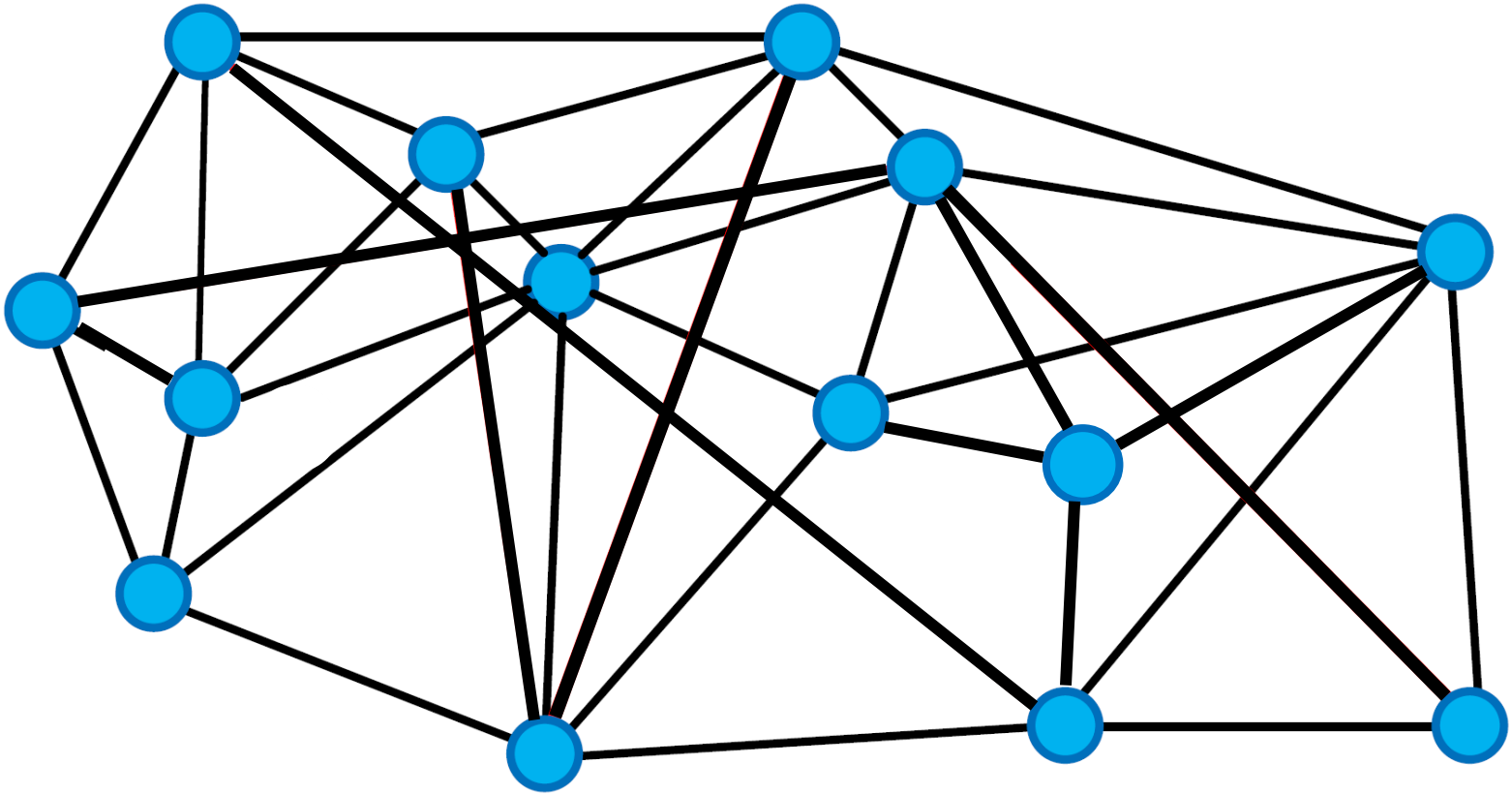
画像は[Malkov+, Information Systems, 2013]から引用



- データ点がノード
- 新しい入力データ点に対し、近傍のものにリンクを張る、
としてグラフを作っていく

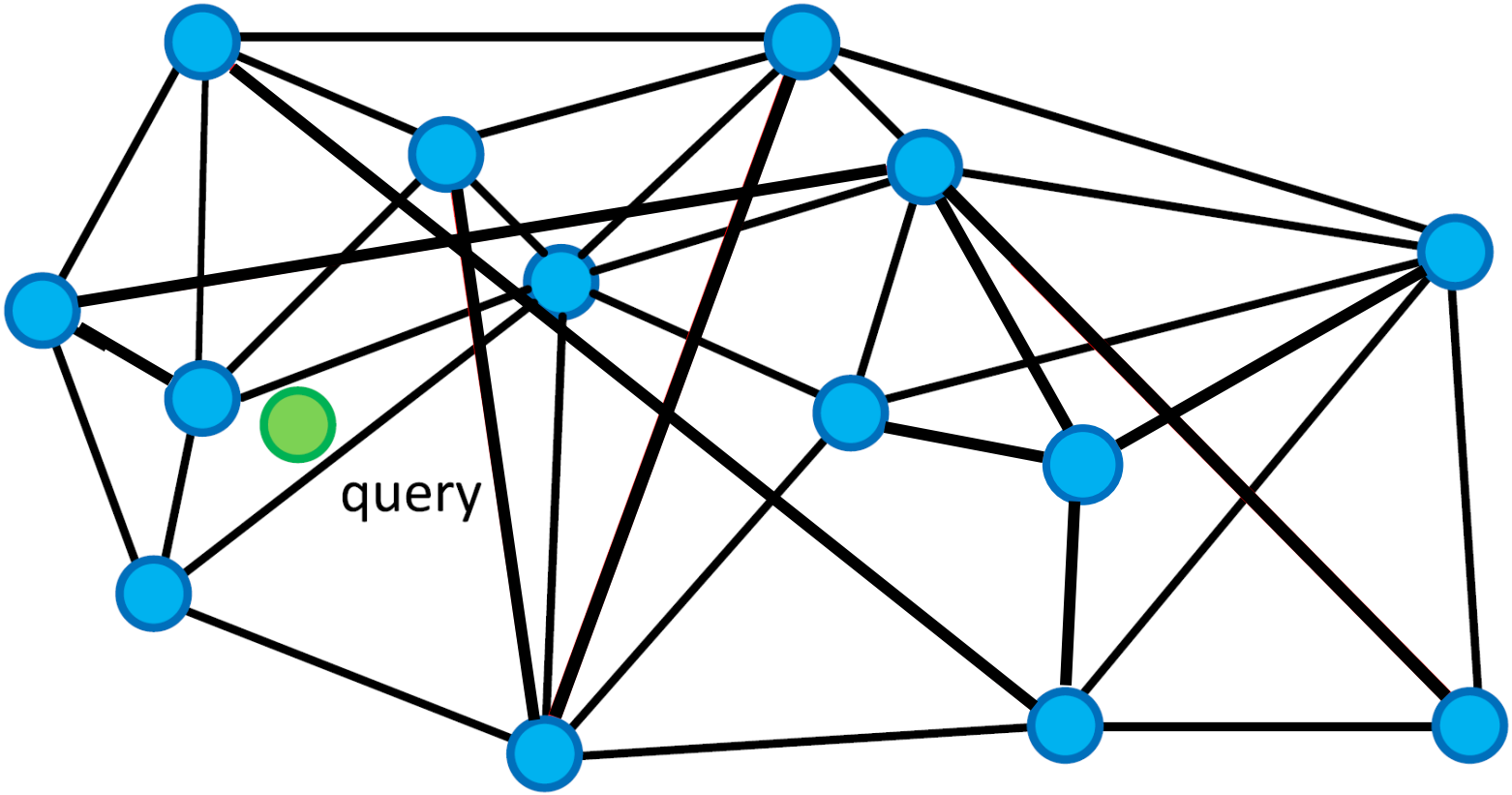
探索

画像は[Malkov+, Information Systems, 2013]から引用



探索

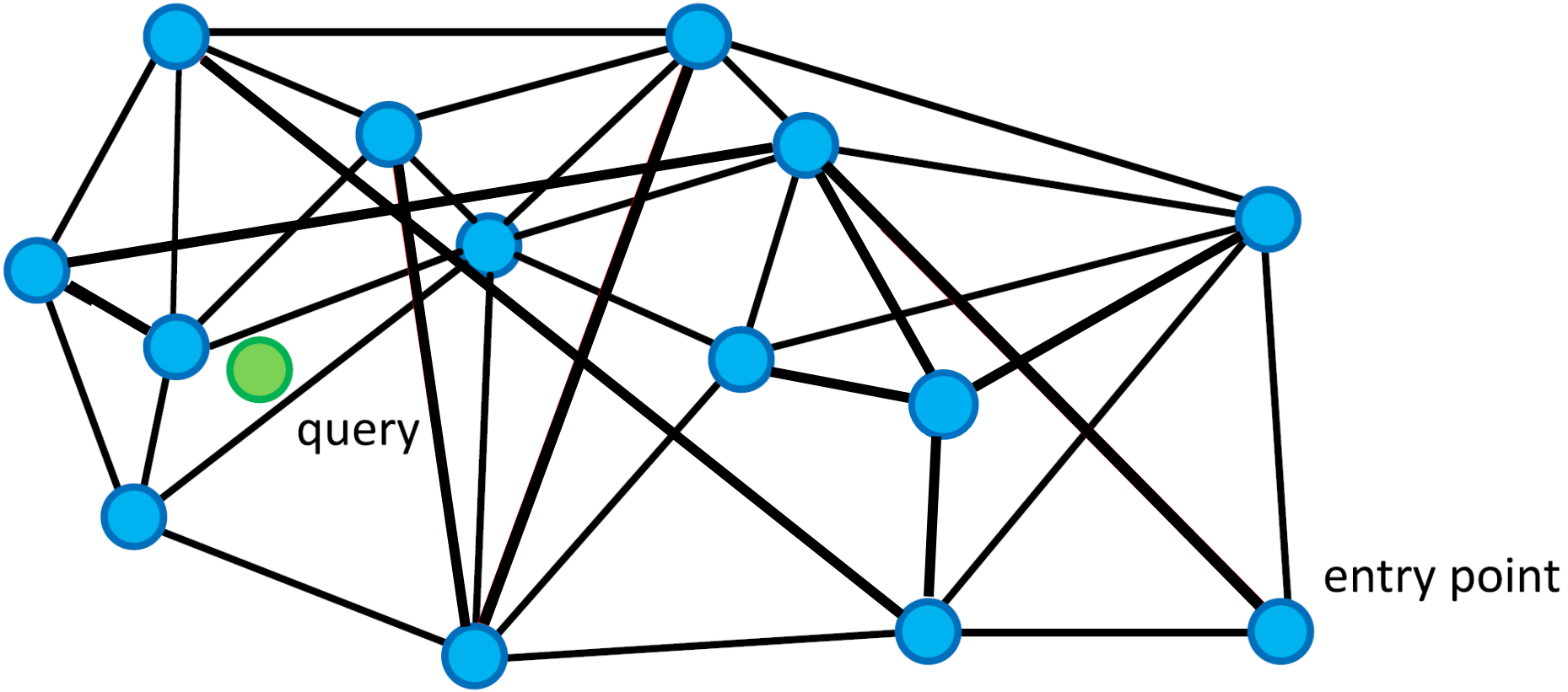
画像は[Malkov+, Information Systems, 2013]から引用



- クエリが与えられる

探索

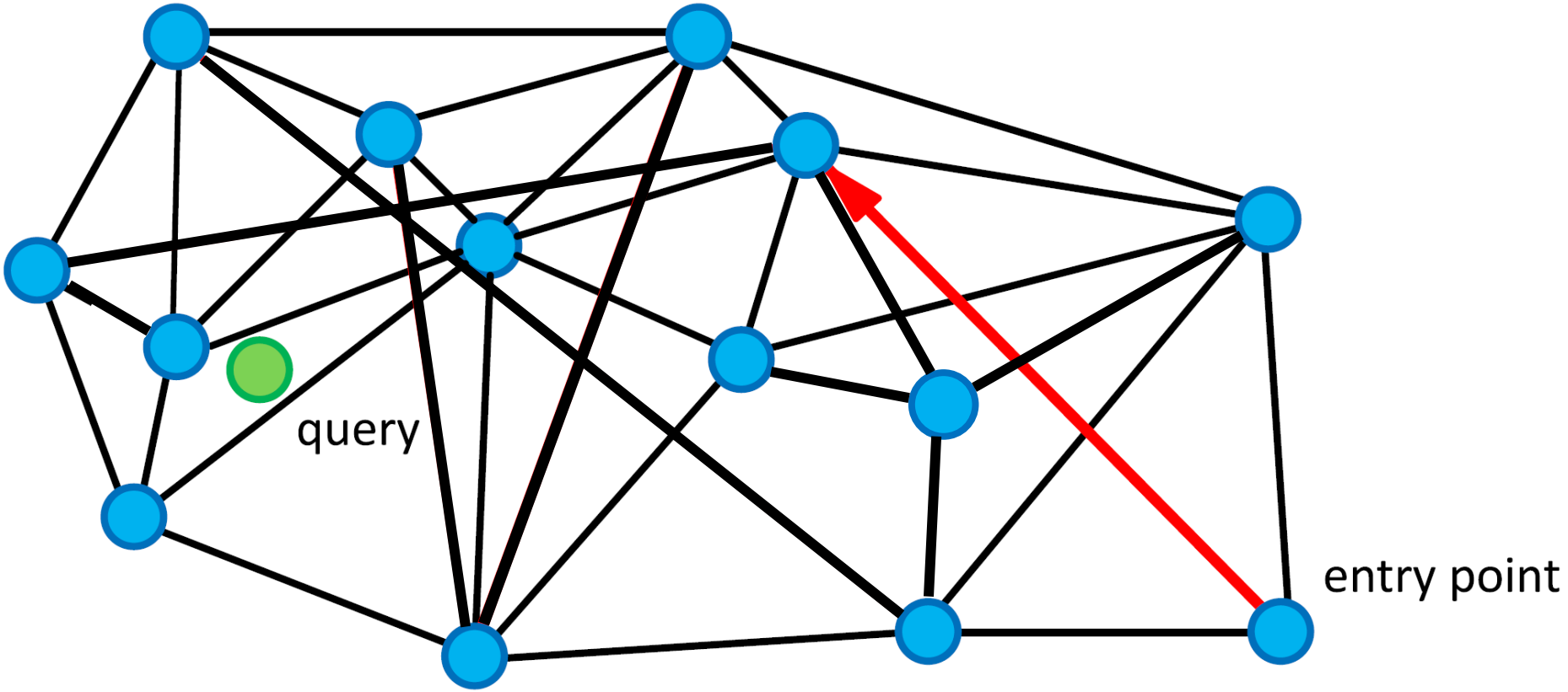
画像は[Malkov+, Information Systems, 2013]から引用



- クエリが与えられる
- ランダム点から出発

探索

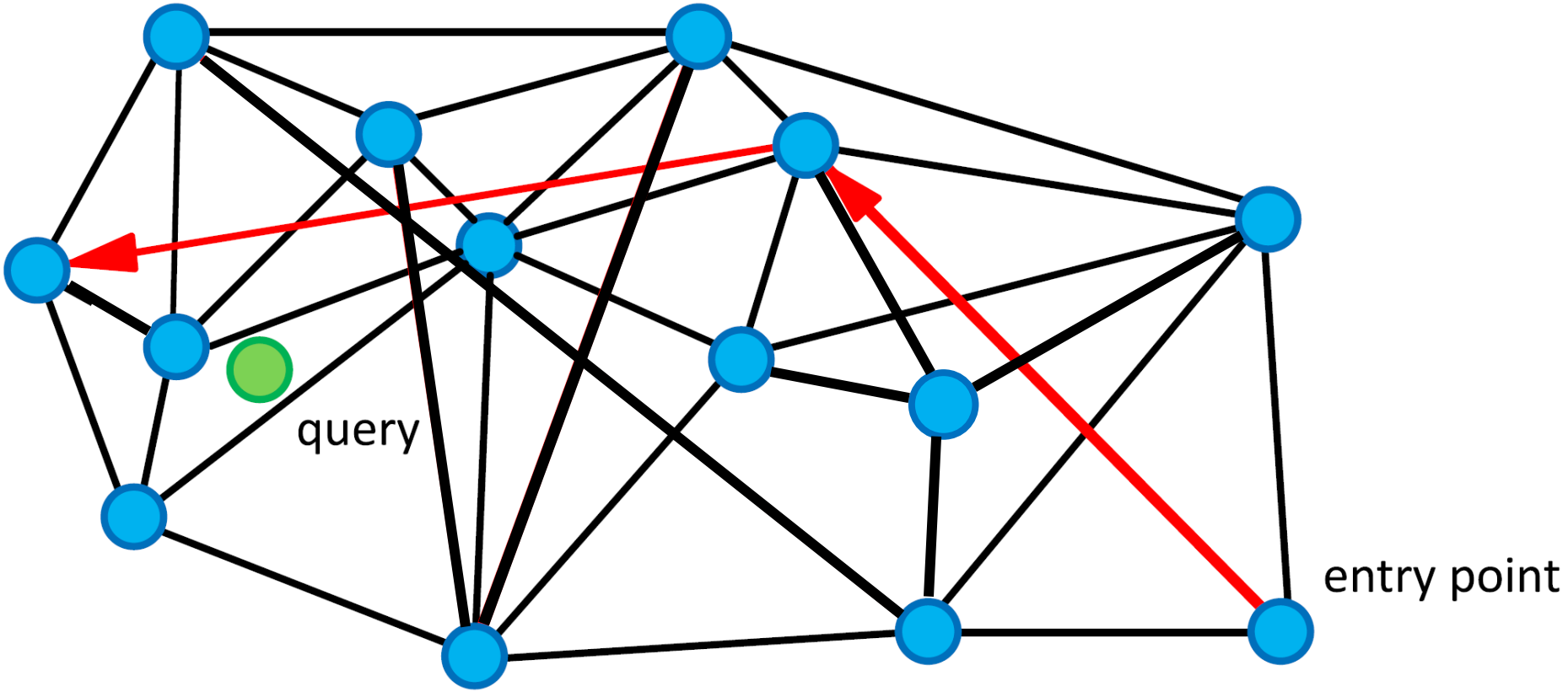
画像は[Malkov+, Information Systems, 2013]から引用



- クエリが与えられる
- ランダム点から出発
- 繋がっているノードのうちクエリに近いものにgreedyに移動

探索

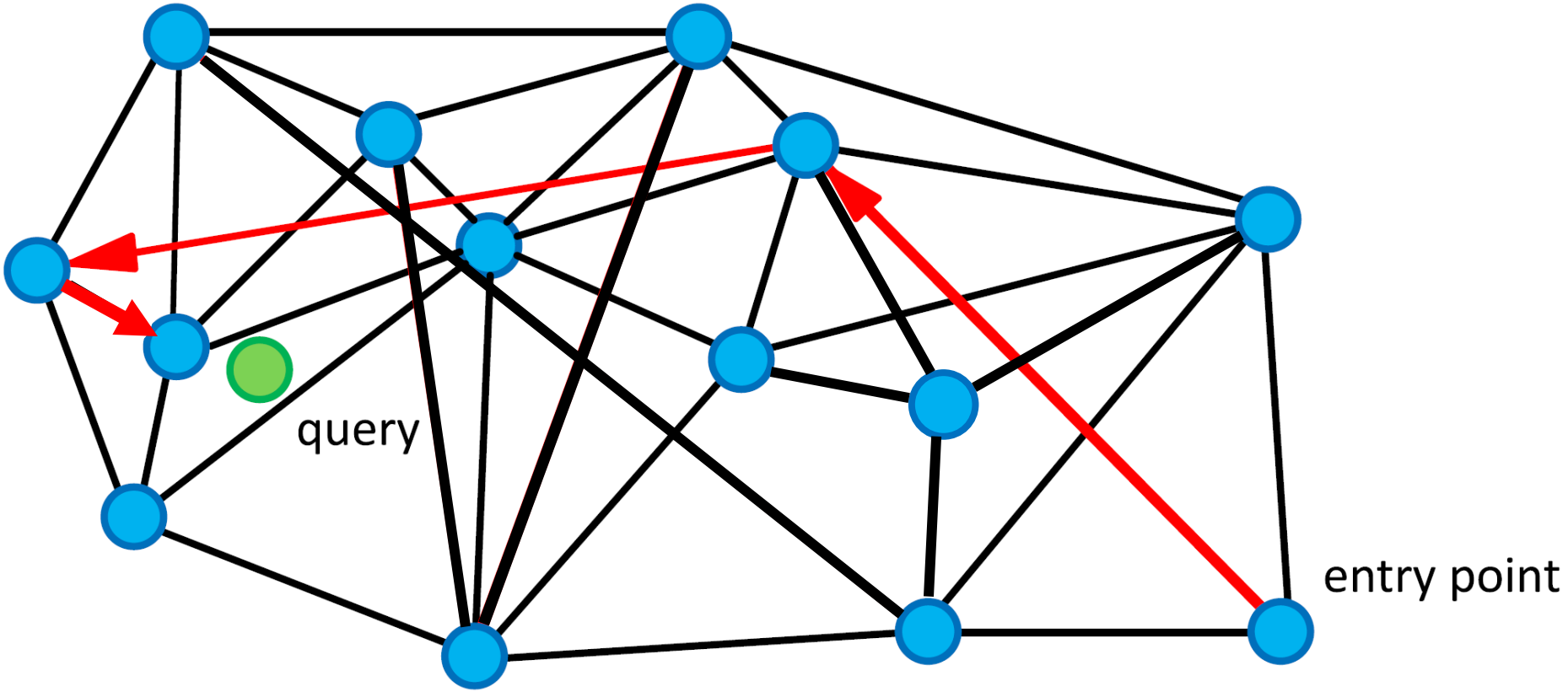
画像は[Malkov+, Information Systems, 2013]から引用



