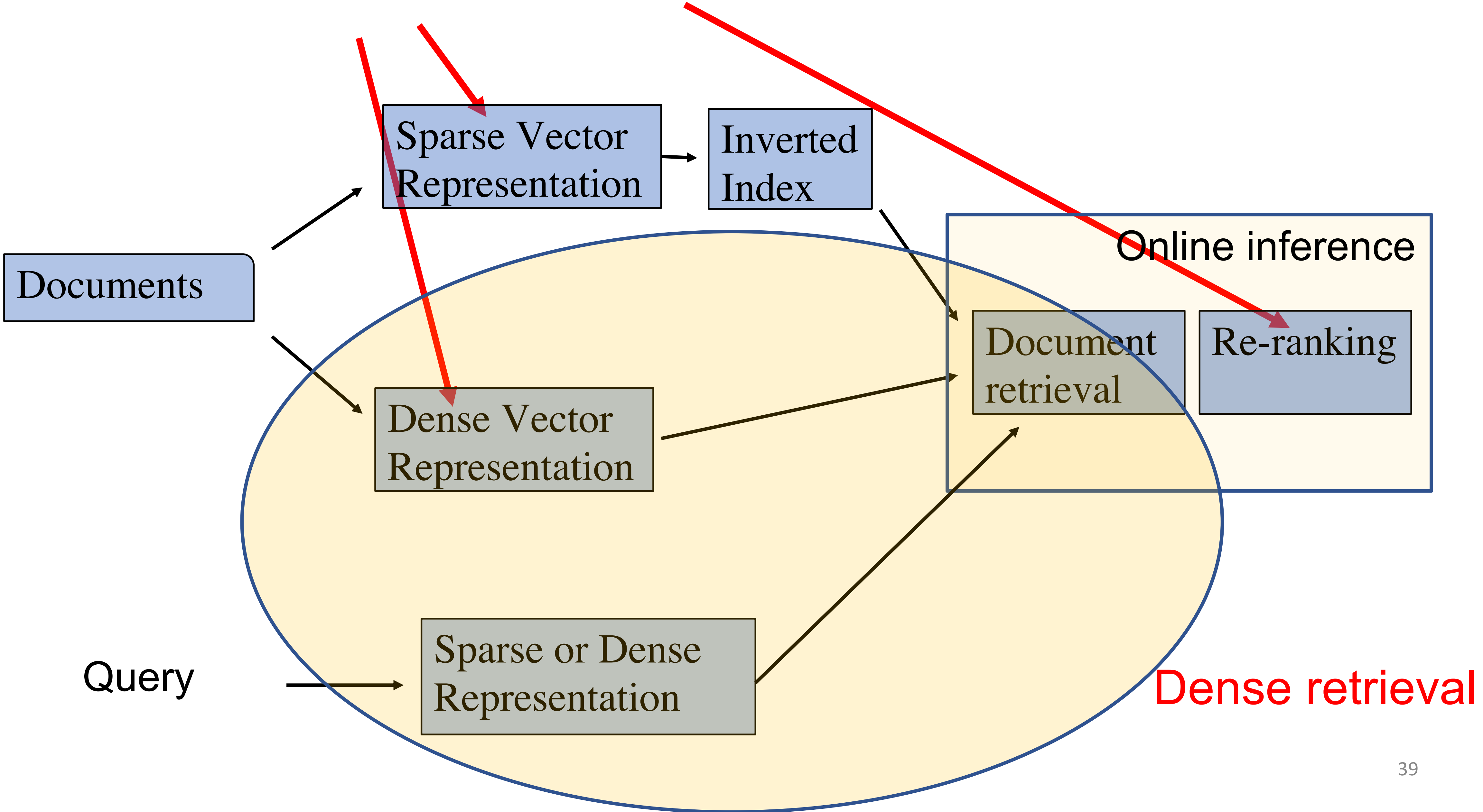
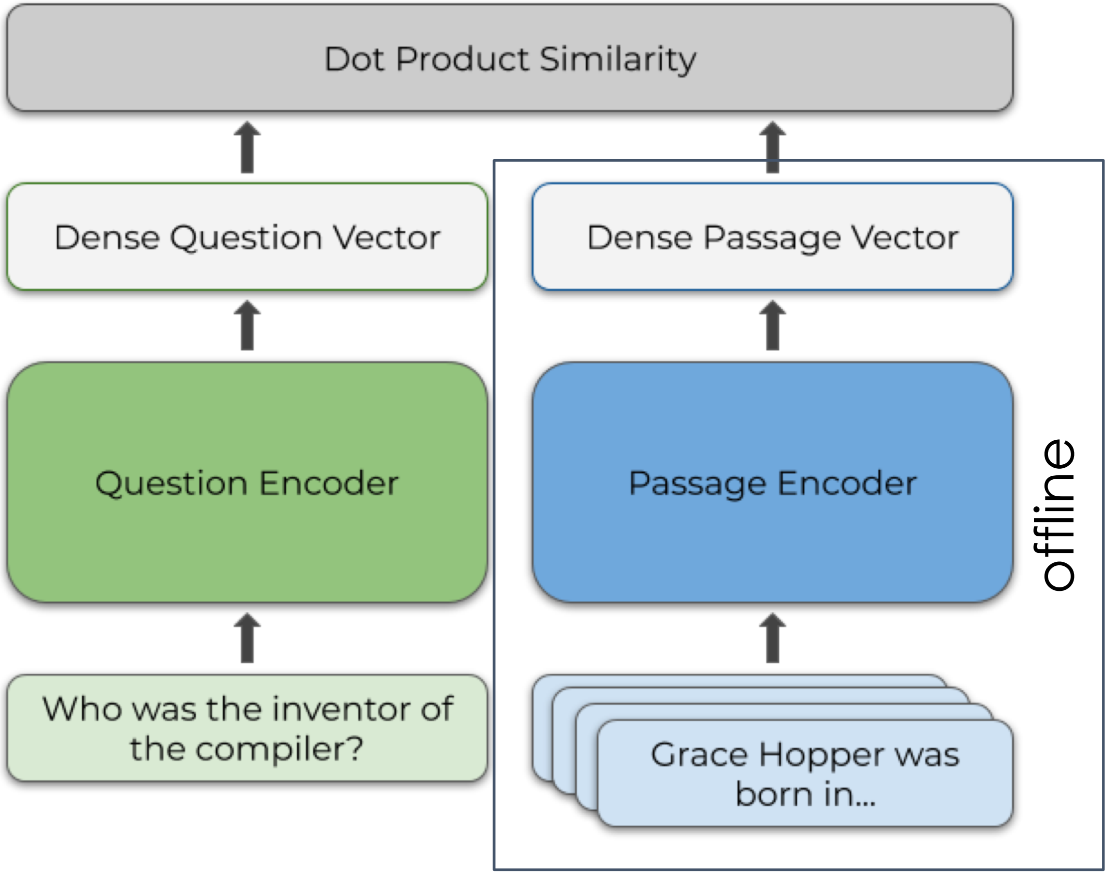
Neural Models for Information Retrieval: Where are we?