- クエリが与えられる
- ランダム点から出発
- 繋がっているノードのうちクエリに近いものにgreedyに移動

探索

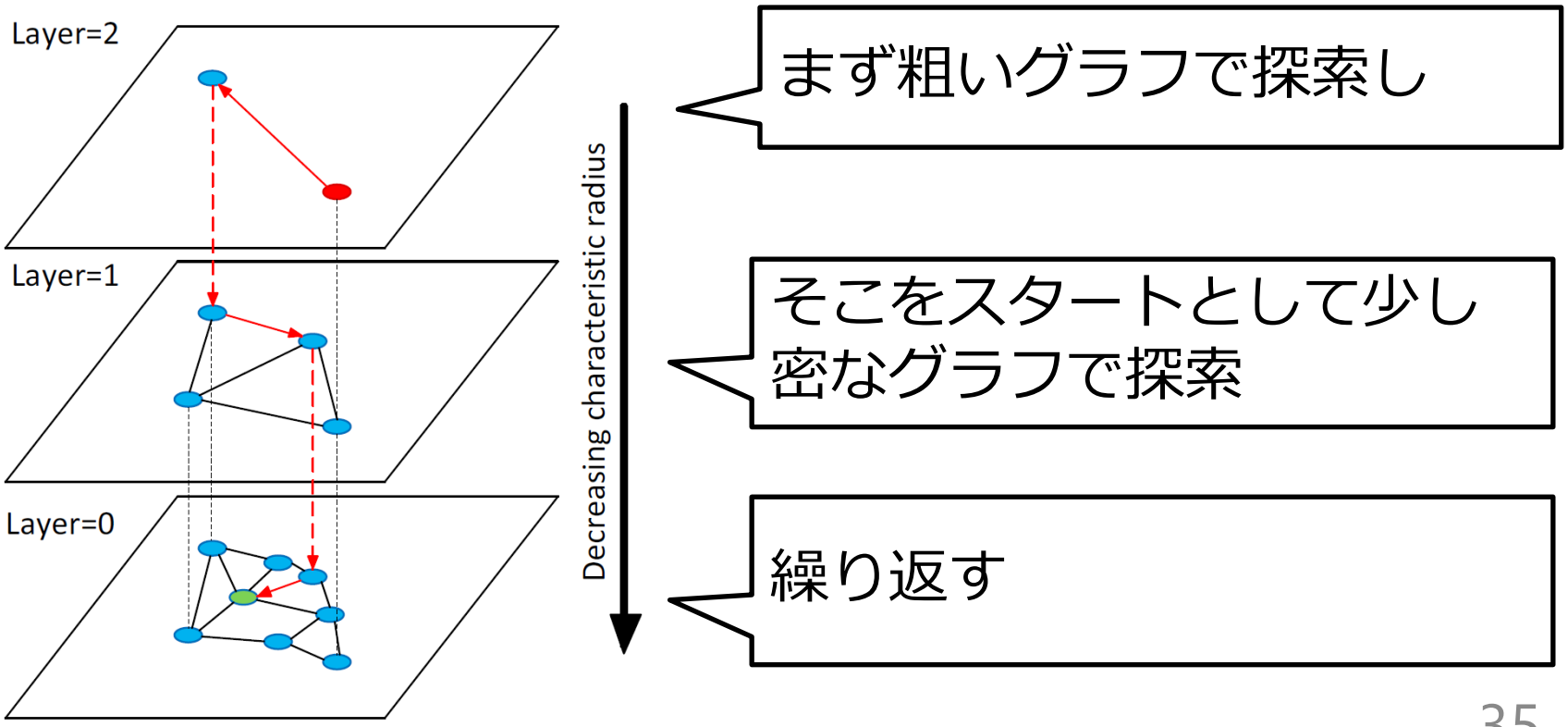
画像は[Malkov+, Information Systems, 2013]から引用



- クエリが与えられる
- ランダム点から出発
- 繋がっているノードのうちクエリに近いものにgreedyに移動

工夫 画像は[Malkov and Yashunin, TPAMI, 2019]から引用

- K-NNのとき、一度訪れたノードはもう考慮しない [Malkov+, Information Systems, 2013]
- グラフを**階層的**に作る（実データで非常に強い） [Malkov and Yashunin, TPAMI, 2019]

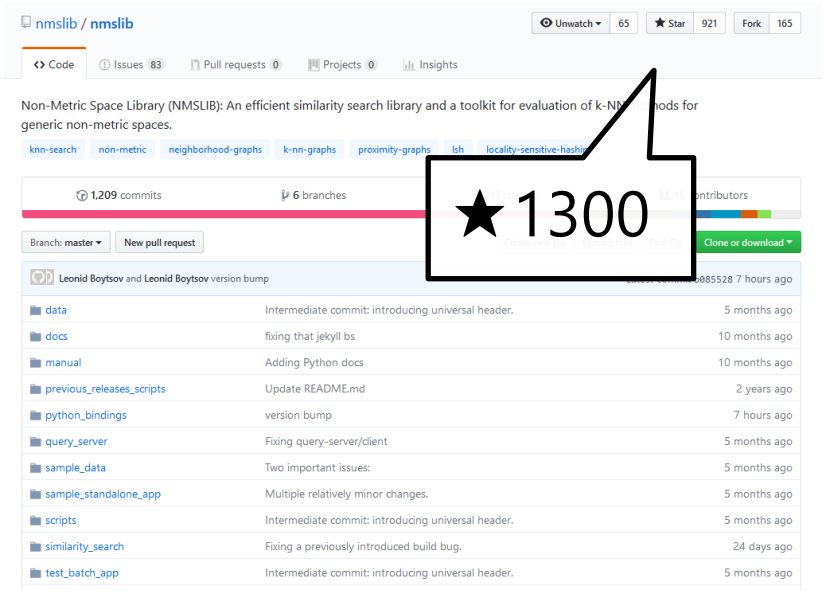


NMSLIB (Non-Metric Space Library)

https://github.com/nmslib/nmslib

```
$> pip install nmslib
```

```
index = nmslib.init(method='hnsw')
index.addDataPointBatch(X)
index.createIndex(params1)

index.setQueryTimeParams(params2)
index.knnQuery(q, topk)
```



☺

- "hnsw"は実データ（SIFTやディープ特徴量）で精度・速度バランスで2018年現在ベスト（メモリに載りさえすれば）
- インタフェースがシンプル
- データがメモリに載るのであればこれを使えばいい

☹

- 実データを保持するのでメモリ消費大
- データ追加は早くない（annoyと同程度。falconnより遅い）

参考資料

- Navigable Small World Graphの元論文：Y. Malkov et al., “Approximate Nearest Neighbor Algorithm based on Navigable Small World Graphs,” Information Systems 2013

- Navigable Small World Graphの階層版：Y. Malkov and D. Yashunin, “Efficient and Robust Approximate Nearest Neighbor search using Hierarchical Navigable Small World Graphs,” IEEE TPAMI 2019

- nmslibの公式python binding [link]

N

10^9

billion-scale

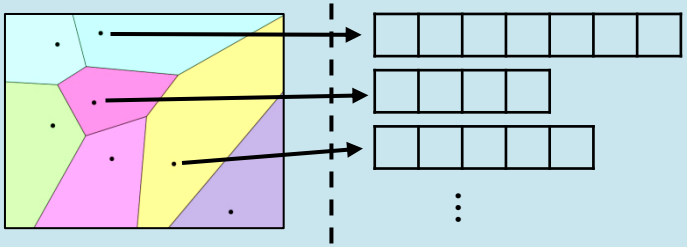
10^6

million-scale

転置インデクス+データ圧縮

ざっくり空間分割
- k-means
- 複数k-means
- PQ/OPQ
- etc…

データ圧縮
- 生データのまま
- Scalar quantization
- PQ/OPQ
- etc…



圧縮データを直接探索

PQTable

Multi hash table

生データを直接扱う：
精度○，メモリ効率×

Locality Sensitive Hashing (LSH)系

Tree系 / Space Partitioning系

Graph探索系

データを圧縮する：
精度△，メモリ効率○

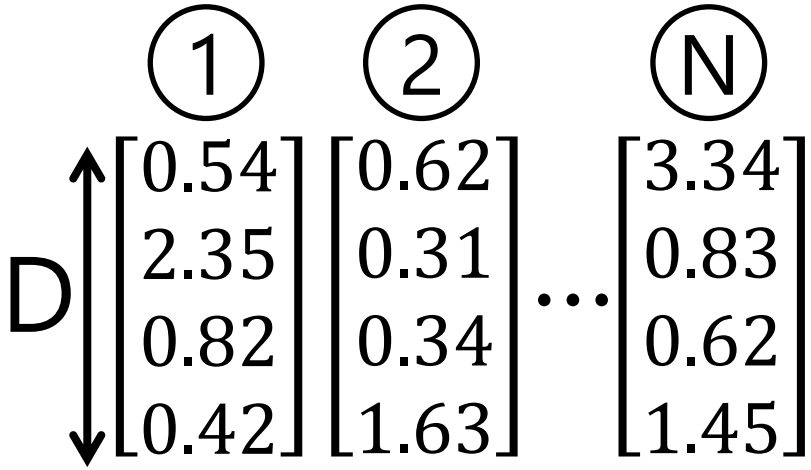
ルックアップ系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → ID: 2 / ID: 123 → ADC 線形探索

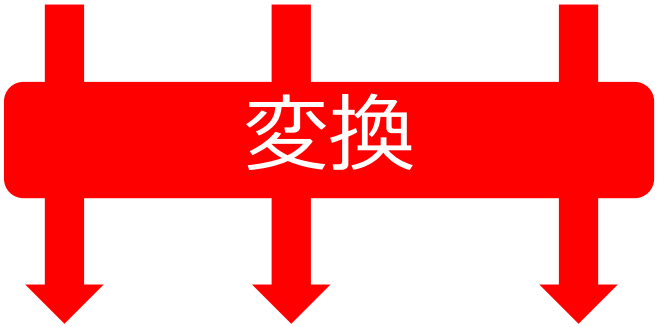
ハミング系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → 0 1 0 0 → ハミング 線形探索

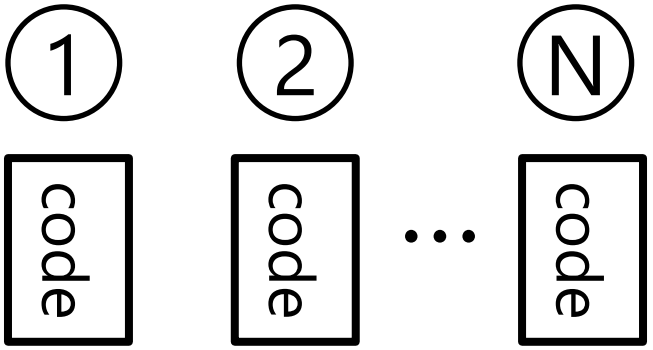
基本的な考え方



- N個の実数値ベクトルを表現するにはfloatを用いて$4ND$ byte 必要
- NやDが大きいとメモリに載らない
- 例：$D = 128, N = 10^9$の時，512 GB

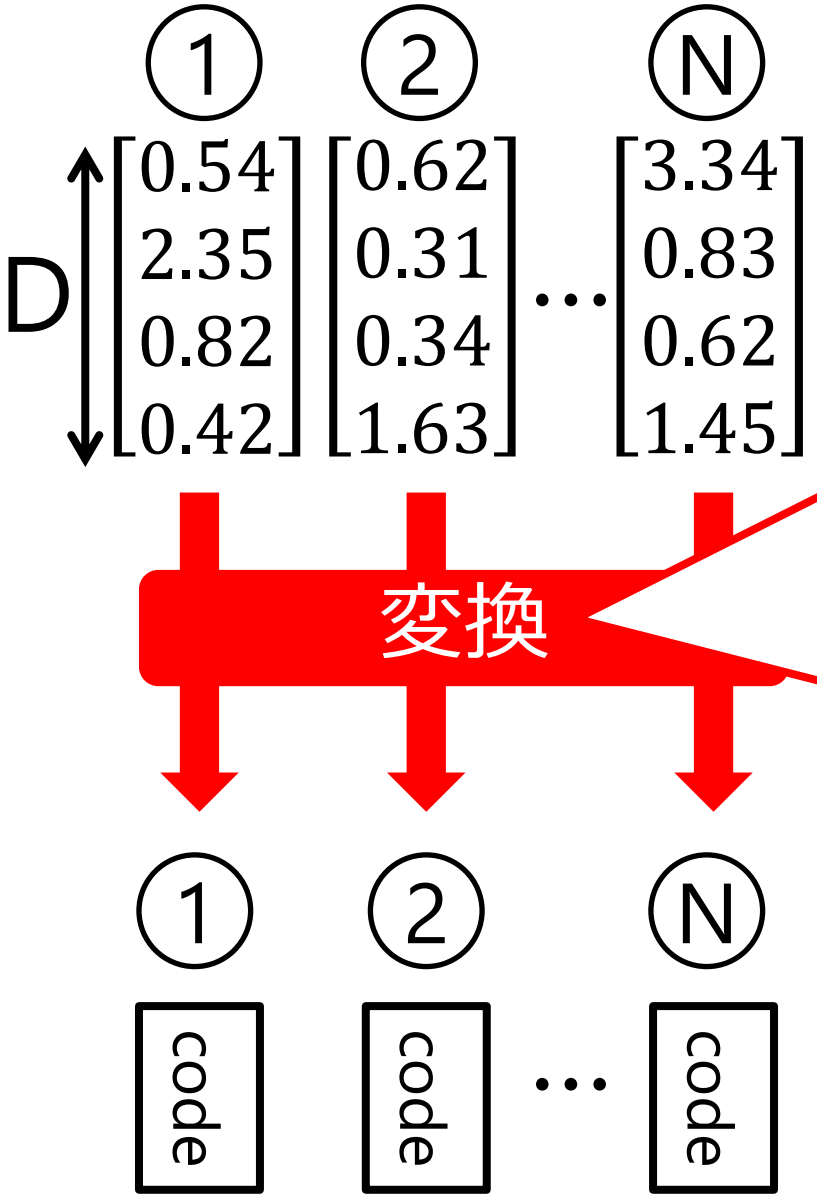


- 各ベクトル「変換」し「ショートコード」に圧縮する



- ショートコードはメモリ効率が良い
- 例：上のデータを32bitコードに圧縮すると，わずか4GB
- ショートコードの世界で探索を考える

基本的な考え方



どのような変換がいいか？

1. コード間の「距離」が計算できる．その距離は元のベクトル間の距離を近似する
2. コード間距離は高速に計算できる
3. 上の二つを，十分に小さいコードで実現できる

- メモリ効率が良い
- 例：上のデータを32bitコードに圧縮すると，わずか4GB
- ショートコードの世界で探索を考える

N

10^9

billion-scale

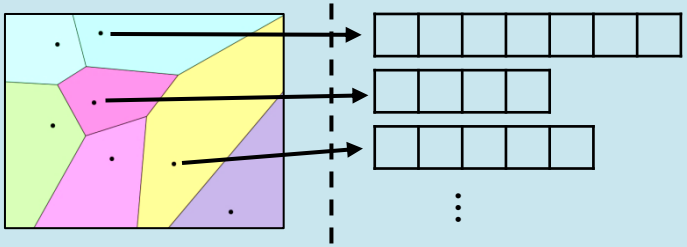
10^6

million-scale

転置インデクス+データ圧縮

ざっくり空間分割
- k-means
- 複数k-means
- PQ/OPQ
- etc…

データ圧縮
- 生データのまま
- Scalar quantization
- PQ/OPQ
- etc…



圧縮データを直接探索

PQTable

Multi hash table

生データを直接扱う：
精度○，メモリ効率×

Locality Sensitive Hashing (LSH)系

Tree系 / Space Partitioning系

Graph探索系

データを圧縮する：
精度△，メモリ効率○

ルックアップ系

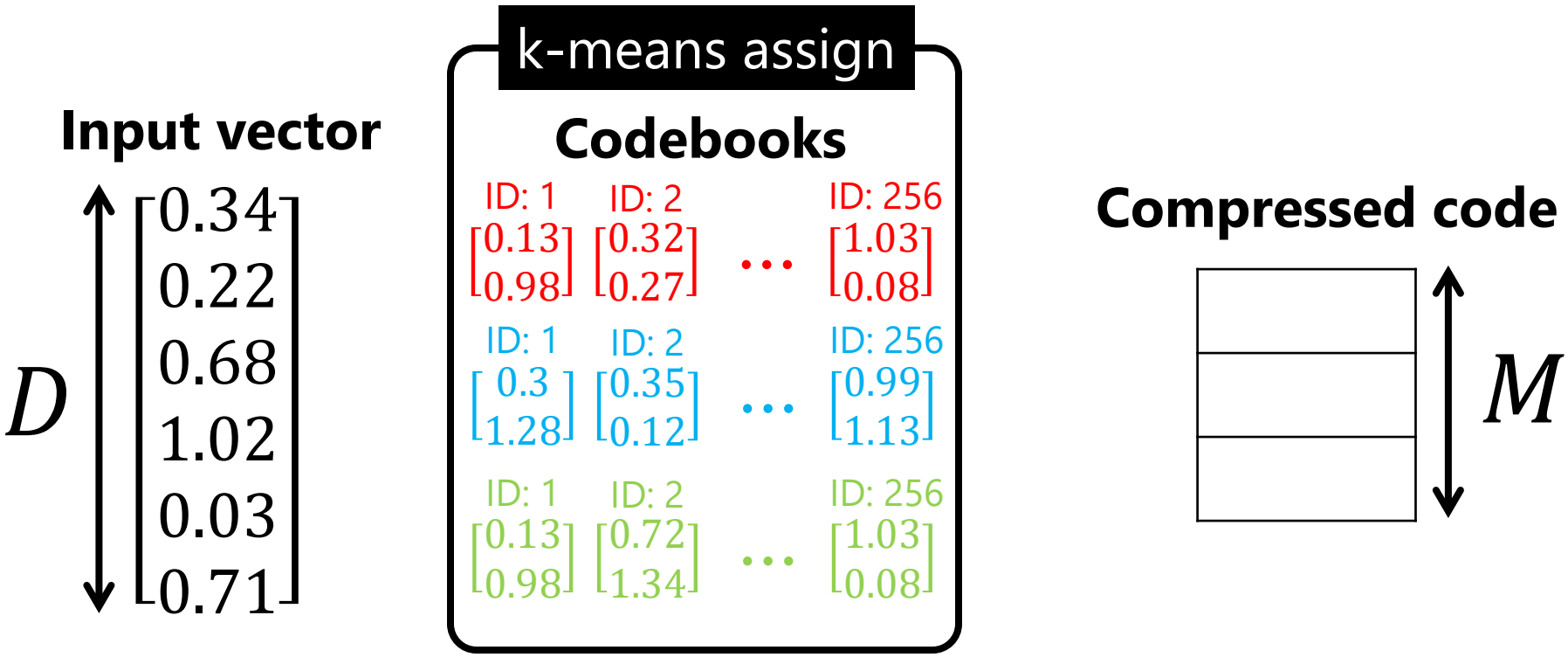
$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → ID: 2 / ID: 123 → ADC 線形探索

ハミング系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → 0 1 0 0 → ハミング 線形探索

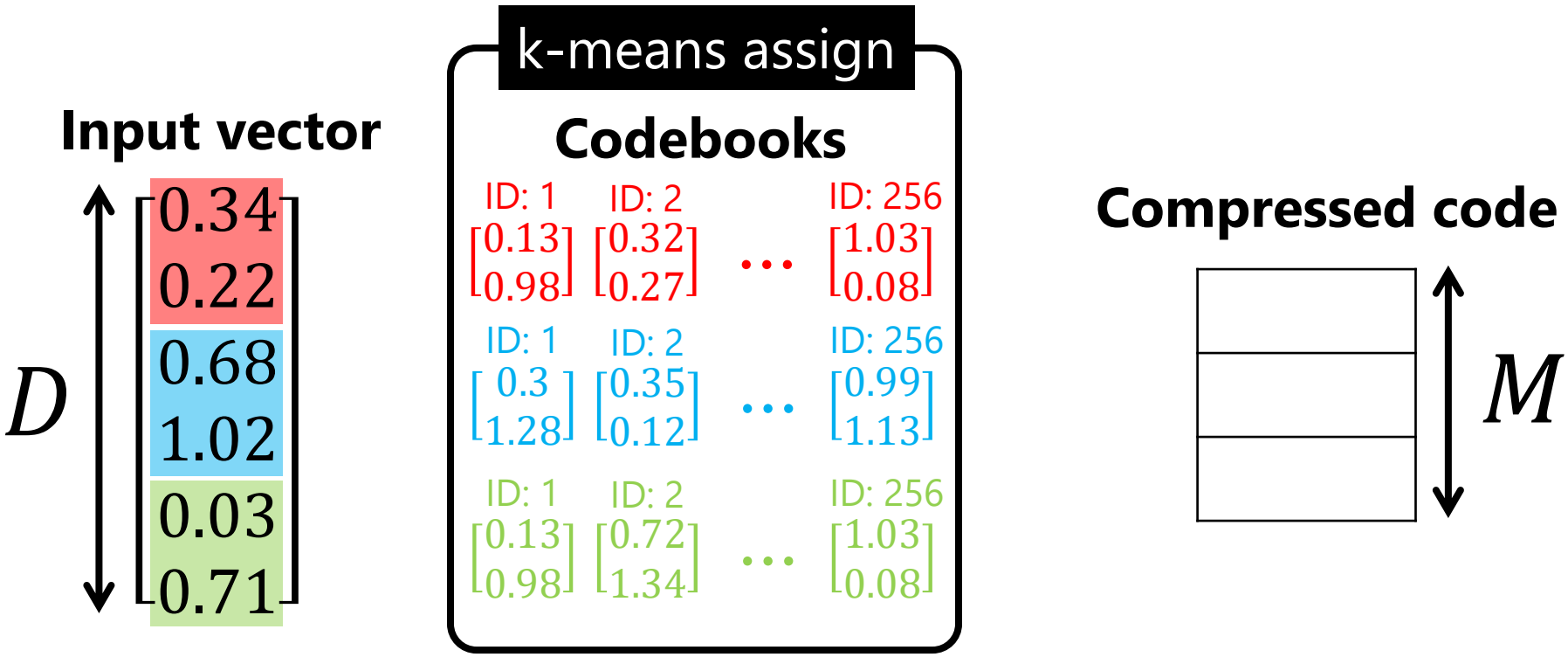
Product Quantization [Jégou, TPAMI 2011]

➢ ベクトルを分割してk-meansする



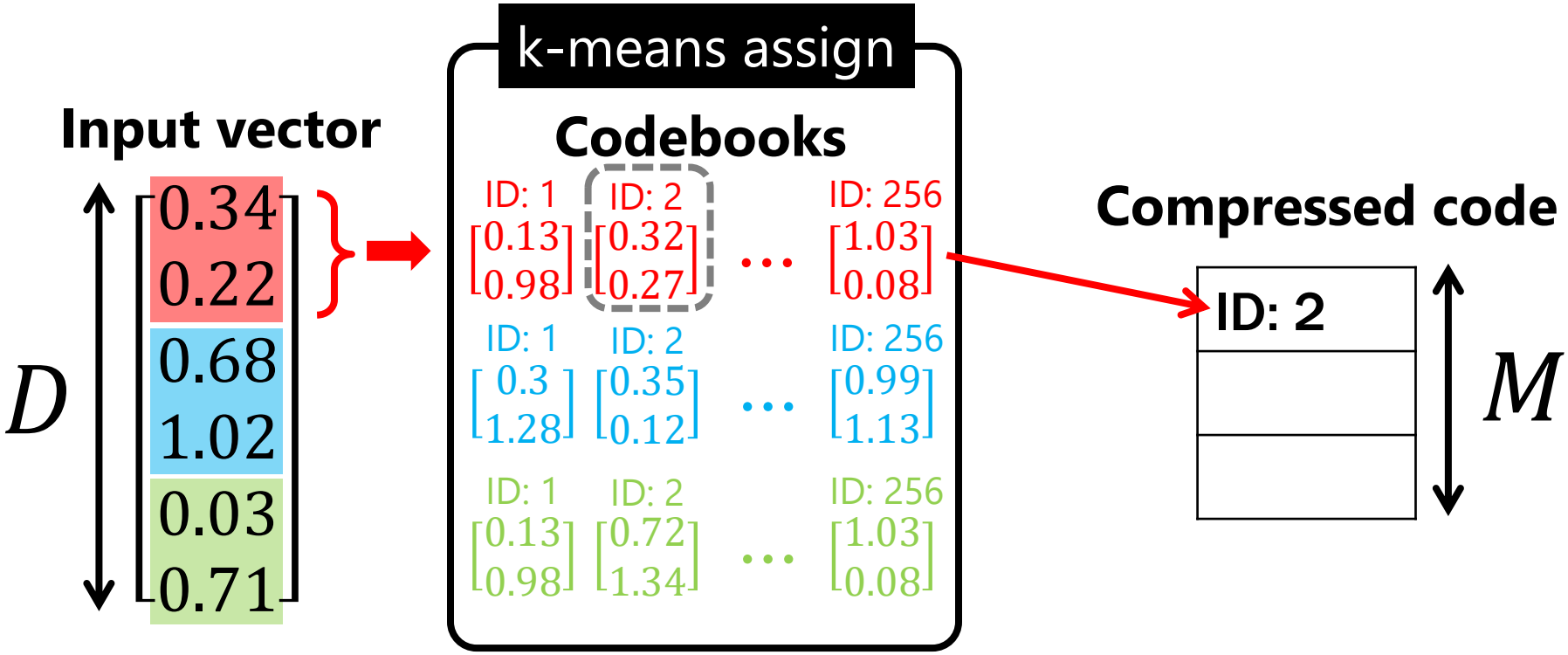
Product Quantization [Jégou, TPAMI 2011]

➢ ベクトルを分割してk-meansする



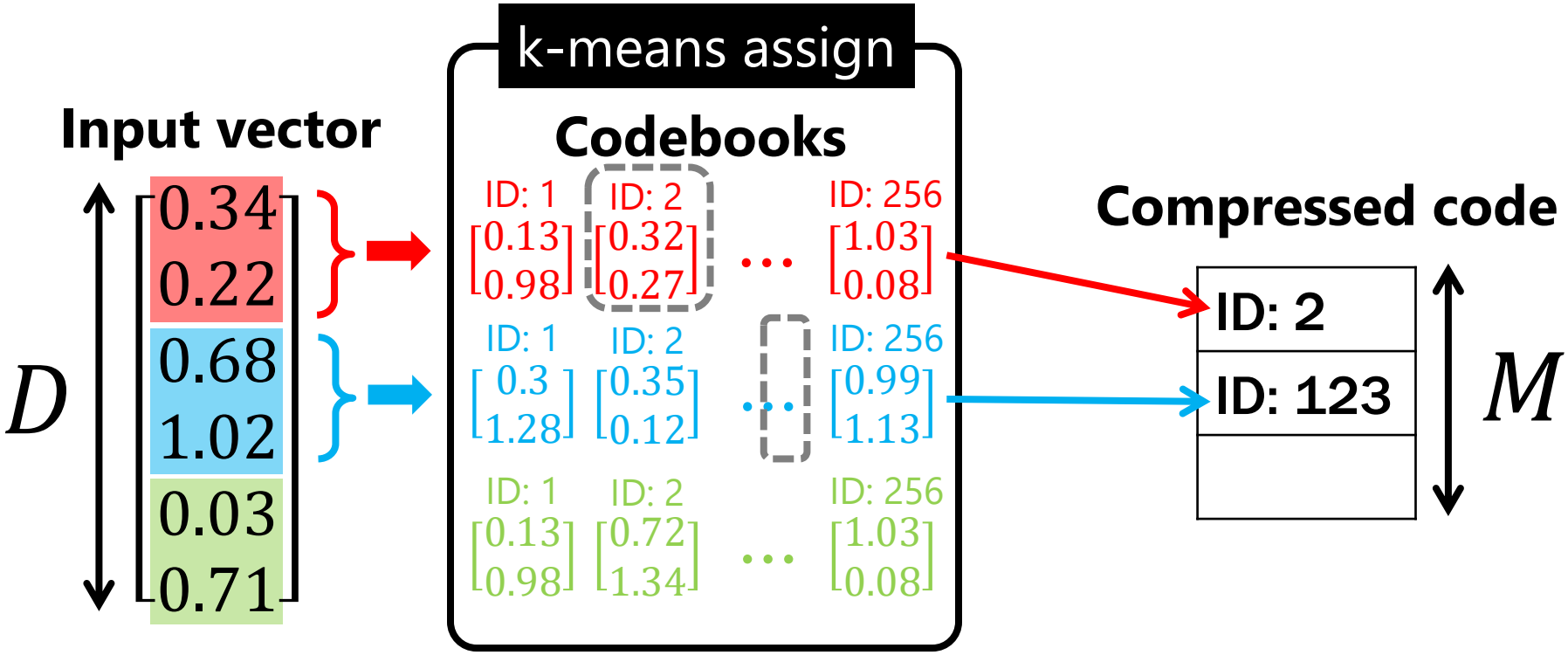
Product Quantization [Jégou, TPAMI 2011]

➢ ベクトルを分割してk-meansする



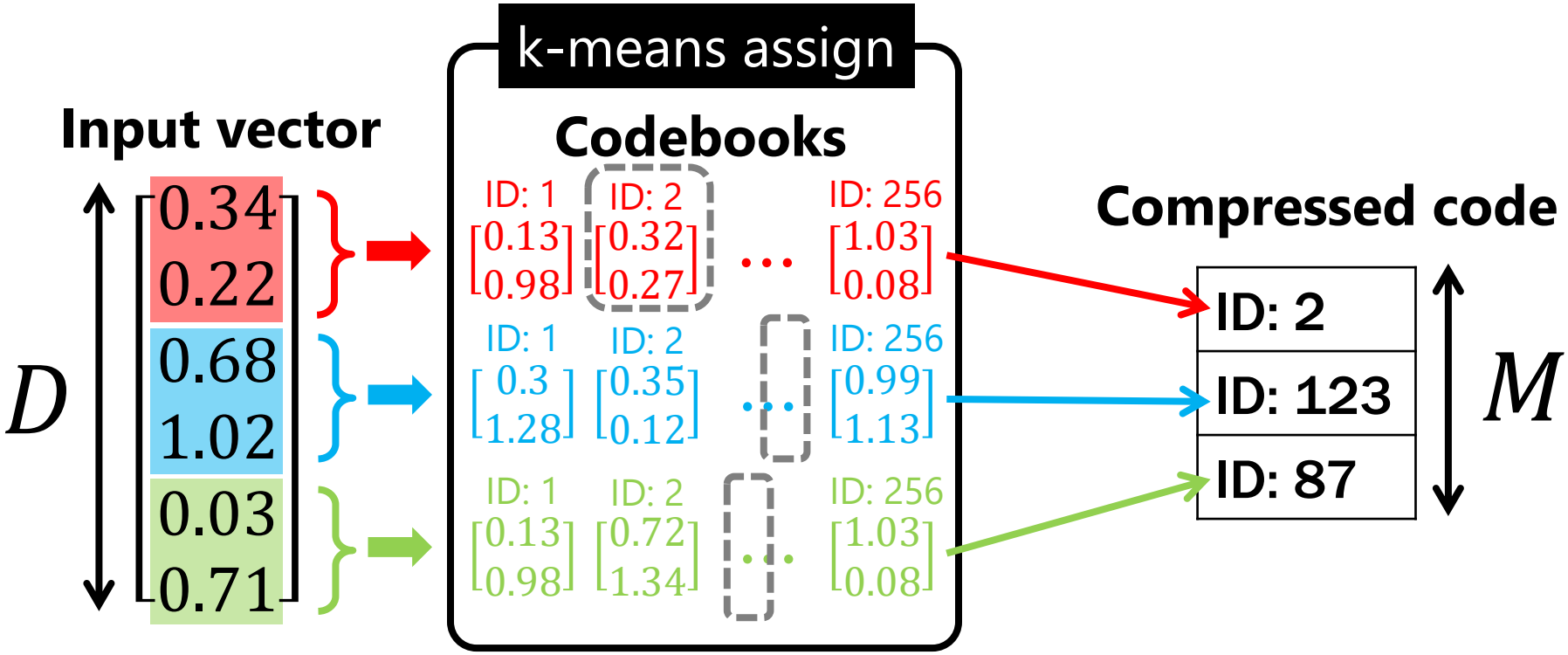
Product Quantization [Jégou, TPAMI 2011]

➢ ベクトルを分割してk-meansする



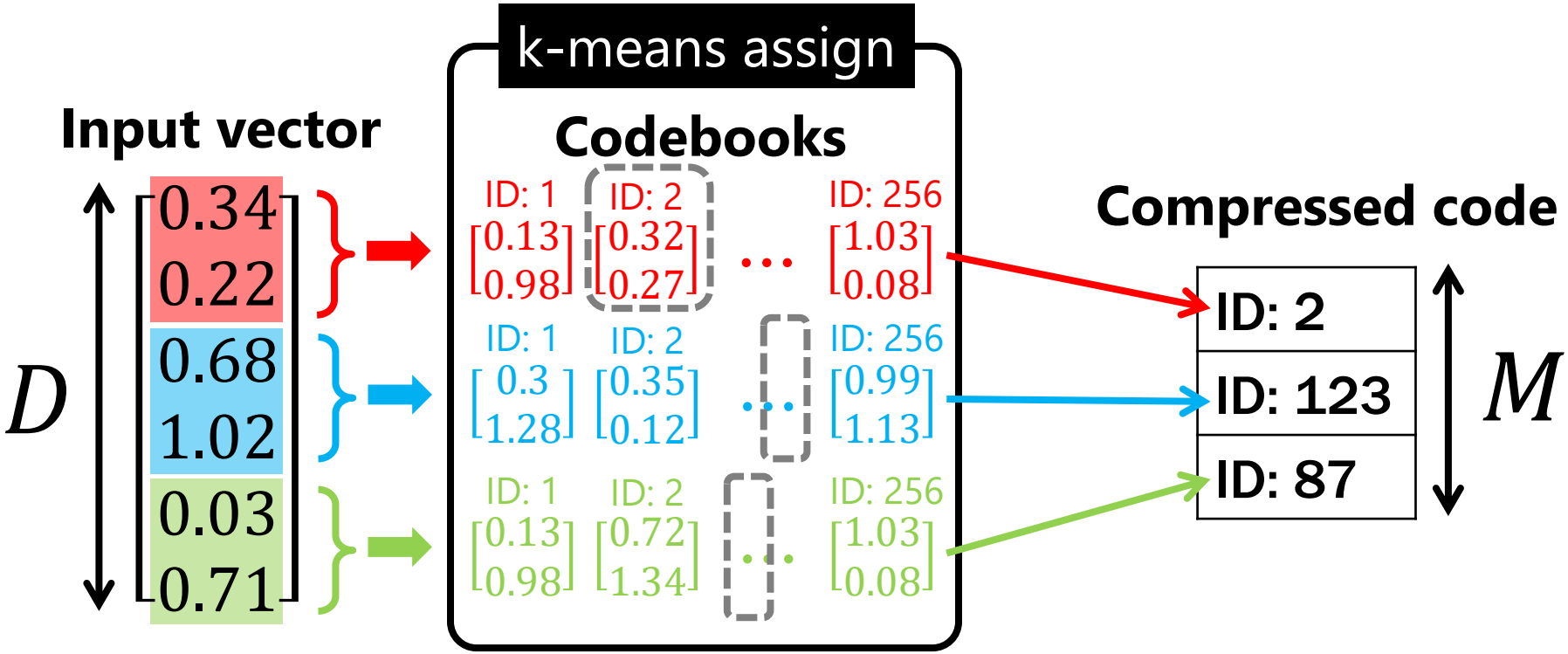
Product Quantization [Jégou, TPAMI 2011]

➢ ベクトルを分割してk-meansする



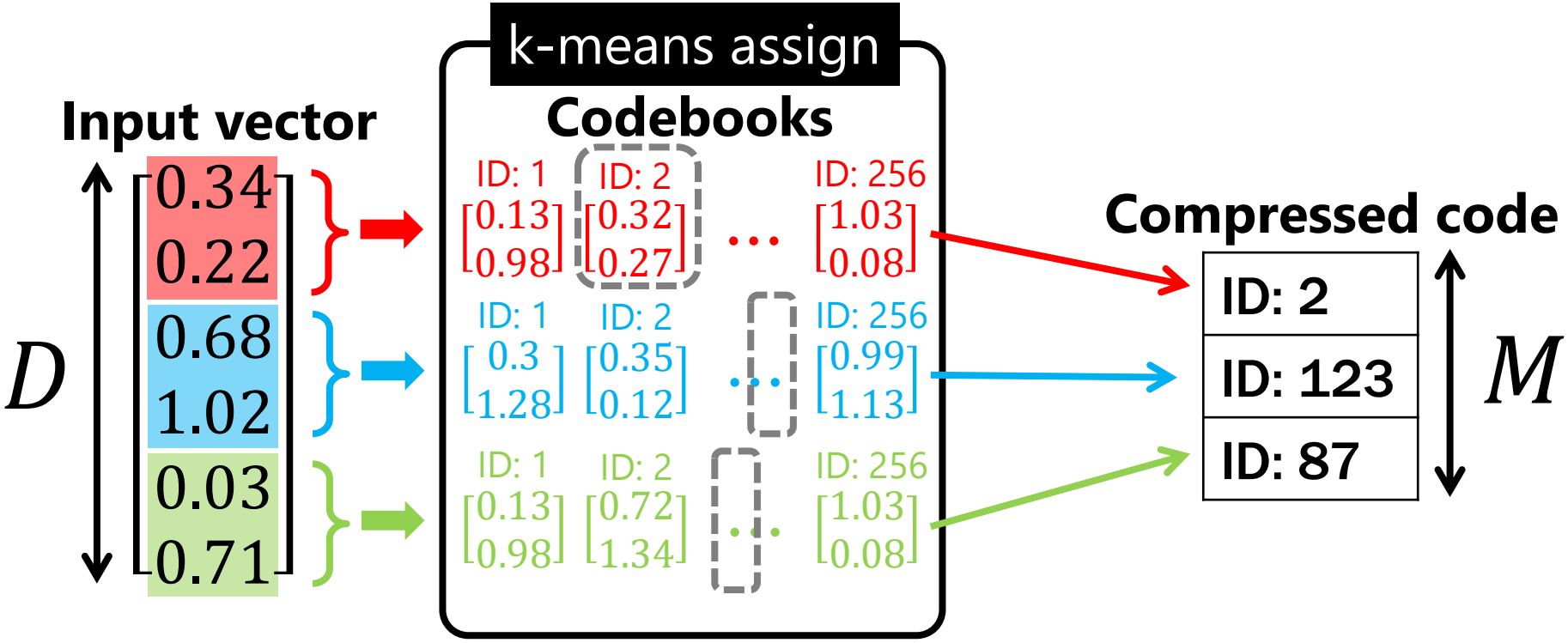
Product Quantization [Jégou, TPAMI 2011]

- ベクトルを分割してk-meansする

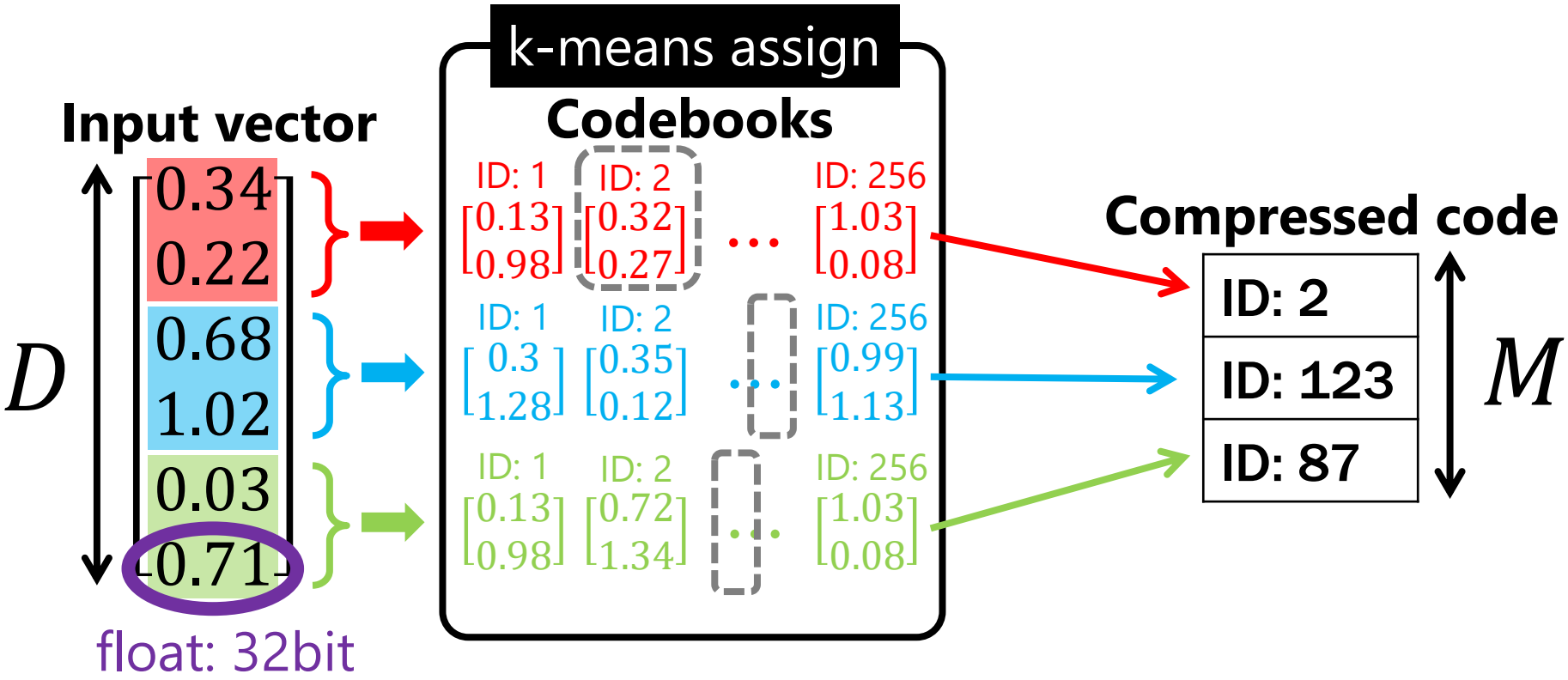


- 単純
- メモリ効率良い
- 距離 $d(input, code)^2$ を近似計算可能

Product Quantization: メモリ効率良



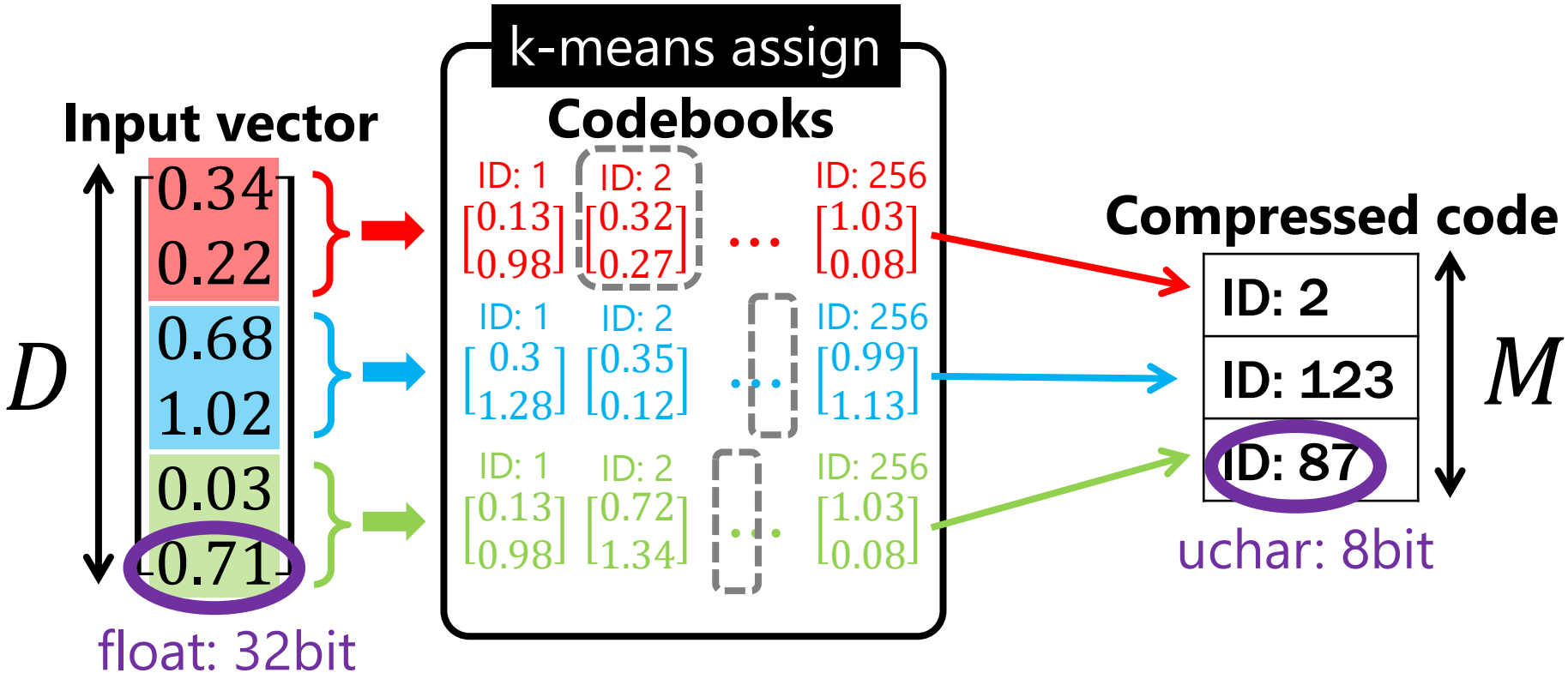
Product Quantization: メモリ効率良



e.g., $D = 128$

$128 \times 32 = 4096$ [bit]