Dense Retrieval: Basic Computation Flow and Techniques



Queries and documents are encoded into single vectors respectively.

- **Time and efficiency optimization**
  a. Nearest Neighbor Search with Approximation
     - GPU implementation: FAISS [Facebook AI, 2017]
  b. Vector Compression
     - Product quantization (PQ), RepCONC [Zhan et al., WSDM22]

- **Vector representations**
  a. Multi Vectors, e.g., ColBERT [Khattab et al., SIGIR20]
  b. Single Vectors, e.g., TCT-ColBERT [Lin et al., arXiv20], DPR [Karpukhin et al., ACL20]

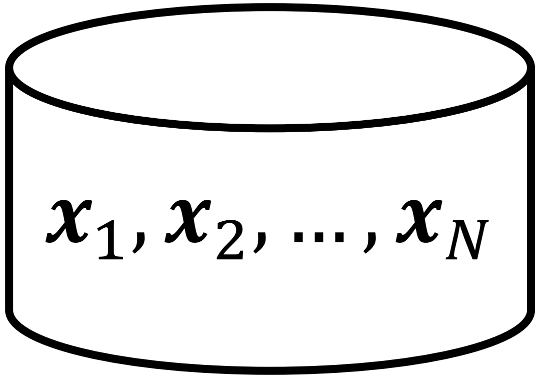
- **Training methods**
  a. Negative doc selection, e.g., DPR [Karpukhin et al., ACL20], ANCE [Microsoft, ICLR 21]
  b. Distillation, e.g., TCT-ColBERT [Lin et al., arXiv20], RocketQA [Qu et al., ACL21]