Product Quantization: メモリ効率良



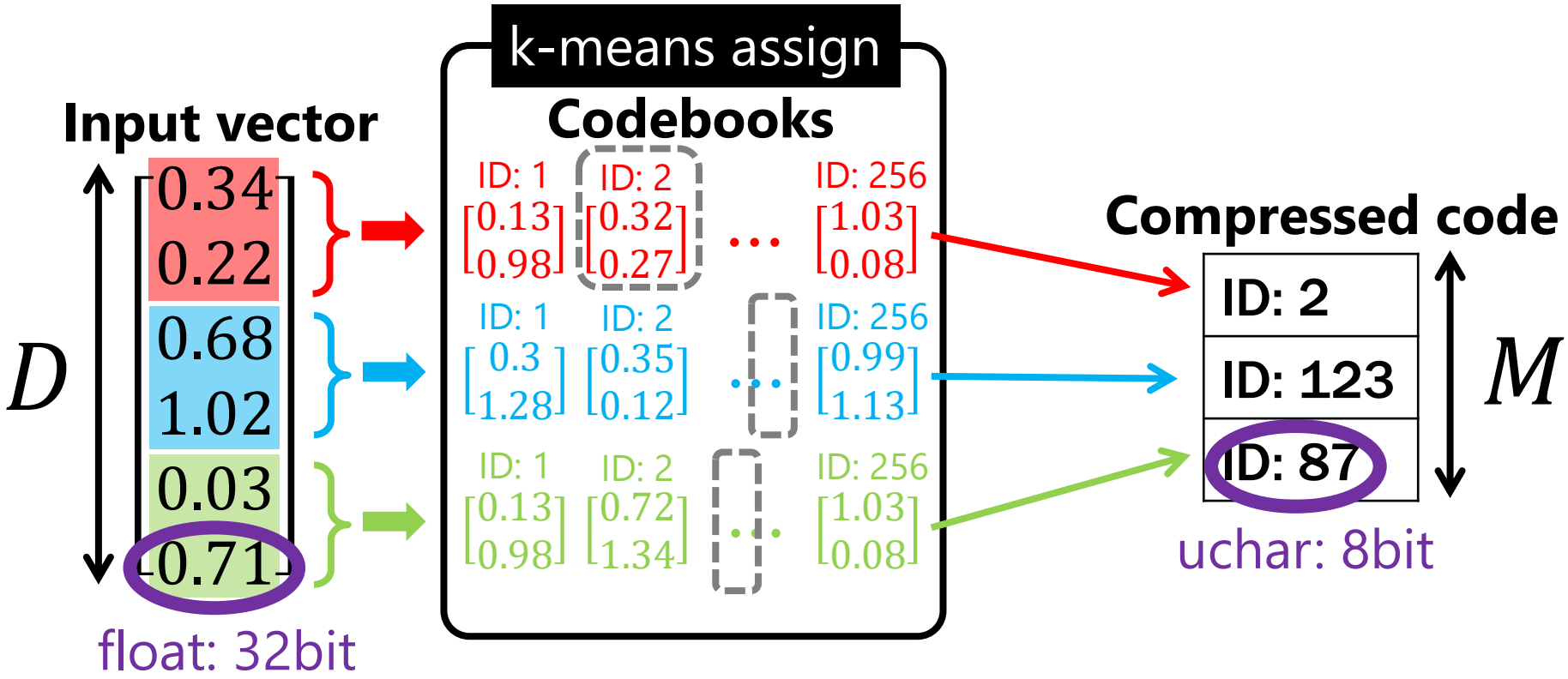
e.g., $D = 128$

$128 \times 32 = 4096$ [bit]

e.g., $M = 8$

$8 \times 8 = 64$ [bit]

Product Quantization: メモリ効率良



e.g., $D = 128$

$128 \times 32 = 4096$ [bit]

e.g., $M = 8$

$8 \times 8 = 64$ [bit]

1/64に圧縮

N

10^9

billion-scale

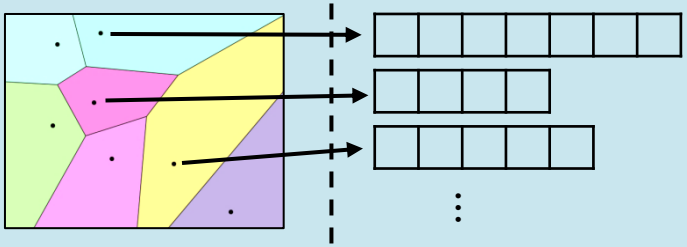
10^6

million-scale

転置インデクス+データ圧縮

ざっくり空間分割
- k-means
- 複数k-means
- PQ/OPQ
- etc…

データ圧縮
- 生データのまま
- Scalar quantization
- PQ/OPQ
- etc…



圧縮データを直接探索

PQTable

Multi hash table

生データを直接扱う：
精度○，メモリ効率×

Locality Sensitive Hashing (LSH)系

Tree系 / Space Partitioning系

Graph探索系

データを圧縮する：
精度△，メモリ効率○

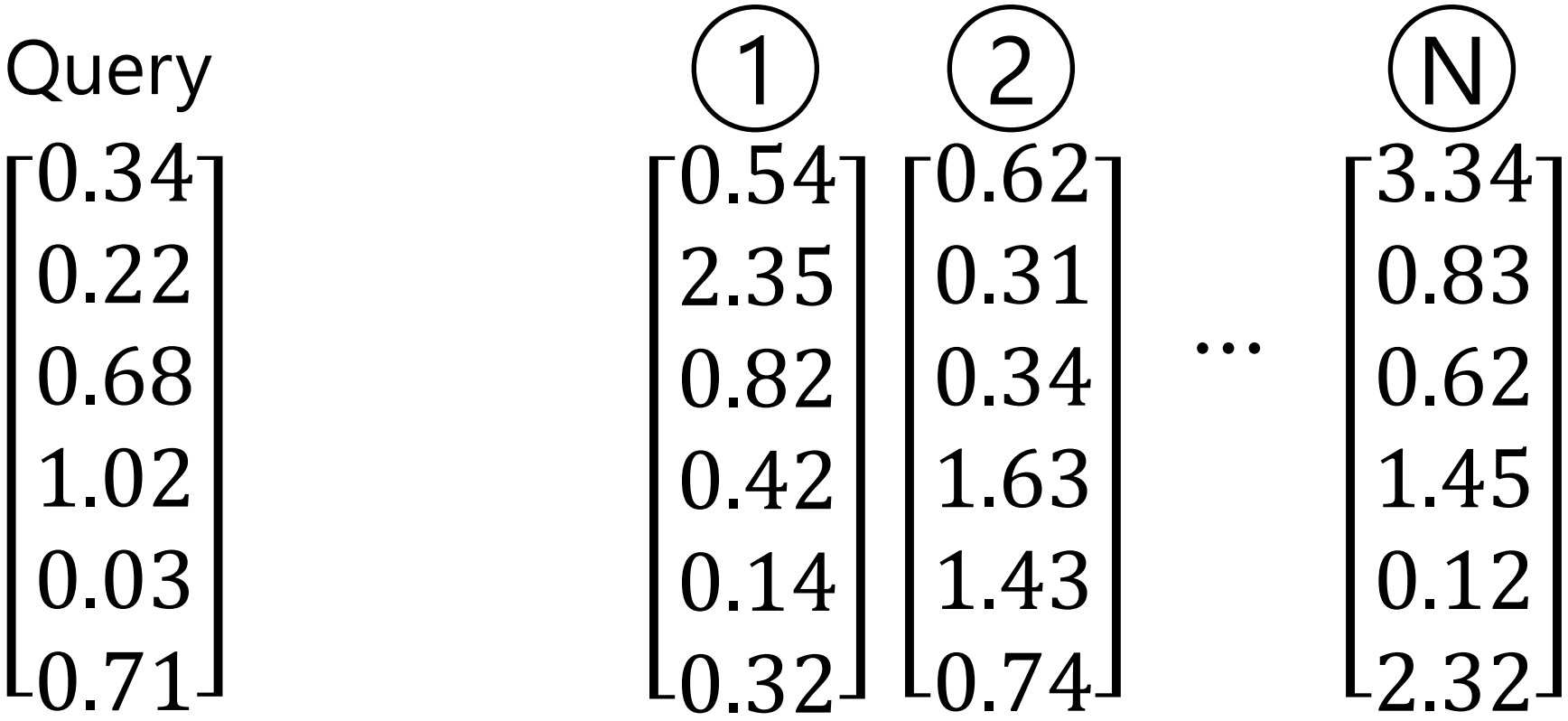
ルックアップ系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → ID: 2 / ID: 123 → ADC 線形探索

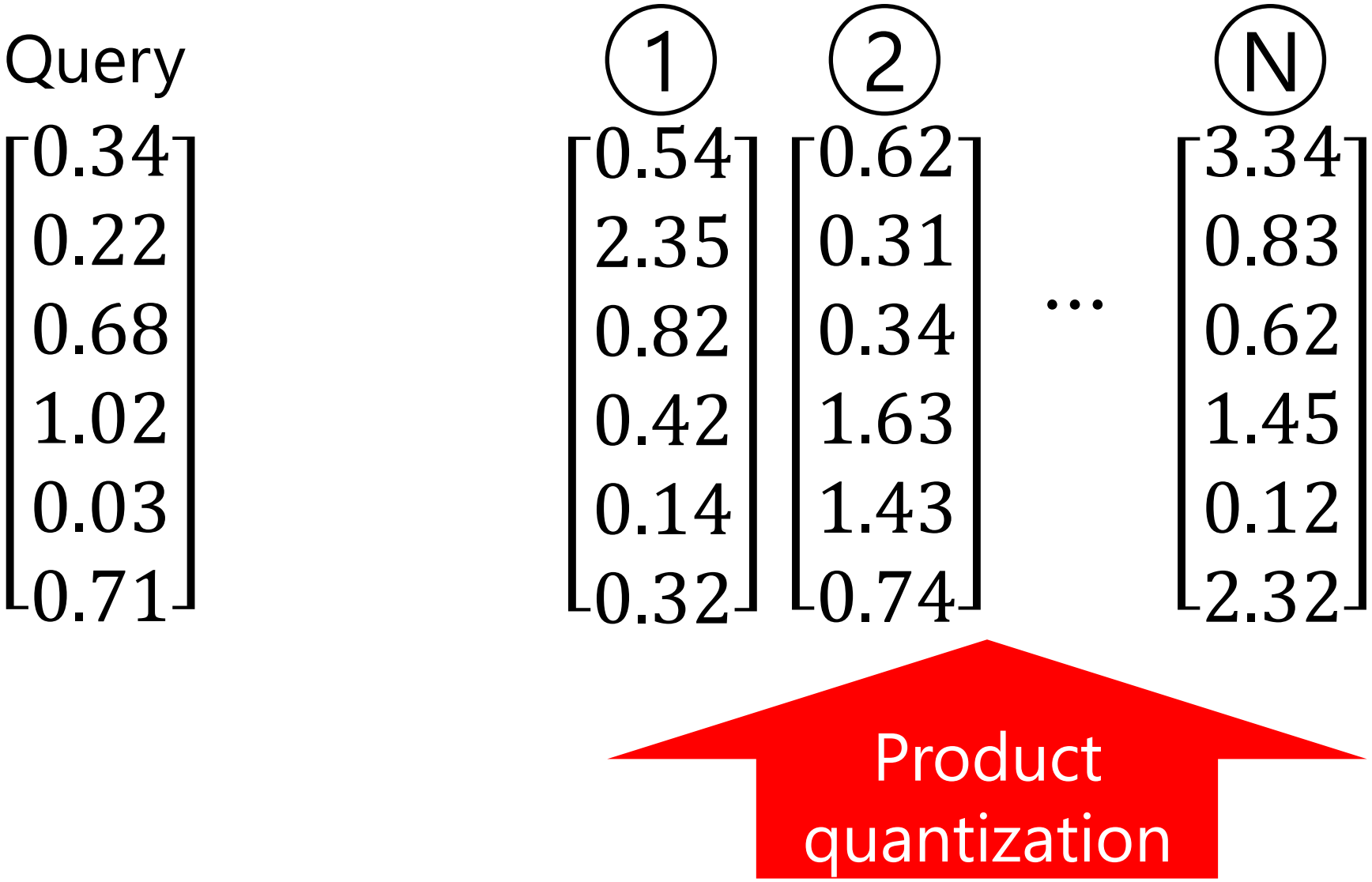
ハミング系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → 0 1 0 0 → ハミング 線形探索