Billion-scale Approximate Nearest Neighbor Search

Yusuke Matsui
The University of Tokyo

Edited for CS293S

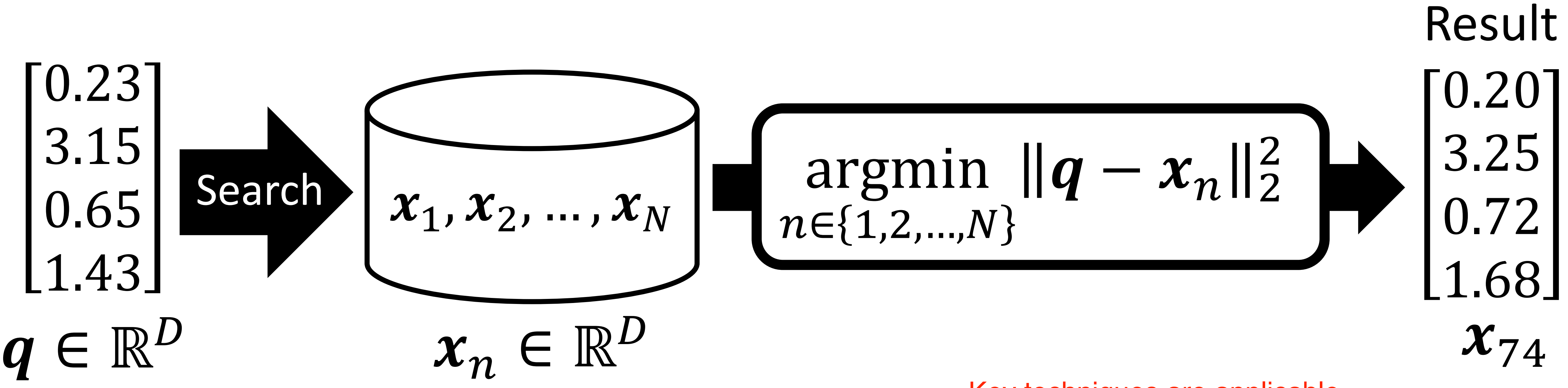
Nearest Neighbor Search; NN



$\boldsymbol{x}_n \in \mathbb{R}^D$

- N D-dim database vectors: $\{\boldsymbol{x}_n\}_{n=1}^{N}$

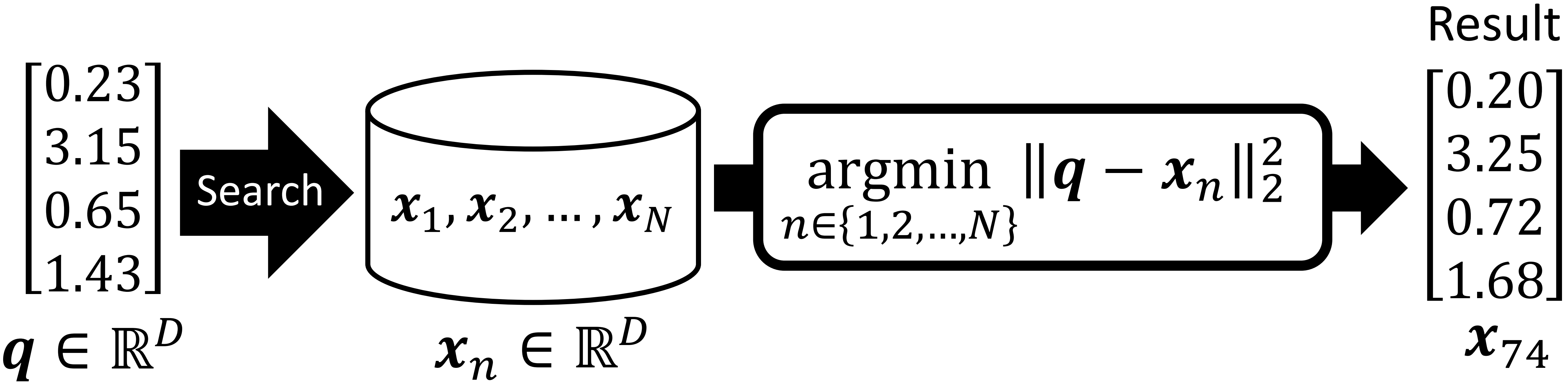
Nearest Neighbor Search; NN



Key techniques are applicable
to dot-product based similarity search
because ||q-x||^2 = ||q||^2 -2q*x +||x||^2

- ➢ N D-dim database vectors: $\{\boldsymbol{x}_n\}_{n=1}^N$
- ➢ Given a query $\boldsymbol{q}$, find the closest vector from the database
- ➢ One of the fundamental problems in computer science
- ➢ Solution: linear scan, $O(ND)$, slow ☹