Product Quantization: 距離近似



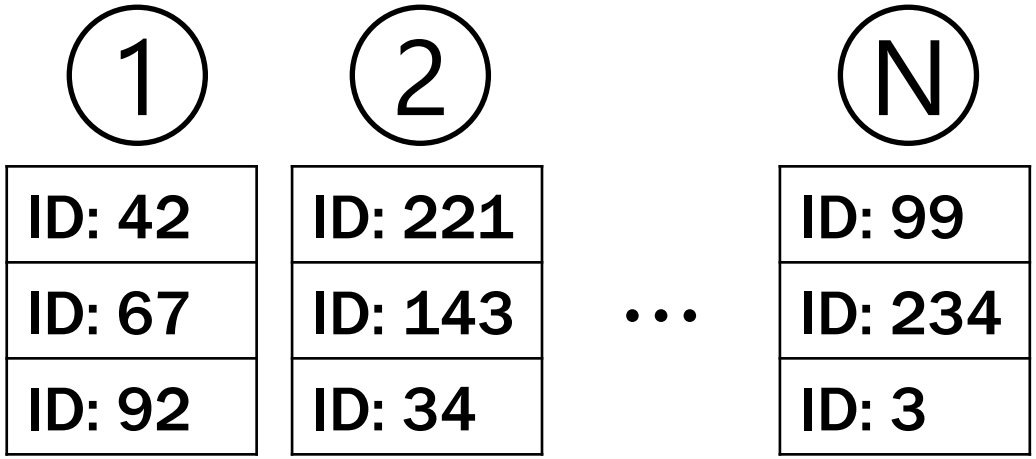
Product Quantization: 距離近似



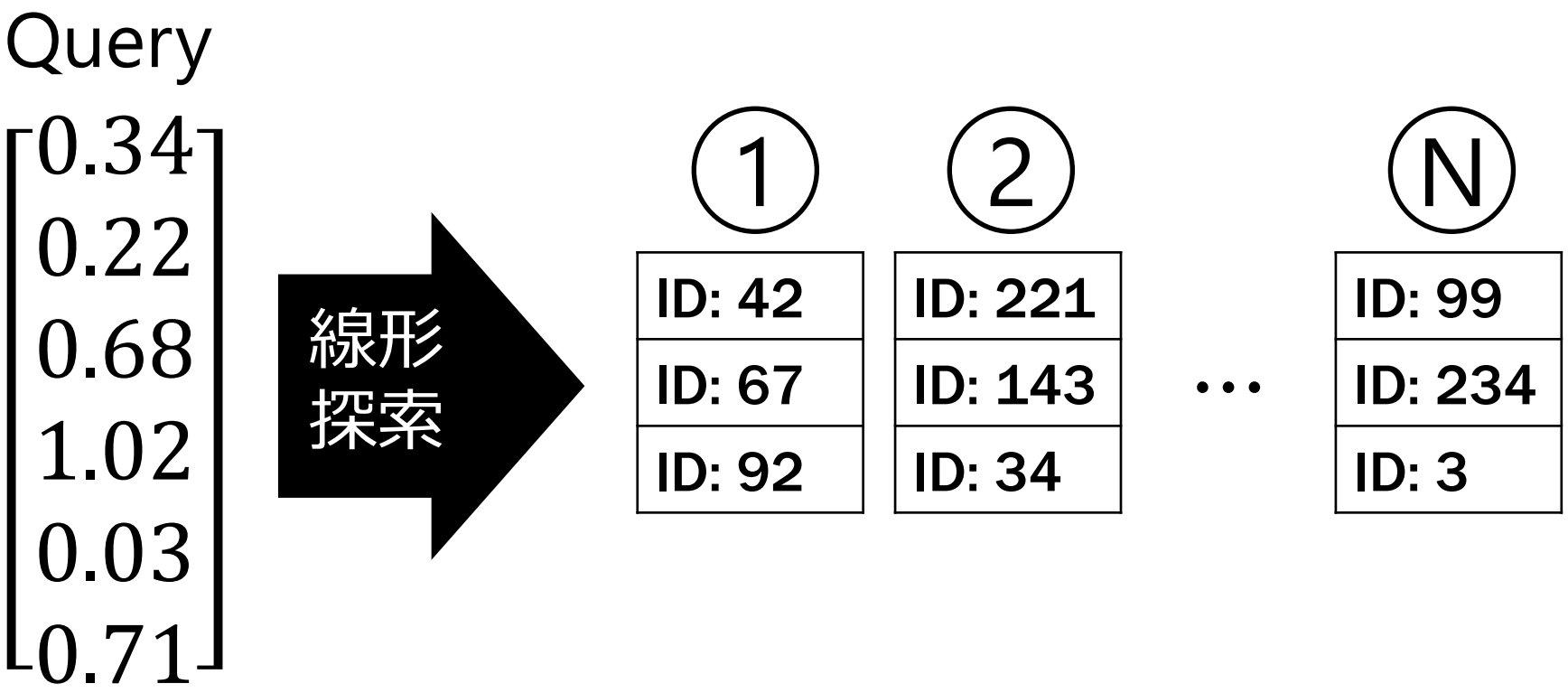
Product Quantization: 距離近似

Query

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



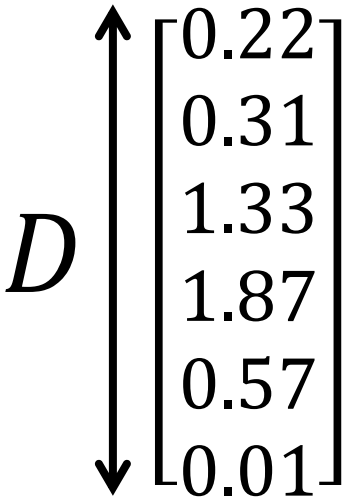
Product Quantization: 距離近似



（近似的距離で）線形探索が出来る

Product Quantization: 距離近似

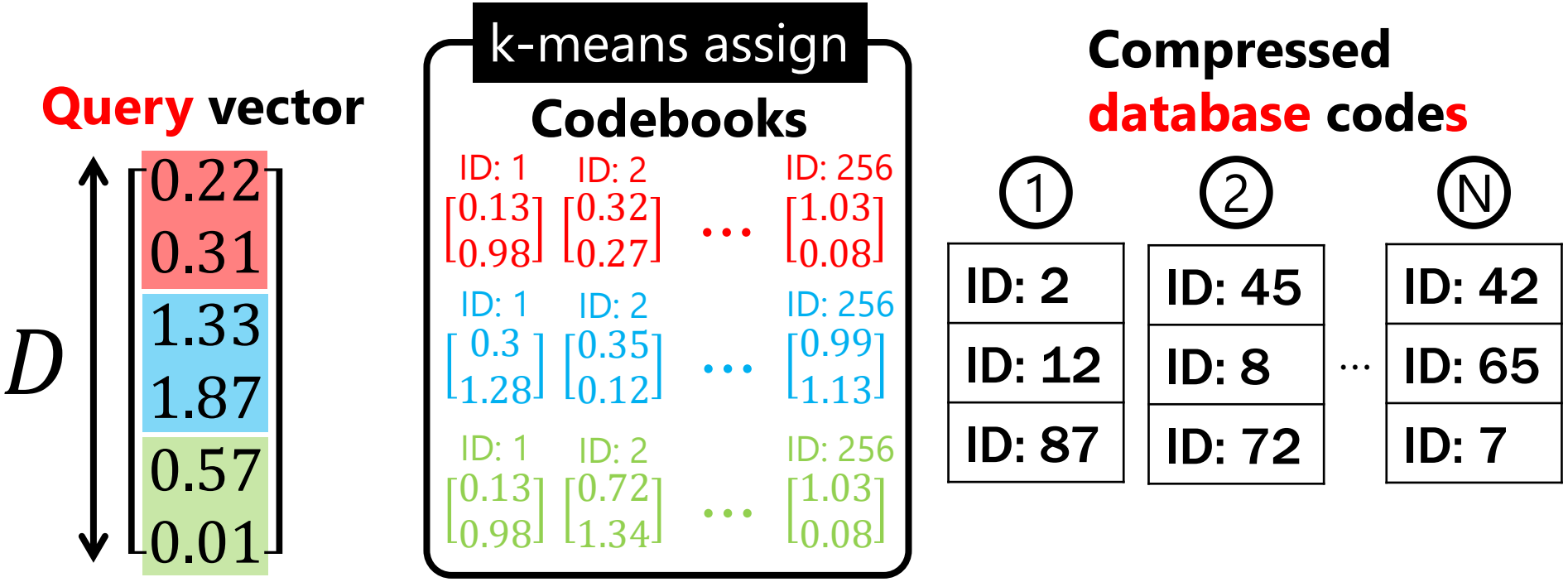
Query vector



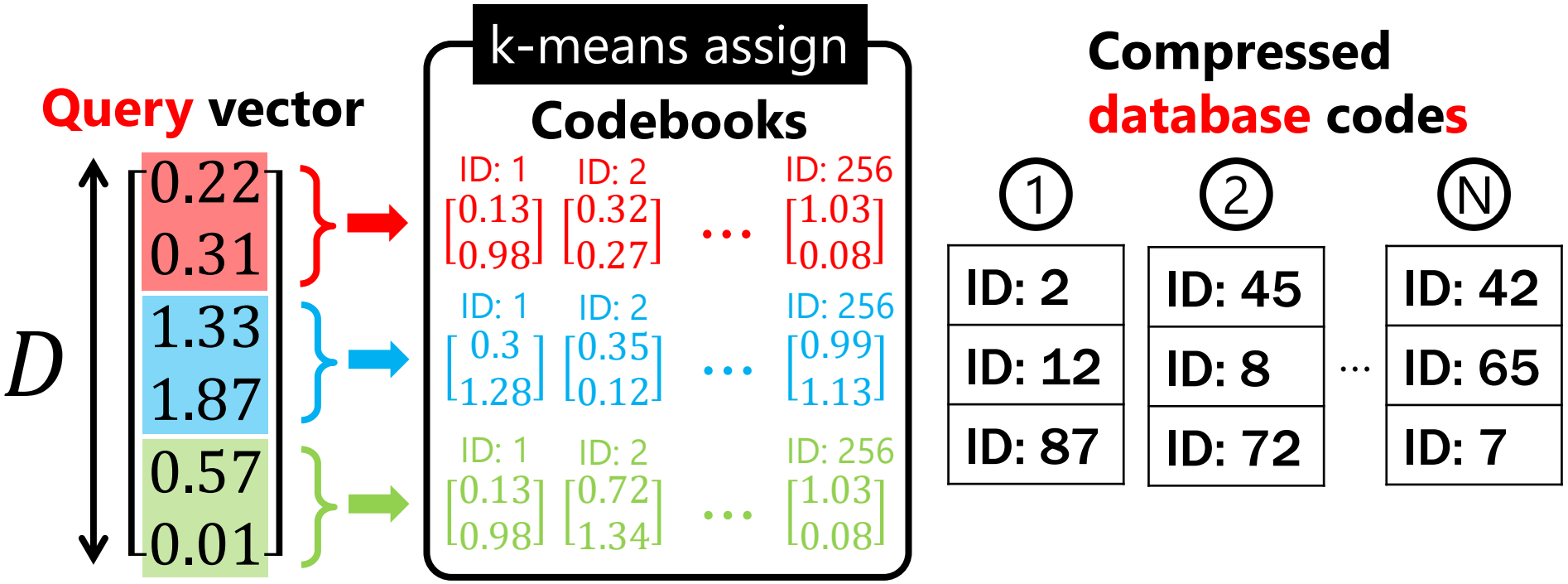
Compressed database codes

①	②		Ⓝ
ID: 2	ID: 45		ID: 42
ID: 12	ID: 8	…	ID: 65
ID: 87	ID: 72		ID: 7

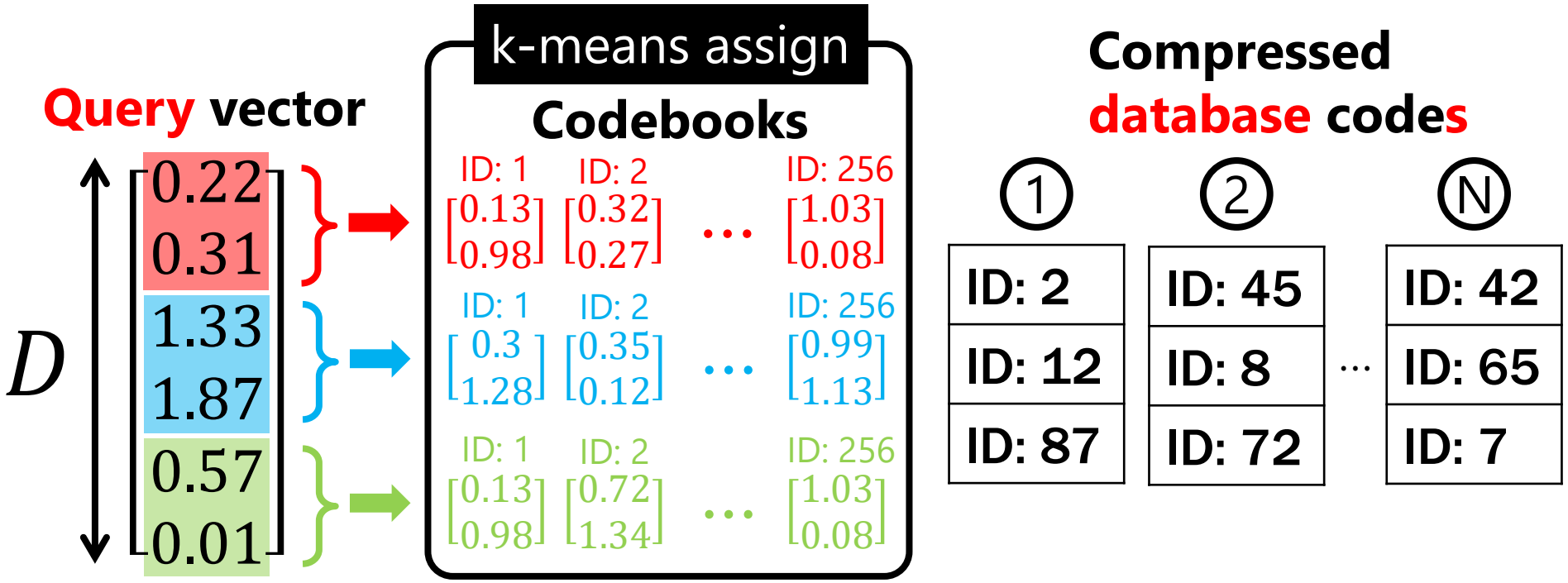
Product Quantization: 距離近似



Product Quantization: 距離近似



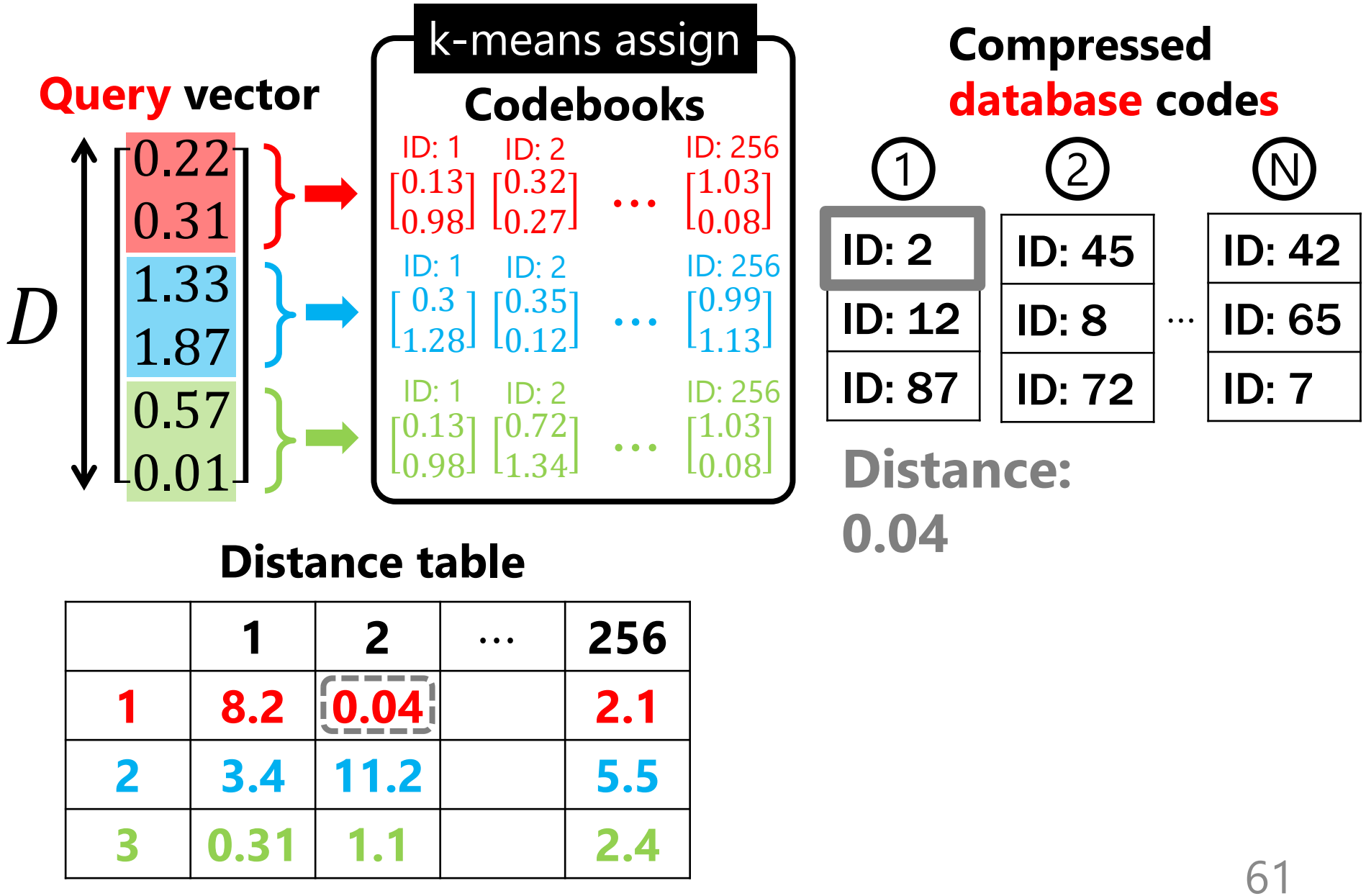
Product Quantization: 距離近似



Distance table

	1	2	…	256
1	8.2	0.04		2.1
2	3.4	11.2		5.5
3	0.31	1.1		2.4

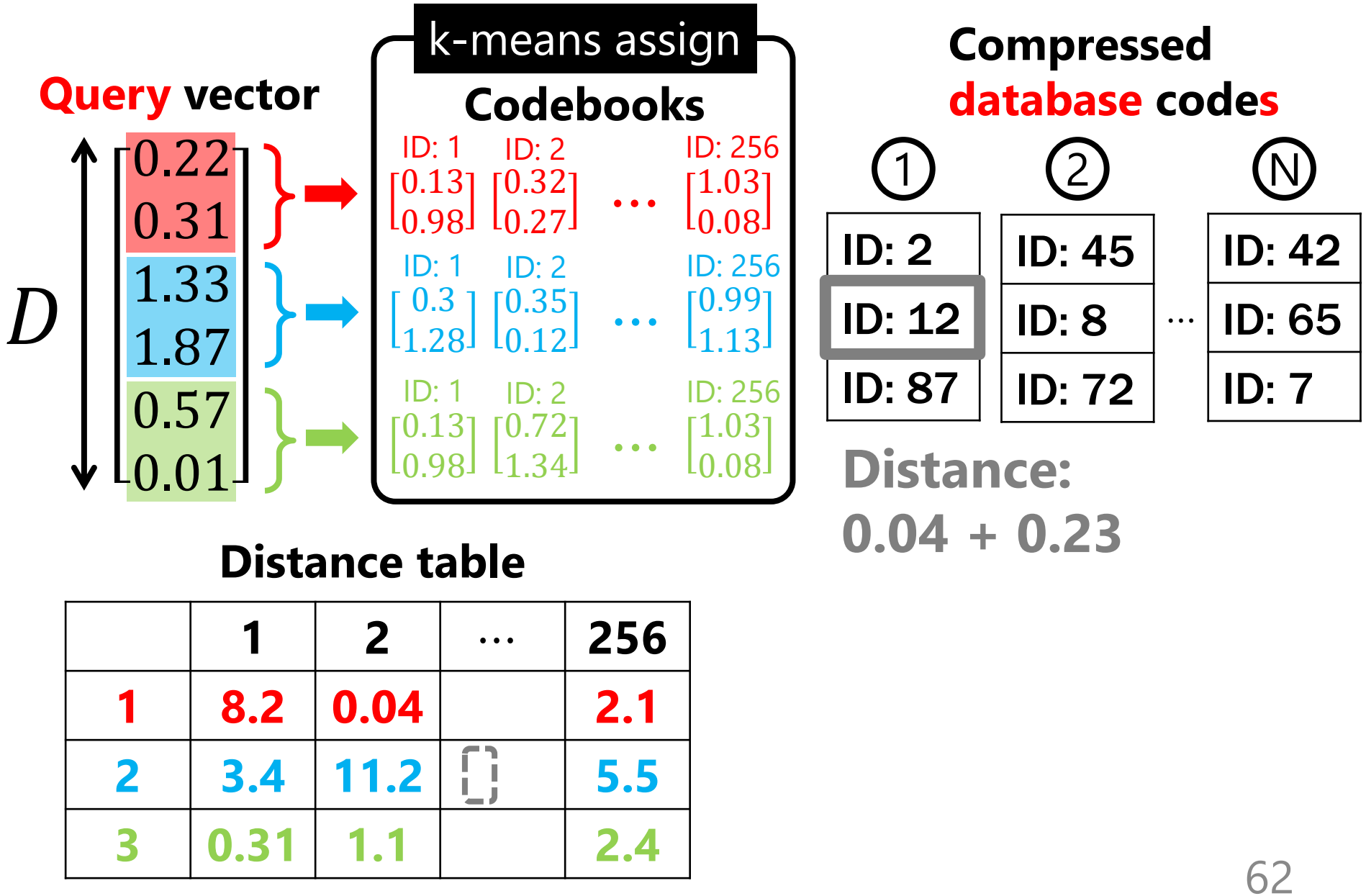
Product Quantization: 距離近似



Distance table

	1	2	…	256
1	8.2	0.04		2.1
2	3.4	11.2		5.5
3	0.31	1.1		2.4

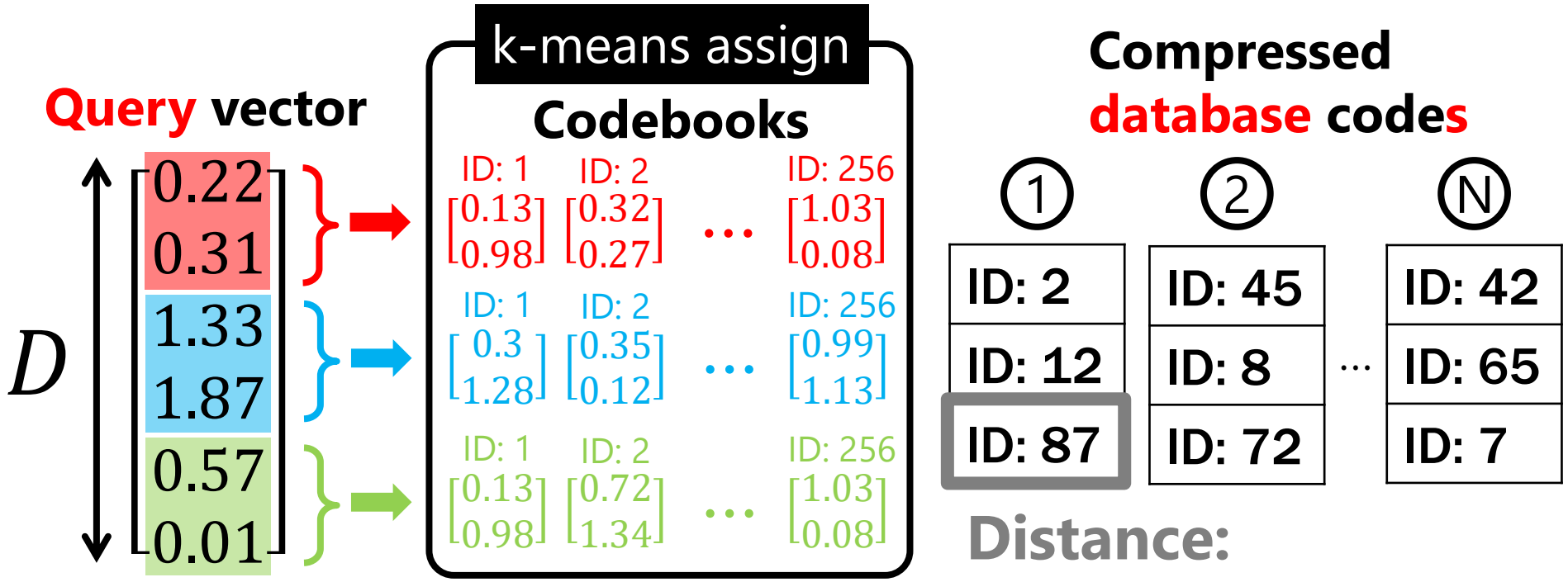
Product Quantization: 距離近似



Distance table

	1	2	…	256
1	8.2	0.04		2.1
2	3.4	11.2		5.5
3	0.31	1.1		2.4

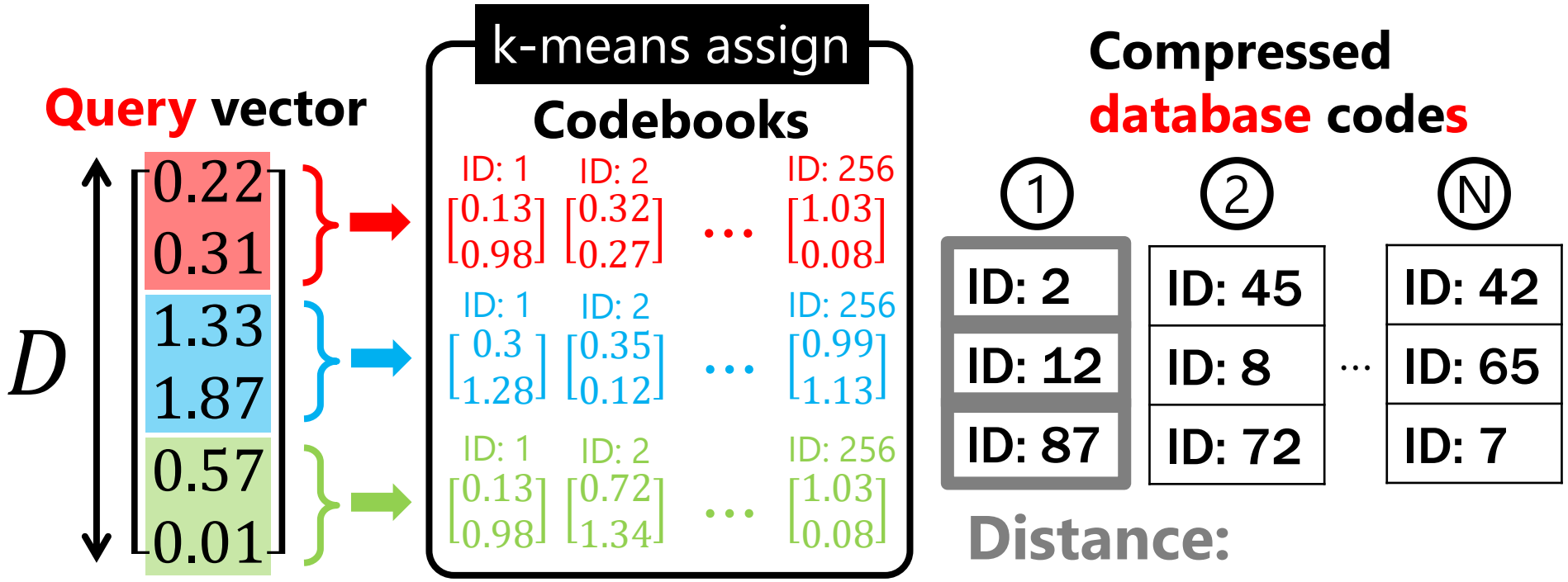
Product Quantization: 距離近似



Distance table

	1	2	…	256
1	8.2	0.04		2.1
2	3.4	11.2		5.5
3	0.31	1.1		2.4

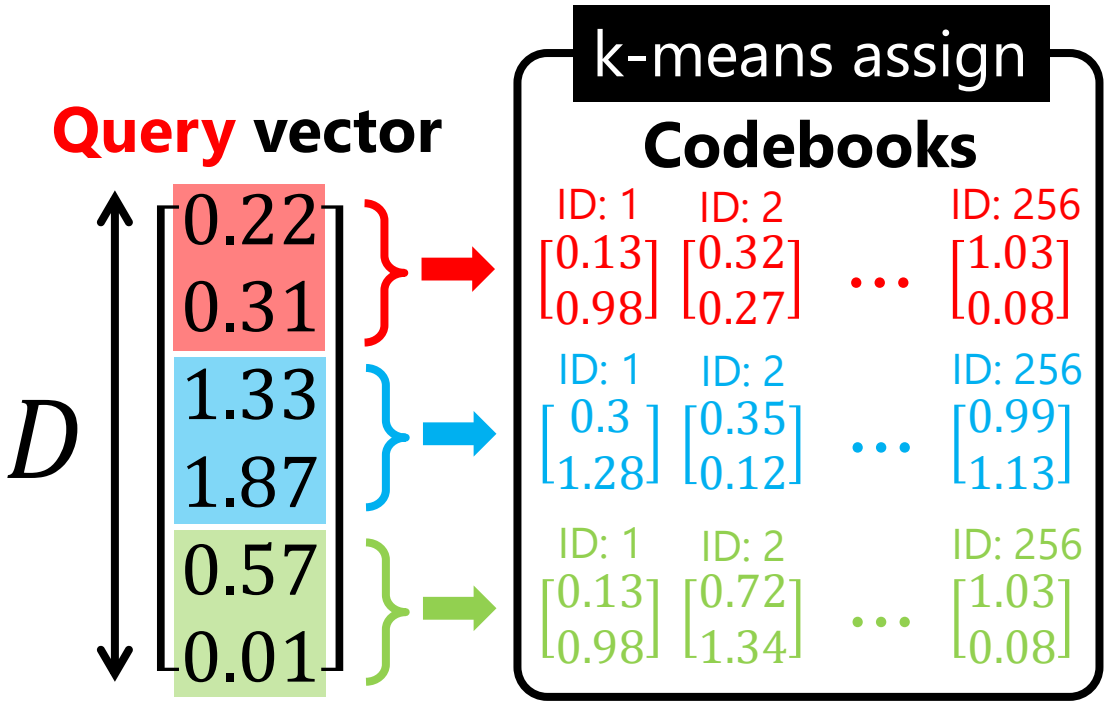
Product Quantization: 距離近似



Distance table

	1	2	…	256
1	8.2	0.04		2.1
2	3.4	11.2		5.5
3	0.31	1.1		2.4

Product Quantization: 距離近似



Compressed database codes

①	②	…	Ⓝ
ID: 2	ID: 45		ID: 42
ID: 12	ID: 8	…	ID: 65
ID: 87	ID: 72		ID: 7

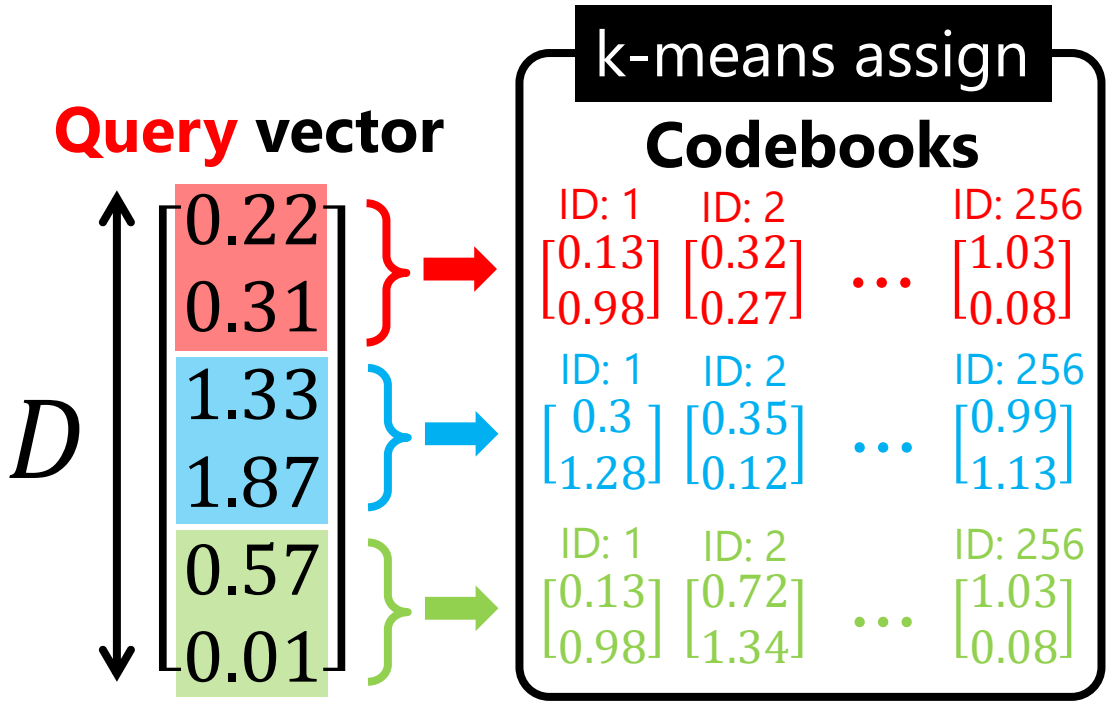
Distance:

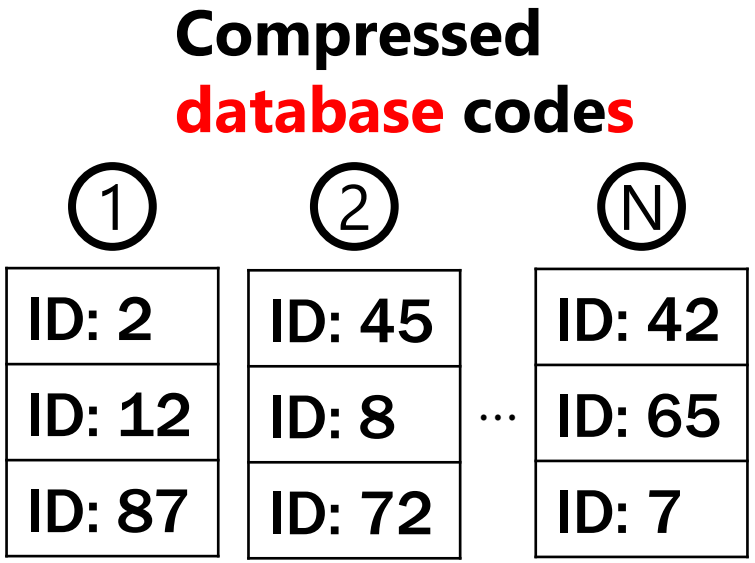
1.29 0.03 7.34

Distance table

	1	2	…	256
1	8.2	0.04		2.1
2	3.4	11.2		5.5
3	0.31	1.1		2.4

Product Quantization: 距離近似





Distance:

1.29 0.03 7.34

Distance table

	1	2	…	256
1	8.2	0.04		2.1
2	3.4	11.2		5.5
3	0.31	1.1		2.4

早い:

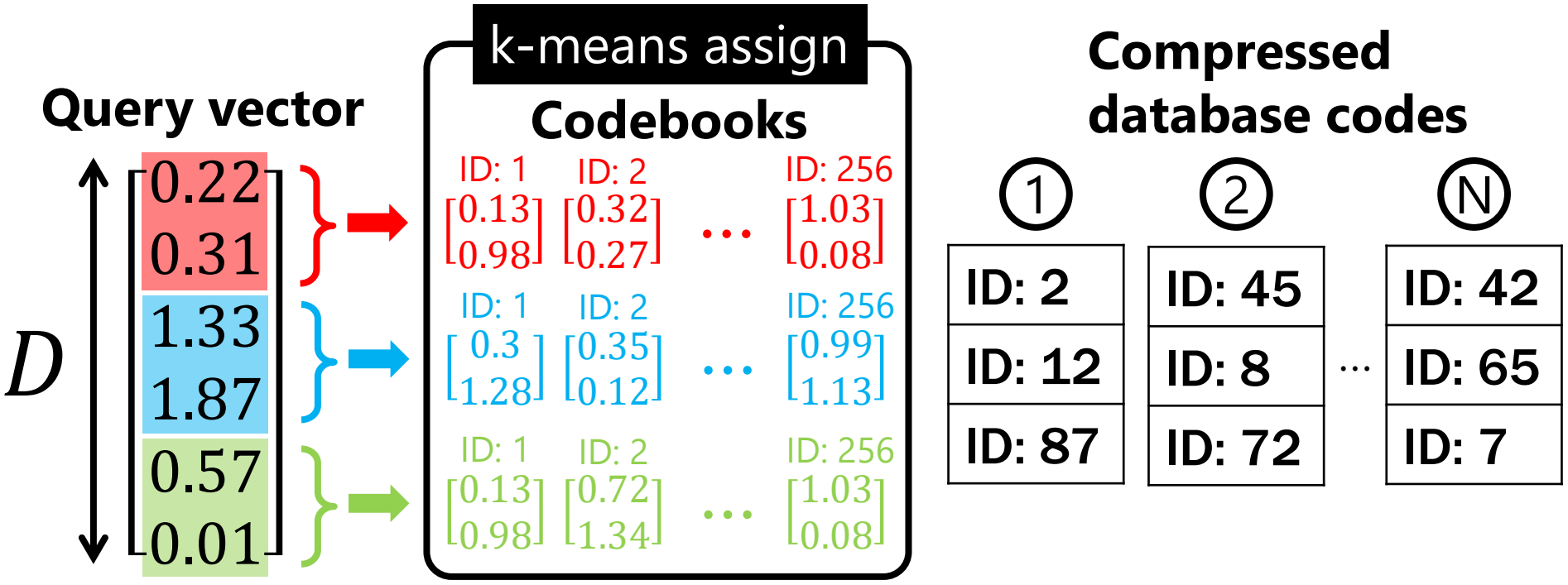
- Table lookup

計算量：
$O(DK + MN)$

正確に近似:

- 空間を 2^{8^M} に分割

Product Quantization



- 単純
- メモリ効率良い
- 距離 $d(input, code)^2$ を近似計算可能

Product Quantization

```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
```

```python
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)
```

- 非常に単純
- Pure python lib: [nanopq]
- `pip install nanopq`

N

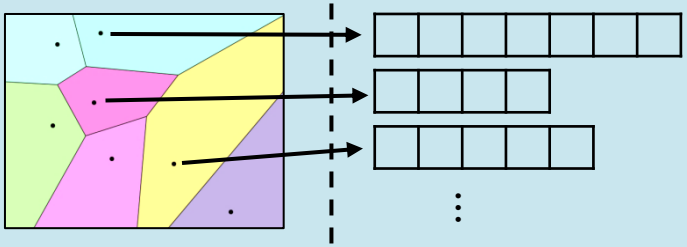
10^9

billion-scale

転置インデクス+データ圧縮

ざっくり空間分割
- k-means
- 複数k-means
- PQ/OPQ
- etc…

データ圧縮
- 生データのまま
- Scalar quantization
- PQ/OPQ
- etc…



圧縮データを直接探索

PQTable

Multi hash table

10^6

million-scale

生データを直接扱う：
精度○，メモリ効率×

Locality Sensitive Hashing (LSH)系

Tree系 / Space Partitioning系

Graph探索系

データを圧縮する：
精度△，メモリ効率○

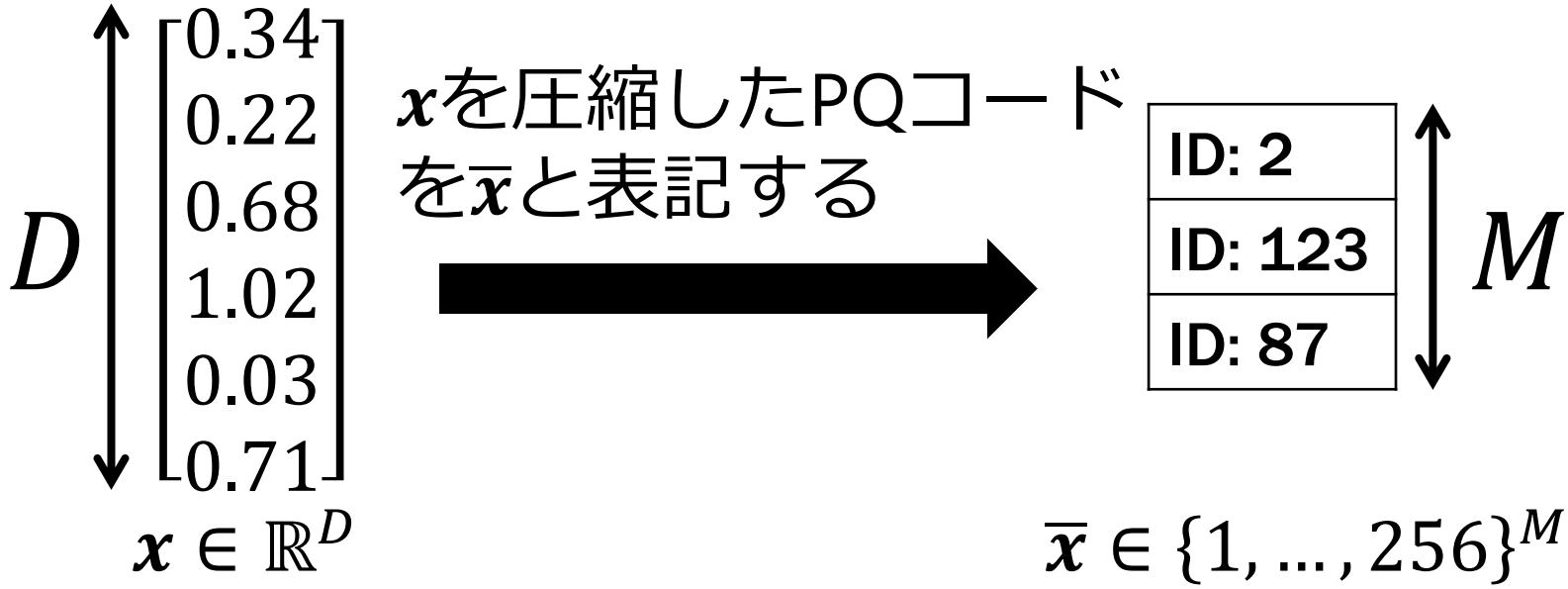
ルックアップ系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → ID: 2 / ID: 123 → ADC 線形探索

ハミング系

$\begin{bmatrix}0.34\\0.22\\0.68\\0.71\end{bmatrix}$ → 0 1 0 0 → ハミング 線形探索

PQを用いた探索システム：復習と表記



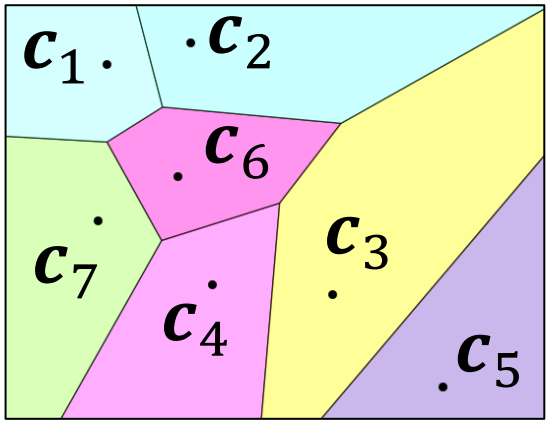
$\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^D$として，$\boldsymbol{x}$が圧縮されて$\bar{\boldsymbol{x}}$となっているとき，
二乗距離$d(\boldsymbol{y}, \boldsymbol{x})^2$はコード$\bar{\boldsymbol{x}}$を用いて高速近似計算出来る

$$d(\boldsymbol{y}, \boldsymbol{x})^2 \sim d_A(\boldsymbol{y}, \bar{\boldsymbol{x}})^2$$

ベクトルとベクトル
の二乗距離は・・

ベクトルとコードで
近似出来る

PQを用いた探索システム：データ登録



粗量子化

$k = 1$

$k = 2$

$\vdots$

$k = K$

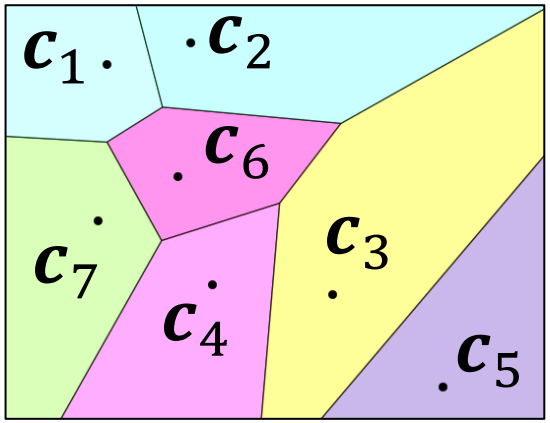
- 空間をK分割する「**粗量子化器**」を用意しておく．ここでは単純なボロノイ空間分割（k-means割り当てそのもの）
- 各$\{\boldsymbol{c}_k\}_{k=1}^{K}$は訓練データに単純にk-meansを適用し作る

PQを用いた探索システム：データ登録

- ベクトル$\boldsymbol{x}_1$の登録を考える

$$\begin{bmatrix} 1.02 \\ 0.73 \\ 0.56 \\ 1.37 \\ 1.37 \\ 0.72 \end{bmatrix}$$

$\boldsymbol{x}_1$



粗量子化

$k = 1$

$k = 2$

$\vdots$

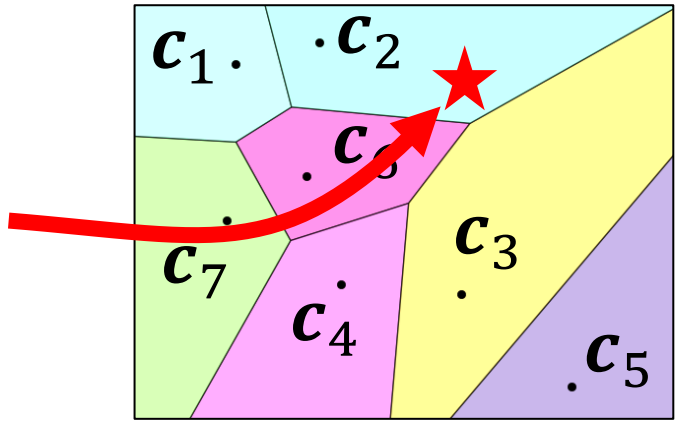
$k = K$

PQを用いた探索システム：データ登録

- ベクトル$\boldsymbol{x}_1$の登録を考える

$$\begin{bmatrix} 1.02 \\ 0.73 \\ 0.56 \\ 1.37 \\ 1.37 \\ 0.72 \end{bmatrix}$$

$\boldsymbol{x}_1$



粗量子化

$k = 1$

$k = 2$

$\vdots$

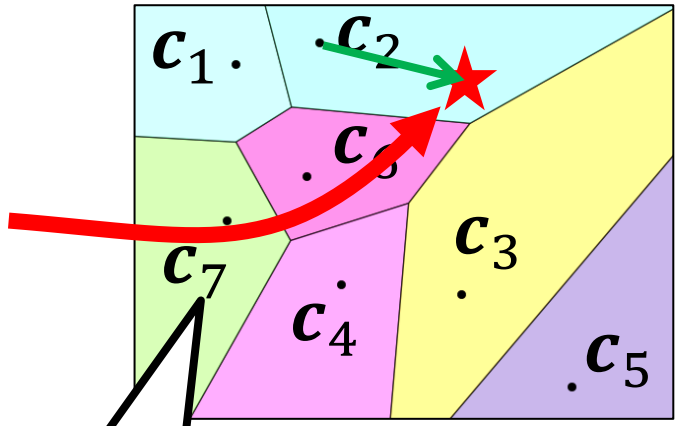
$k = K$

PQを用いた探索システム：データ登録

- ベクトル$\boldsymbol{x}_1$の登録を考える

$$\begin{bmatrix}1.02\\0.73\\0.56\\1.37\\1.37\\0.72\end{bmatrix}$$

$\boldsymbol{x}_1$



粗量子化

$k = 1$

$k = 2$

$\vdots$

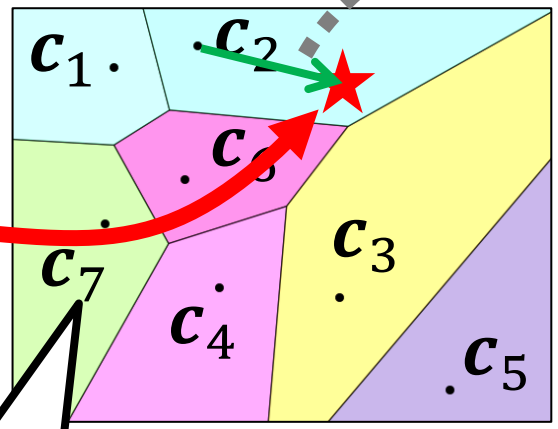
$k = K$

- $\boldsymbol{x}_1$に一番近いものは$\boldsymbol{c}_2$
- $\boldsymbol{x}_1$と$\boldsymbol{c}_2$の**残差** $\boldsymbol{r}_1 = \boldsymbol{x}_1 - \boldsymbol{c}_2$ (→) を計算する

PQを用いた探索システム：データ登録

- ベクトル$\boldsymbol{x}_1$の登録を考える

$$\boldsymbol{x}_1 = \begin{bmatrix} 1.02 \\ 0.73 \\ 0.56 \\ 1.37 \\ 1.37 \\ 0.72 \end{bmatrix}$$

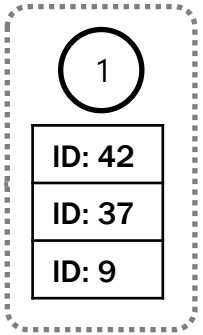


粗量子化

$k = 1$

$k = 2$

$k = K$



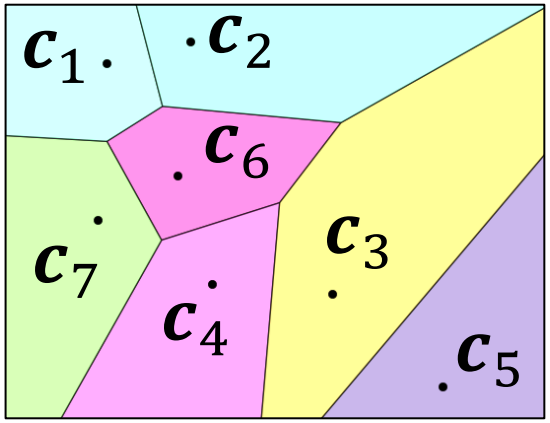
$\bar{\boldsymbol{r}}_1$

- $\boldsymbol{x}_1$に一番近いものは$\boldsymbol{c}_2$
- $\boldsymbol{x}_1$と$\boldsymbol{c}_2$の**残差** $\boldsymbol{r}_1 = \boldsymbol{x}_1 - \boldsymbol{c}_2$ (→) を計算する

- 残差 $\boldsymbol{r}_1$ をPQで圧縮し，コード$\bar{\boldsymbol{r}}_1$を作り，番号「1」とともに記録する
- すなわち，$(i, \bar{\boldsymbol{r}}_i)$を記録する

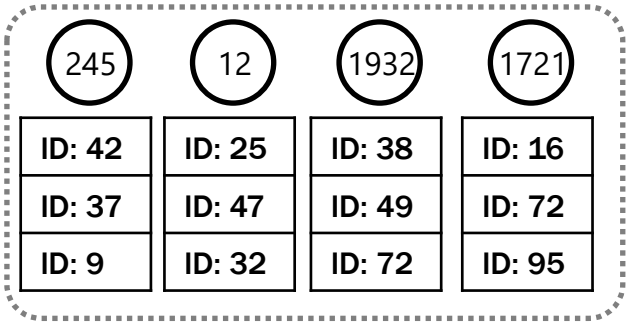
PQを用いた探索システム：データ登録

- 全データに関して，「番号＋残差」をリストとして保存

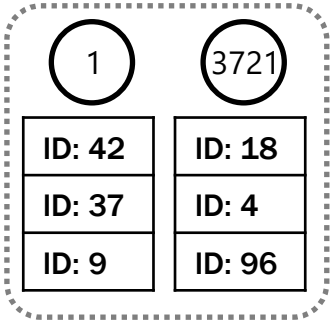


粗量子化

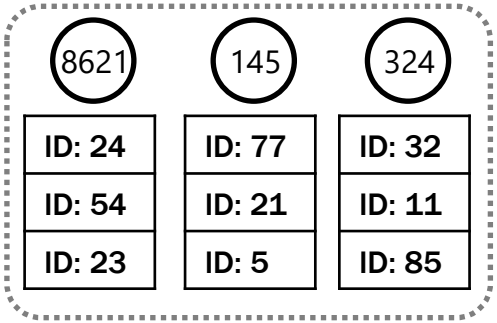
$k = 1$



$k = 2$



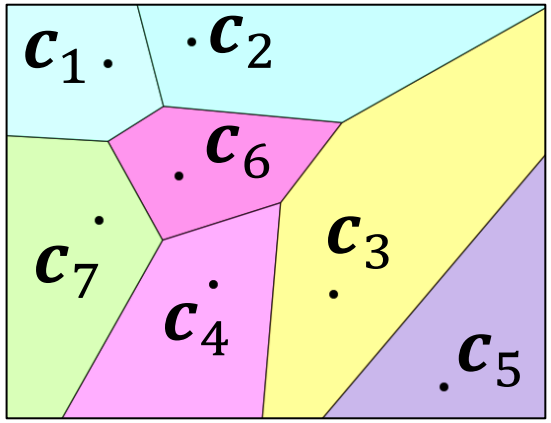
$k = K$



PQを用いた探索システム：探索

- クエリ$\boldsymbol{q}$に近いデータを探す

$$\begin{bmatrix}0.54\\2.35\\0.82\\0.42\\0.14\\0.32\end{bmatrix}$$
$\boldsymbol{q}$



粗量子化

$k = 1$

245	12	1932	1721
ID: 42	ID: 25	ID: 38	ID: 16
ID: 37	ID: 47	ID: 49	ID: 72
ID: 9	ID: 32	ID: 72	ID: 95

$k = 2$

1	3721
ID: 42	ID: 18
ID: 37	ID: 4
ID: 9	ID: 96

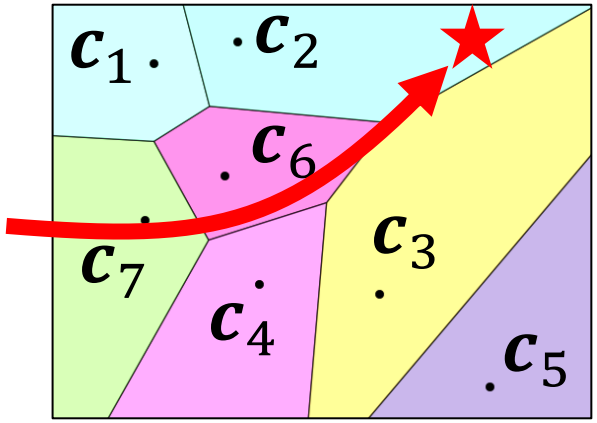
$k = K$

8621	145	324
ID: 24	ID: 77	ID: 32
ID: 54	ID: 21	ID: 11
ID: 23	ID: 5	ID: 85

PQを用いた探索システム：探索

- クエリqに近いデータを探す

$$q = \begin{bmatrix} 0.54 \\ 2.35 \\ 0.82 \\ 0.42 \\ 0.14 \\ 0.32 \end{bmatrix}$$



粗量子化

$k = 1$

245	12	1932	1721
ID: 42	ID: 25	ID: 38	ID: 16
ID: 37	ID: 47	ID: 49	ID: 72
ID: 9	ID: 32	ID: 72	ID: 95

$k = 2$

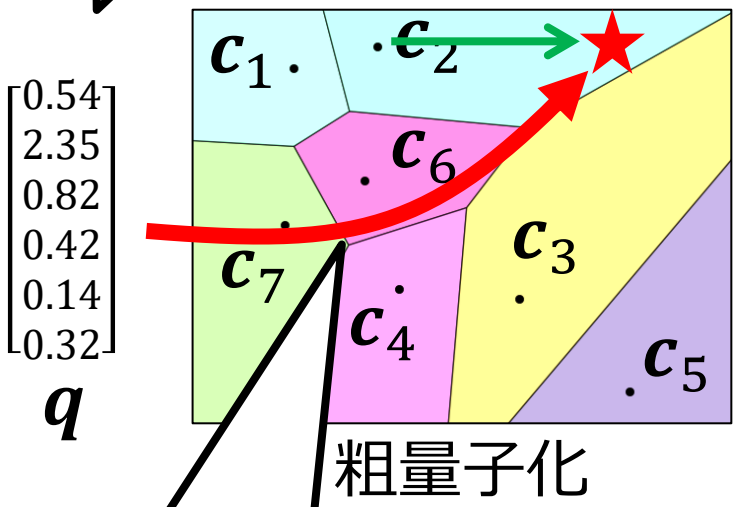
1	3721
ID: 42	ID: 18
ID: 37	ID: 4
ID: 9	ID: 96

$k = K$

8621	145	324
ID: 24	ID: 77	ID: 32
ID: 54	ID: 21	ID: 11
ID: 23	ID: 5	ID: 85

PQを用いた探索システム：探索

- クエリqに近いデータを探す



- qに一番近いものはc_2
- qとc_2の**残差** $r_q = q - c_2$ を計算する

$k = 1$

245	12	1932	1721
ID: 42	ID: 25	ID: 38	ID: 16
ID: 37	ID: 47	ID: 49	ID: 72
ID: 9	ID: 32	ID: 72	ID: 95

$k = 2$

1	3721
ID: 42	ID: 18
ID: 37	ID: 4
ID: 9	ID: 96

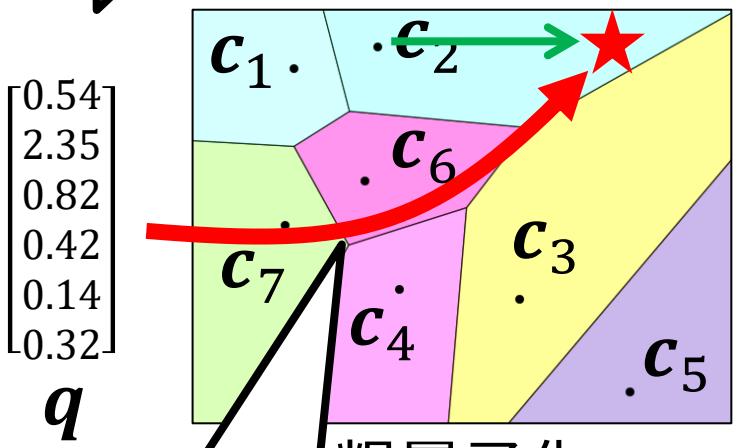
$\vdots$

$k = K$

8621	145	324
ID: 24	ID: 77	ID: 32
ID: 54	ID: 21	ID: 11
ID: 23	ID: 5	ID: 85

PQを用いた探索システム：探索

- クエリ$\boldsymbol{q}$に近いデータを探す



- $\boldsymbol{q}$に一番近いものは$\boldsymbol{c}_2$
- $\boldsymbol{q}$と$\boldsymbol{c}_2$の**残差** $\boldsymbol{r}_q = \boldsymbol{q} - \boldsymbol{c}_2$ を計算する

$k = 1$

245	12	1932	1721
ID: 42	ID: 25	ID: 38	ID: 16
ID: 37	ID: 47	ID: 49	ID: 72
ID: 9	ID: 32	ID: 72	ID: 95

$k = 2$

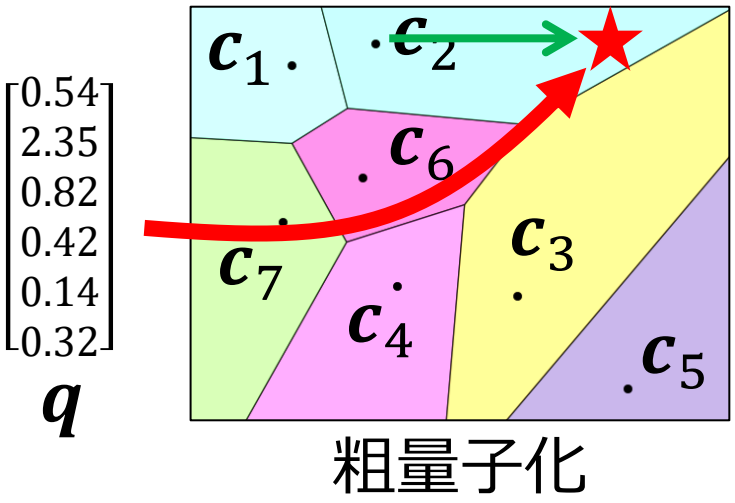
1	3721
ID: 42	ID: 18
ID: 37	ID: 4
ID: 9	ID: 96

- $k = 2$に登録されている各$(i, \bar{\boldsymbol{r}}_i)$について，$\boldsymbol{r}_q$と比べる

$$d(\boldsymbol{q}, \boldsymbol{x}_i)^2 = d(\boldsymbol{q} - \boldsymbol{c}_2, \boldsymbol{x}_i - \boldsymbol{c}_2)^2$$
$$= d(\boldsymbol{r}_q, \boldsymbol{r}_i)^2 \sim d_A(\boldsymbol{r}_q, \bar{\boldsymbol{r}}_i)^2$$

- 最も近いものを選ぶ（リランキング．戦略色々）

PQを用いた探索システム：探索



粗量子化

- 粗量子化のコスト＋リランキングのコストを調整することで，高速な探索を実現
- 残差に注目することで，近似精度を高めた

$k = 1$

245	12	1932	1721
ID: 42	ID: 25	ID: 38	ID: 16
ID: 37	ID: 47	ID: 49	ID: 72
ID: 9	ID: 32	ID: 72	ID: 95

$k = 2$

1	3721
ID: 42	ID: 18
ID: 37	ID: 4
ID: 9	ID: 96

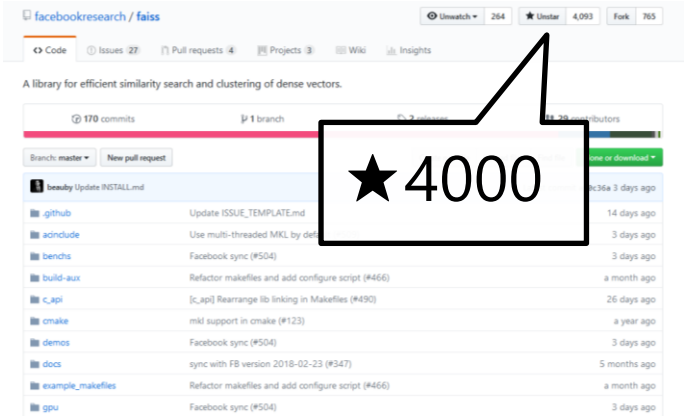
$k = K$

8621	145	324
ID: 24	ID: 77	ID: 32
ID: 54	ID: 21	ID: 11
ID: 23	ID: 5	ID: 85

Faiss

https://github.com/facebookresearch/faiss

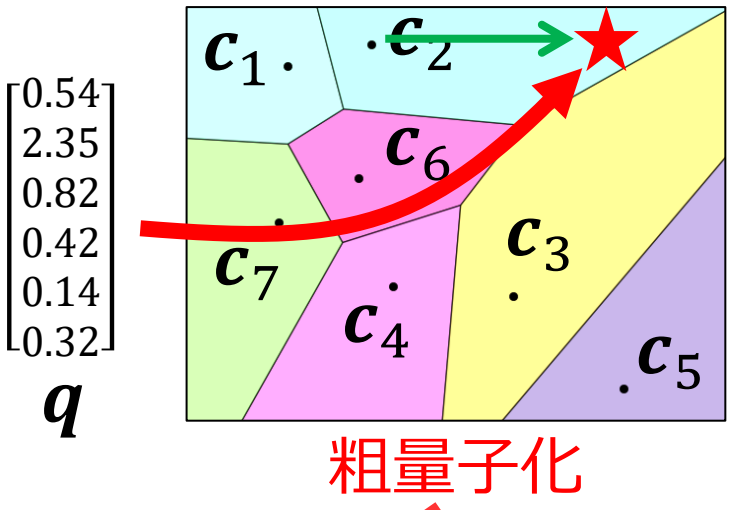
```
$> conda install faiss-cpu -c pytorch
$> conda install faiss-gpu -c pytorch
```



- PQの著者ら（もともとINRIA）がFAIR Parisに移籍し、彼らがもともと持っていたyaelというライブラリを元に、GPU専門家と一緒に作ったANNライブラリ
- CPU版：PQベースの手法を網羅
- GPU版：PQベースの手法の一部を実装

- ボーナス：
  - 通常の線形探索も実装されており、CPU版もGPU版も非常に高速
  - k-meansも実装されており、CPU版もGPU版も非常に高速

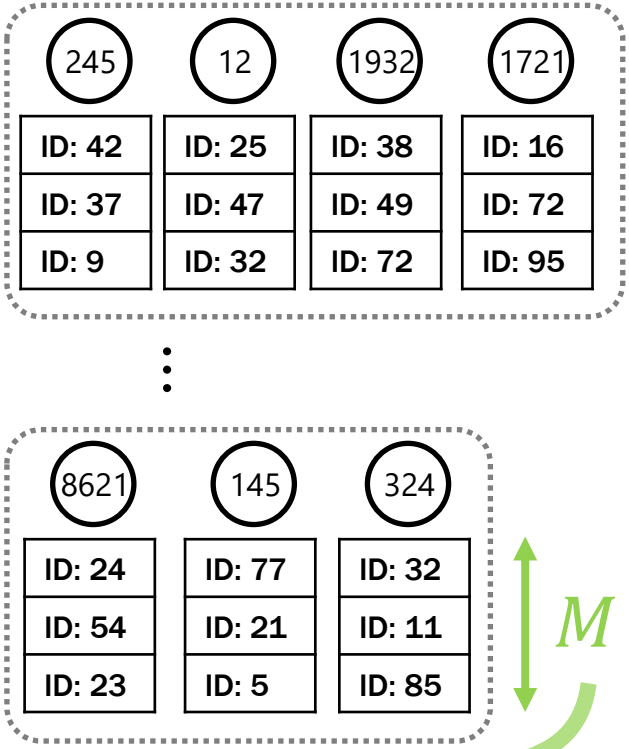
  ベンチマーク：
  https://github.com/DwangoMediaVillage/pqkmeans/blob/master/tutorial/4_comparison_to_faiss.ipynb



粗量子化

$k = 1$

$k = K$

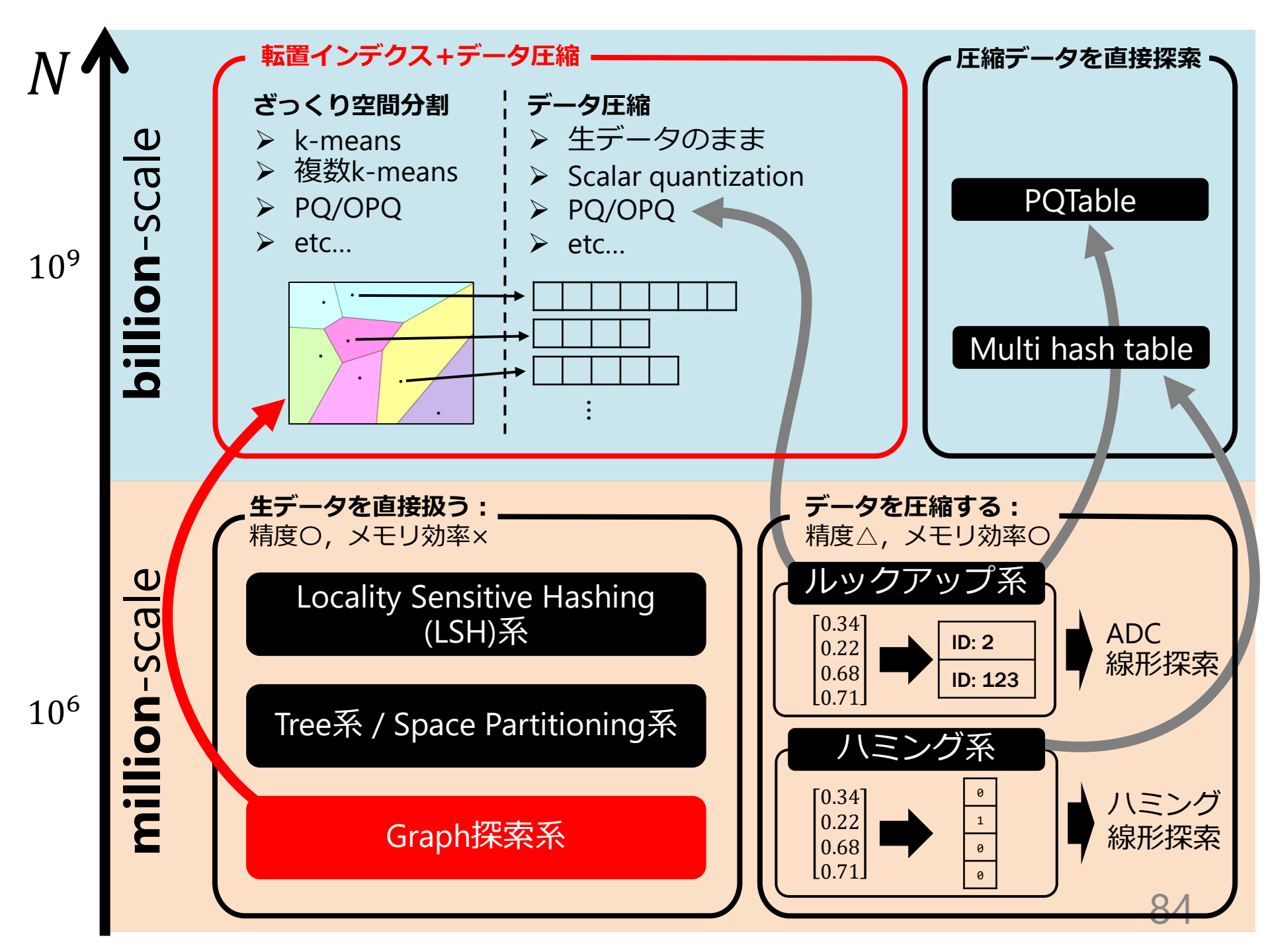


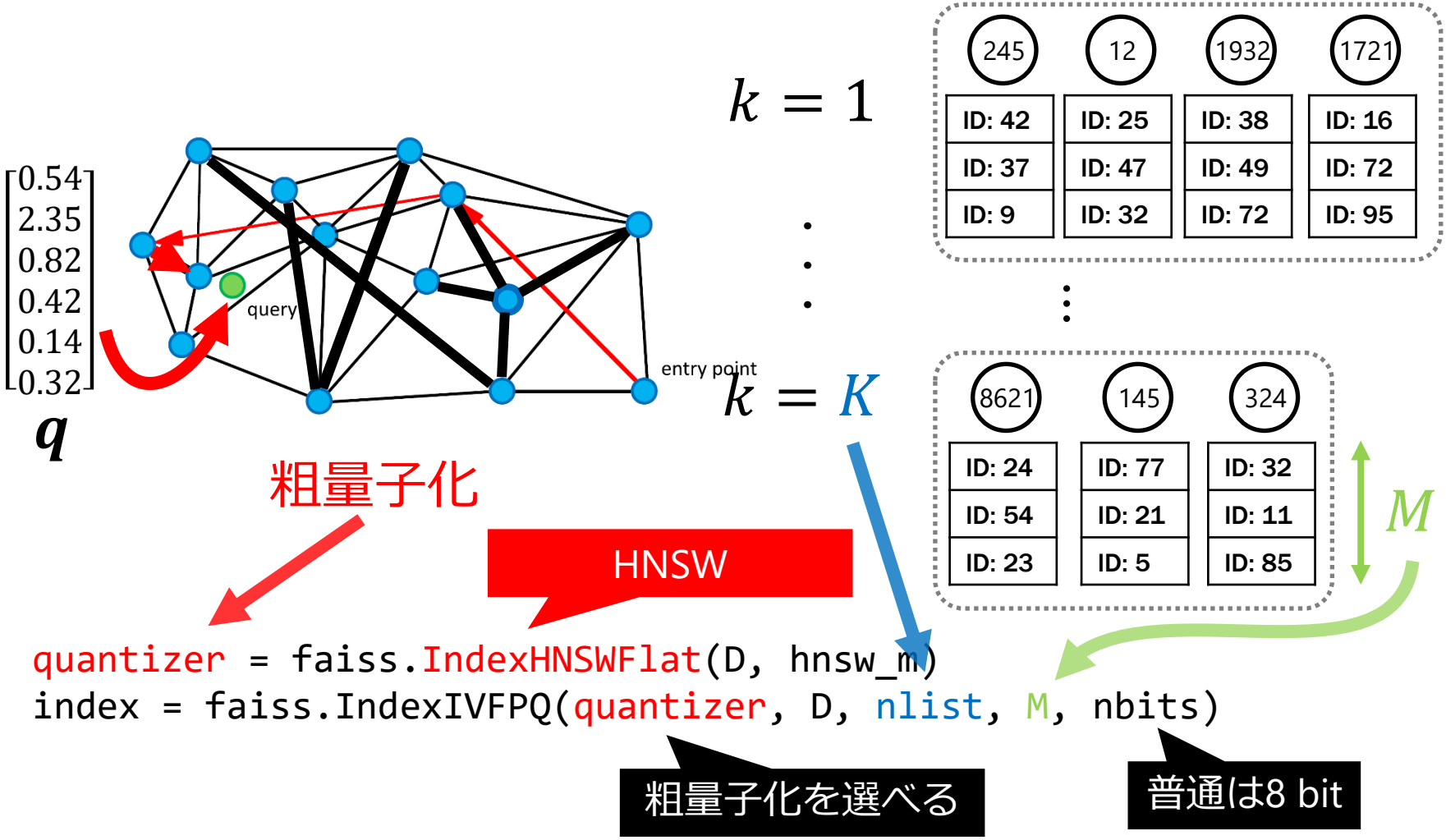
普通の線形探索

```
quantizer = faiss.IndexFlatL2(D)
index = faiss.IndexIVFPQ(quantizer, D, nlist, M, nbits)

index.train(Xt)  # 学習

index.add(X)  # データ追加

index.nprobe = nprobe  # 検索パラメータ

dist, ids = index.search(Q, topk)  # 検索
```

粗量子化を選べる

普通は8 bit


N
billion-scale
10^9
million-scale
10^6
転置インデクス+データ圧縮
ざっくり空間分割
k-means
複数k-means
PQ/OPQ
etc…
データ圧縮
生データのまま
Scalar quantization
PQ/OPQ
etc…
圧縮データを直接探索
PQTable
Multi hash table
生データを直接扱う：
精度○，メモリ効率×
Locality Sensitive Hashing (LSH)系
Tree系 / Space Partitioning系
Graph探索系
データを圧縮する：
精度△，メモリ効率○
ルックアップ系
0.34
0.22
0.68
0.71
ID: 2
ID: 123
ADC
線形探索
ハミング系
0.34
0.22
0.68
0.71
0
1
0
0
ハミング
線形探索



- 粗量子化をHNSWにすると、billion-scaleのデータに対する速度・精度のトレードオフで2018年現在最も有効
- [Douze+, CVPR 2018] [Baranchuk+, ECCV 2018]

☺

- SOTAの研究者が作っており、高速。PQ系最新手法は今後このライブラリ上にインプリされそう
- FAIRで実際に使われている
- 元のデータが直接メモリに載らないならfaiss一択（billion-scaleの問題など）
- 特にクエリがバッチのとき非常に高速（逆に、バッチでないときはそこまで劇的に早くはない）

☹

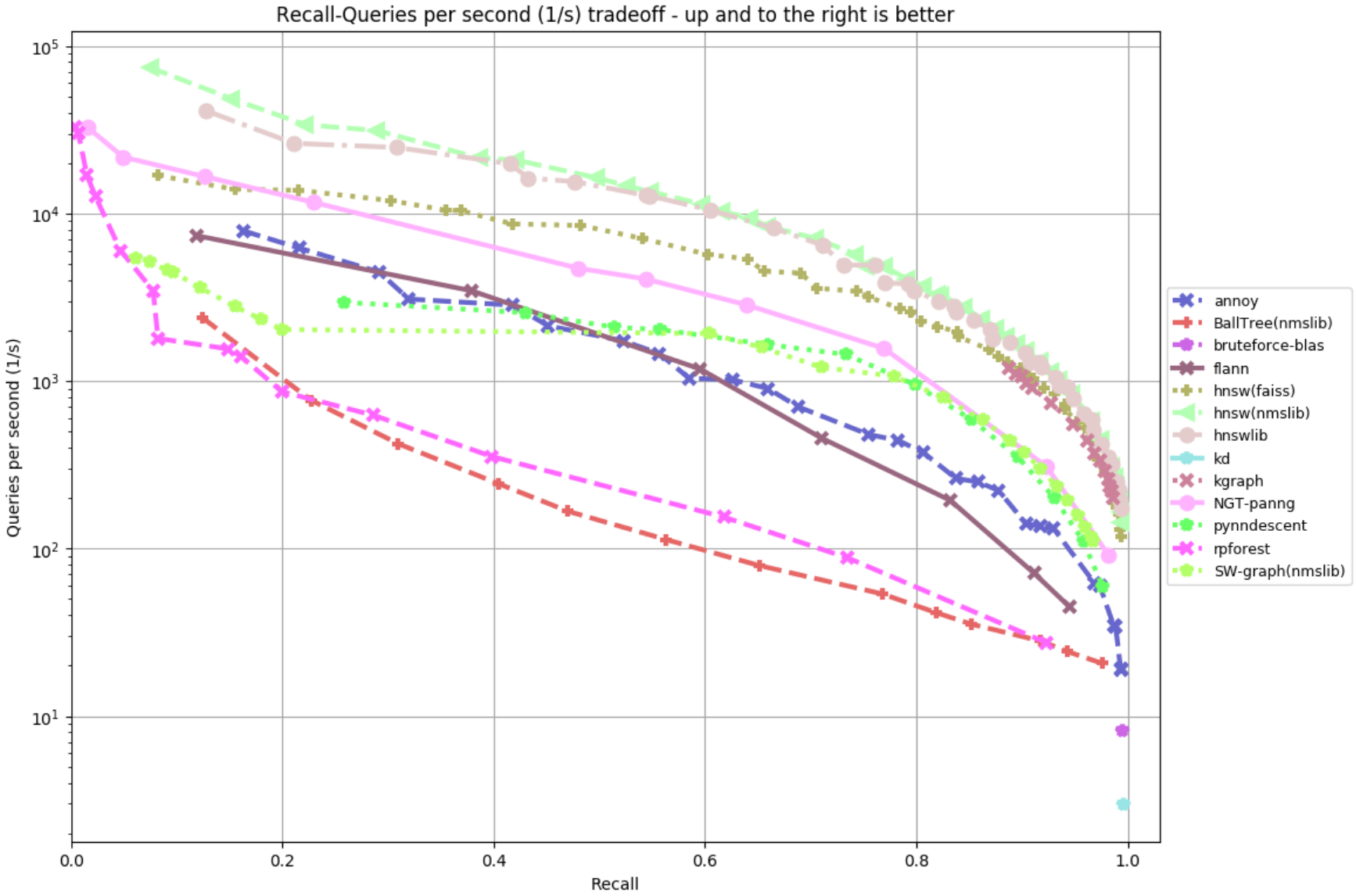
- ドキュメントが不足。特にpythonバインディングとGPU
- conda必須（自前ビルドは苦行）
- 様々な手法がインプリされているが、専門家じゃないとどれを使えばいいかわからない
- C++をswigでPythonから読んでおり、pythonなのにc++的な処理（オブジェクトの所有者、deleteしていいかどうか、等）を考えなければならないことがある

参考資料

- Faissのwiki: [link]
- Faiss tips: [link]
- Lookup系の様々な手法のjulia実装 [link]
- PQ元論文：H. Jégou et al., “Product quantization for nearest neighbor search,” TPAMI 2011
- IVFADC + HNSW (1): M. Douze et al., “Link and code: Fast indexing with graphs and compact regression codes,” CVPR 2018
- IVFADC + NHSW (2): D. Baranchuk et al., “Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors,” ECCV 2018

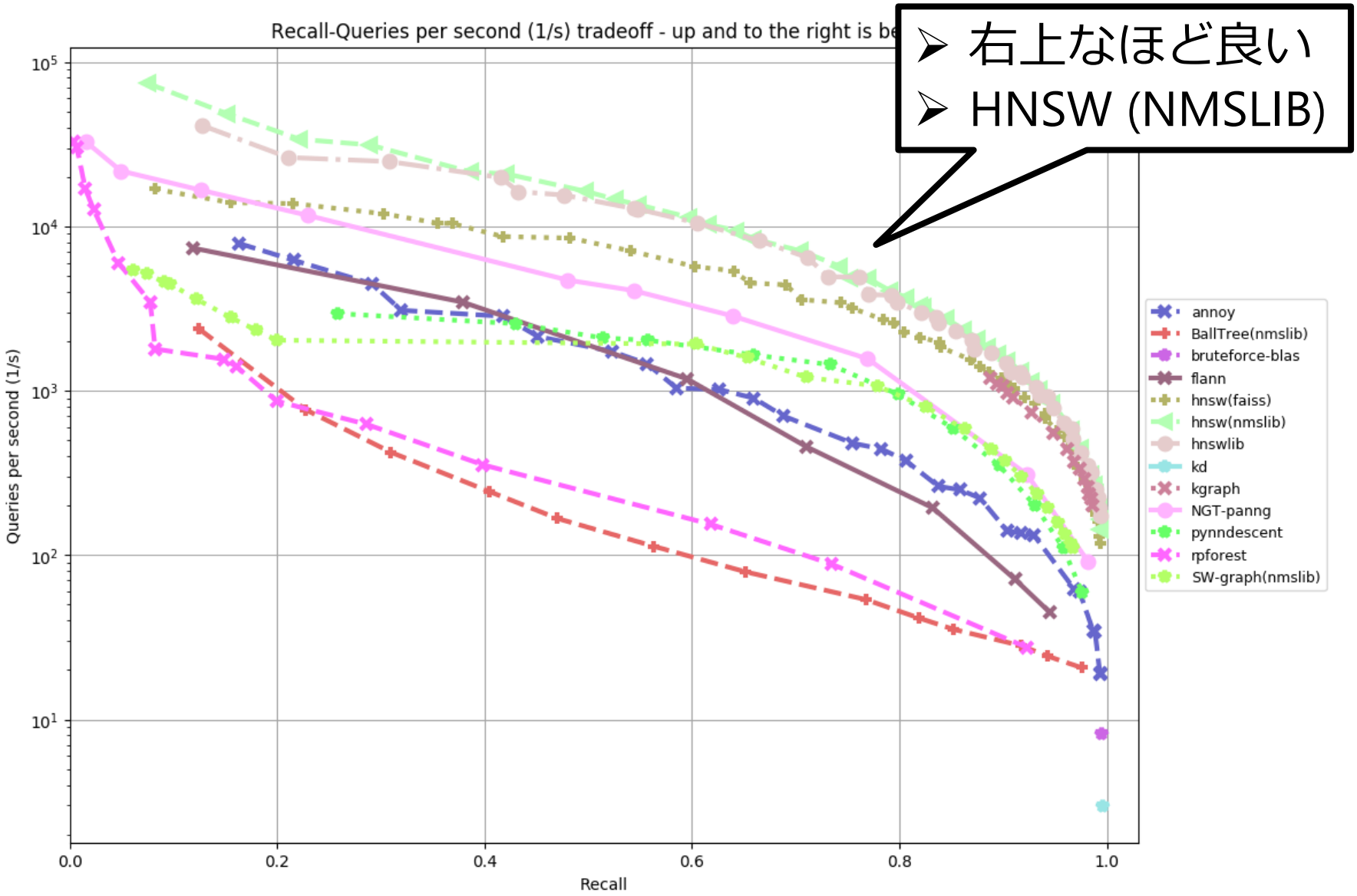
ベンチマーク

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



ベンチマーク

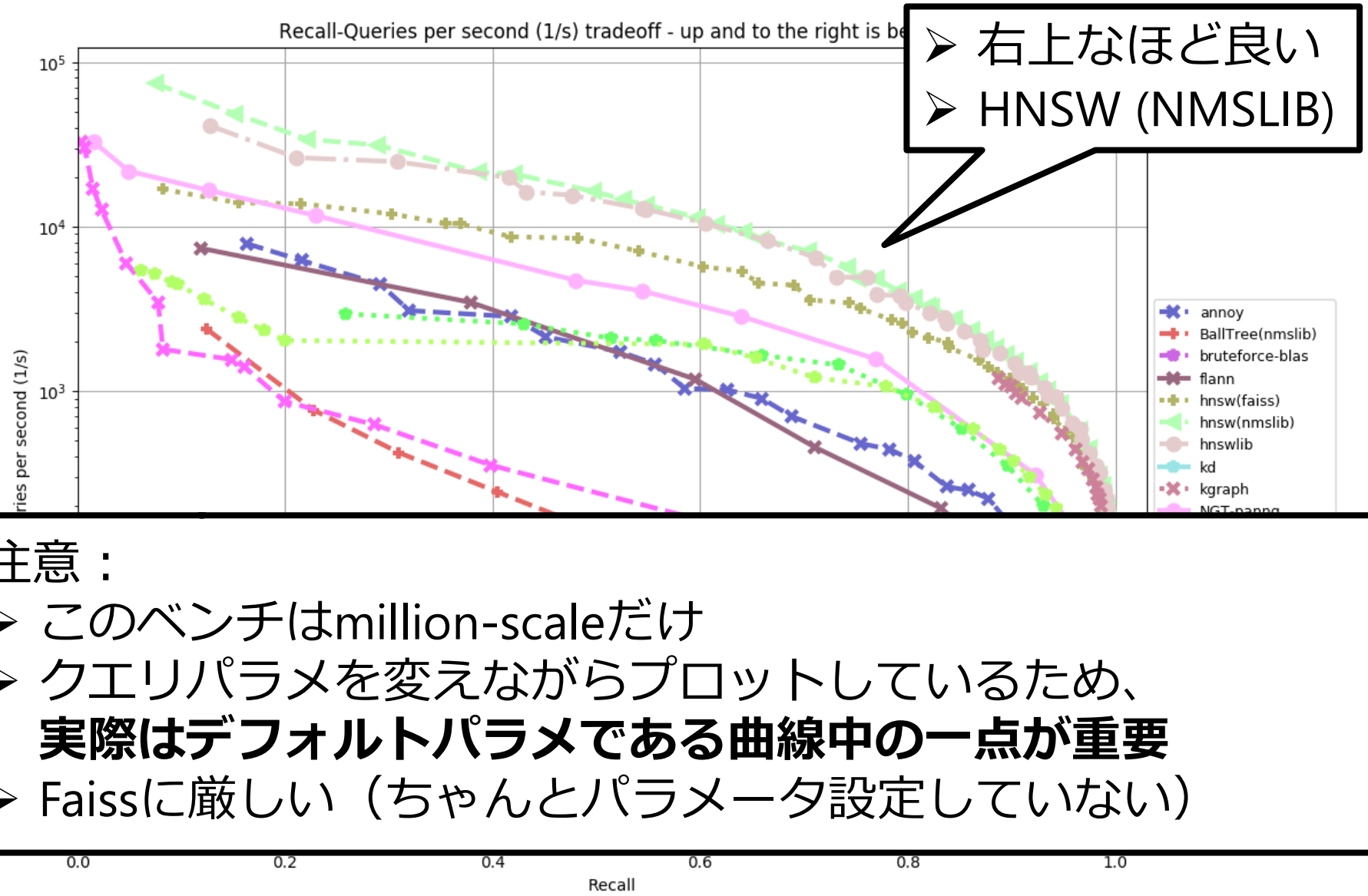
- https://github.com/erikbern/ann-benchmarks



- 右上なほど良い
- HNSW (NMSLIB)

ベンチマーク

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

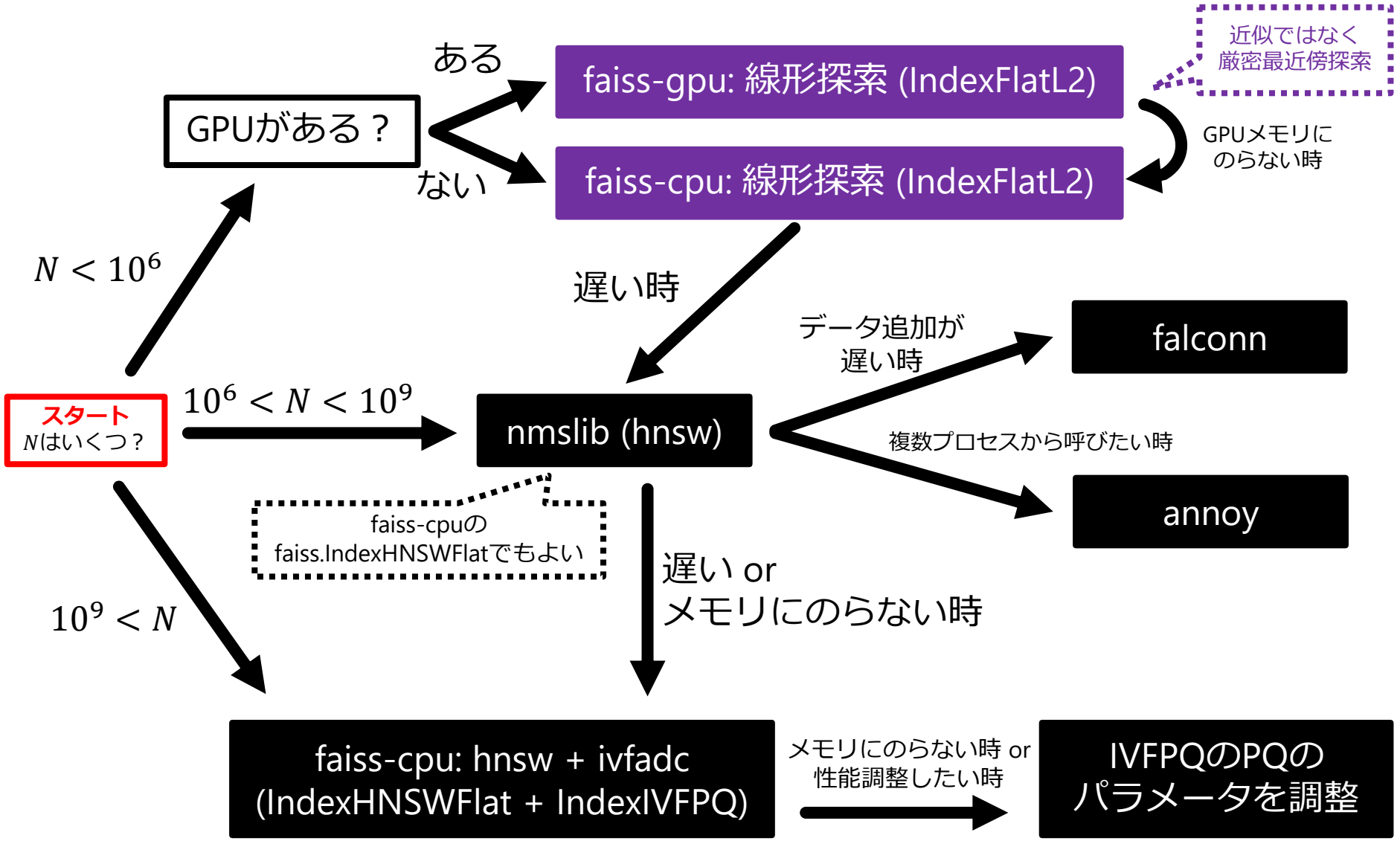


- 右上なほど良い
- HNSW (NMSLIB)

注意：

- このベンチはmillion-scaleだけ
- クエリパラメを変えながらプロットしているため、**実際はデフォルトパラメである曲線中の一点が重要**
- Faissに厳しい（ちゃんとパラメータ設定していない）

PythonのおすすめのANN手法選択フローチャート（2018年度版）



注意：$D = 100$ぐらいを想定している。問題のサイズは大体DNで決まる。
Dが1,000や10,000のときはまずPCAを施し$D = 100$程度にする場合が多い

ANNの問題

- 数学的背景がない。実データで実測で早ければ良いということになっている
  - LSHのころは近似最近傍探索問題が数学的に定義されていたが、近年のデータ依存系ではスルーされている

- なので、手法そのものが良いのか、あるデータに対して偶然強いのか、実装が良いのか、分離が難しい
  - 事実、faissはSIMD芸の恩恵が大きいし、またバックエンドがOpenBLASかIntel MKLかで数倍速度が違う

- データセット不足。現在billion級がSIFT1BとDeep1Bしかないのでこれらで測るしかない。しかしそれでいいのか？
  - 事実、どうやら2010年代のbillion系ANNはSIFT1B（要素が0-255のヒストグラム、ブロックワイズな相関、128D）に特化していたような印象がある
  - billion以上のデータセットを作るのはそもそも大変そう

ANNの問題

- 「全体から探す」以外の処理の議論があまり無い
  - 「データの追加、削除」に対する速度・メモリの議論
  - SQLやElasticsearchと組み合わせられるか？

    G. Amato et al., “Large-Scale Image Retrieval with Elasticsearch,” SIGIR short paper 2018
    C. Mu et al., “Towards Practical Visual Search Engine Within Elasticsearch,” SIGIR Workshop 2018

  - 全体集合ではなく部分集合に対する検索

- Y. Matsui, R. Hinami, and S. Satoh, “Reconfigurable Inverted Index,” ACM Multimedia 2018 (Oral) [github]
- `pip install rii`