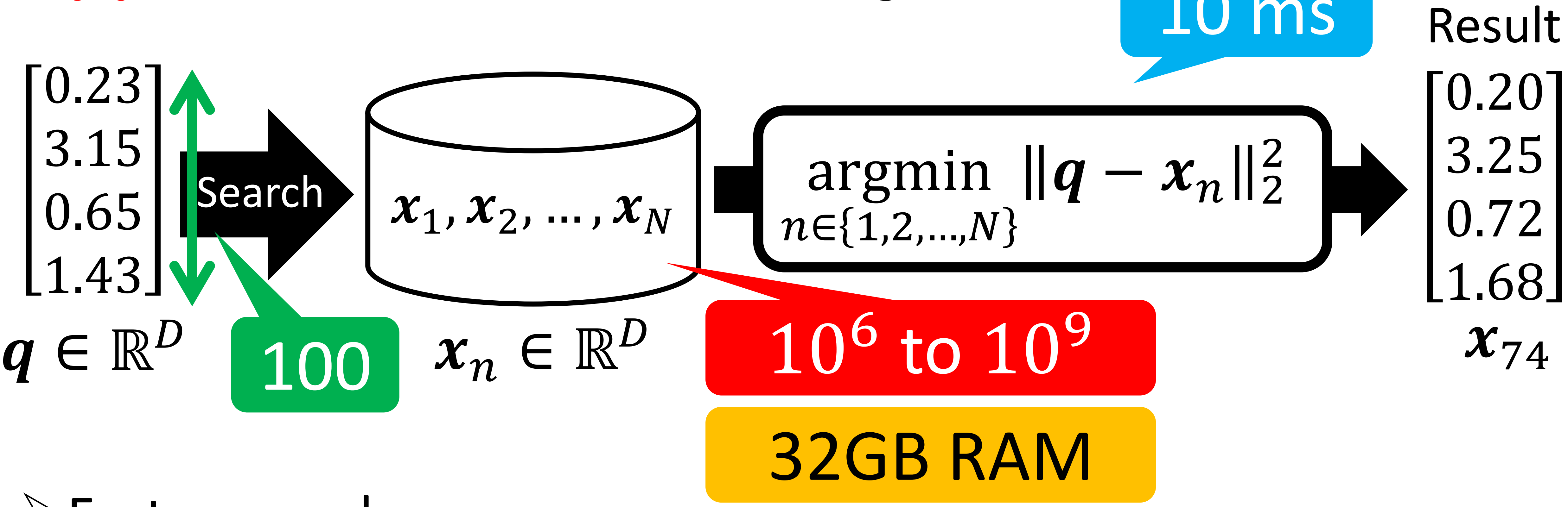
Approximate Nearest Neighbor Search; ANN



May also use other similarity measures such as dot-products

- ➢Faster search
- ➢Don’t necessarily have to be exact neighbors
- ➢Trade off: runtime, accuracy, and memory-consumption

Approximate Nearest Neighbor Search: ANN



- Faster search
- Don't necessarily have to be exact neighbors
- Trade off: runtime, accuracy, and memory-consumption
- A sense of scale: billion-scale data on memory

Part 1:

Nearest Neighbor Search

Part 2:

Approximate Nearest Neighbor Search

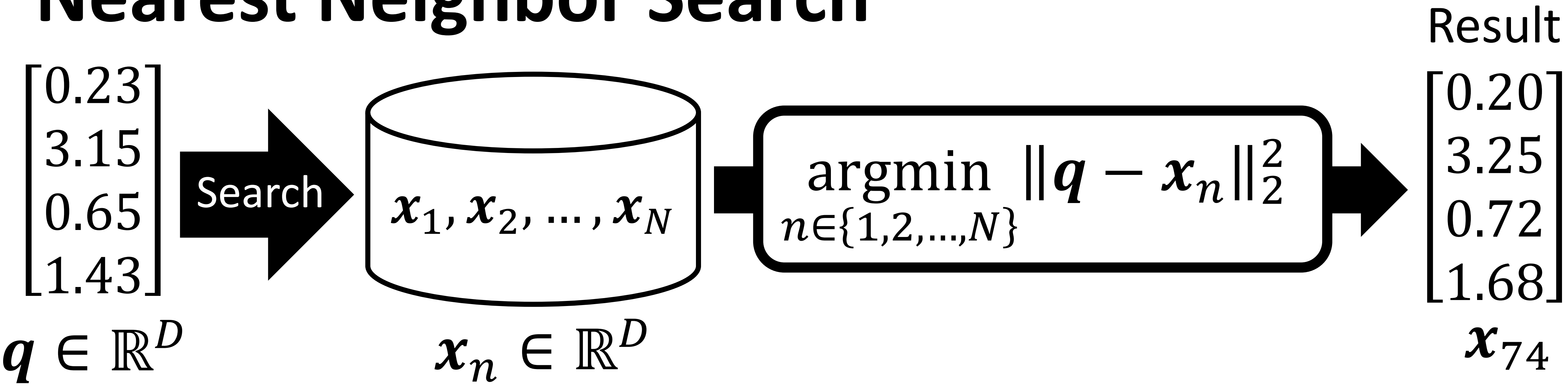
Part 1:

Nearest Neighbor Search

Part 2:

Approximate Nearest Neighbor Search

Nearest Neighbor Search



- Should try this first of all
- Introduce a naïve implementation
- Introduce a fast implementation
  - ✓ **Faiss** library from FAIR (you'll see many times today. CPU & GPU)
- Experience the drastic difference between the two impls

Other similarity measures such as dot-products are available

M D-dim query vectors $\quad \mathcal{Q} = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \dots, \boldsymbol{q}_M\}$

N D-dim database vectors $\quad \mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\} \quad M \ll N$

Task：Given $\boldsymbol{q} \in \mathcal{Q}$ and $\boldsymbol{x} \in \mathcal{X}$, compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2$

M D-dim query vectors $\mathcal{Q} = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \dots, \boldsymbol{q}_M\}$

N D-dim database vectors $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\}$ $M \ll N$

Task：Given $\boldsymbol{q} \in \mathcal{Q}$ and $\boldsymbol{x} \in \mathcal{X}$, compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2$

Naïve impl.

```
def l2sqr(q, x):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (q[d] - x[d])**2
  return diff
```

```
parfor q in Q:
  for x in X:
    l2sqr(q, x)
```

Parallelize query-side

Select min by heap, but omit it now

M D-dim query vectors $\quad Q = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_M\}$

N D-dim database vectors $\quad \mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ $\quad M \ll N$

Task : Given $\boldsymbol{q} \in Q$ and $\boldsymbol{x} \in \mathcal{X}$, compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2$

Naïve impl.

```
def l2sqr(q, x):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (q[d] - x[d])**2
  return diff
```

```
parfor q in Q:
  for x in X:
    l2sqr(q, x)
```

Parallelize query-side

Select min by heap, but omit it now

faiss impl.

if $M < 20$:

compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2$ by SIMD

SIMD instructions provided by Intel/AMD CPU

else :

compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2 = \|\boldsymbol{q}\|_2^2 - 2\boldsymbol{q}^\top \boldsymbol{x} + \|\boldsymbol{x}\|_2^2$ by BLAS

Basic Linear Algebra Subroutine Library

M D-dim query vectors $\mathcal{Q} = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \dots, \boldsymbol{q}_M\}$

N D-dim database vectors $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\}$ $M \ll N$

Task：Given $\boldsymbol{q} \in \mathcal{Q}$ and $\boldsymbol{x} \in \mathcal{X}$, compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2$

Naïve impl.

```
def l2sqr(q, x):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (q[d] - x[d])**2
  return diff
```

```
parfor q in Q:
  for x in X:
    l2sqr(q, x)
```

Parallelize query-side

Select min by heap, but omit it now

faiss impl.

if $M < 20$:

compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2$ by SIMD

else :

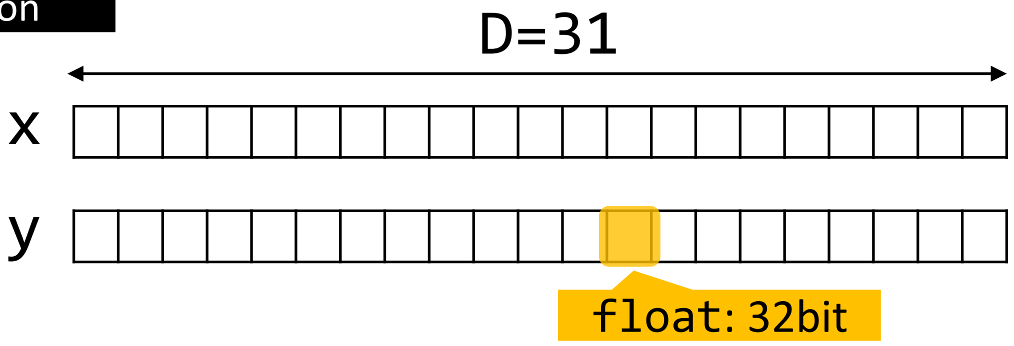
compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2 = \|\boldsymbol{q}\|_2^2 - 2\boldsymbol{q}^\top \boldsymbol{x} + \|\boldsymbol{x}\|_2^2$ by BLAS

Basic Linear Algebra Subroutine Library

$\|\boldsymbol{x}-\boldsymbol{y}\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



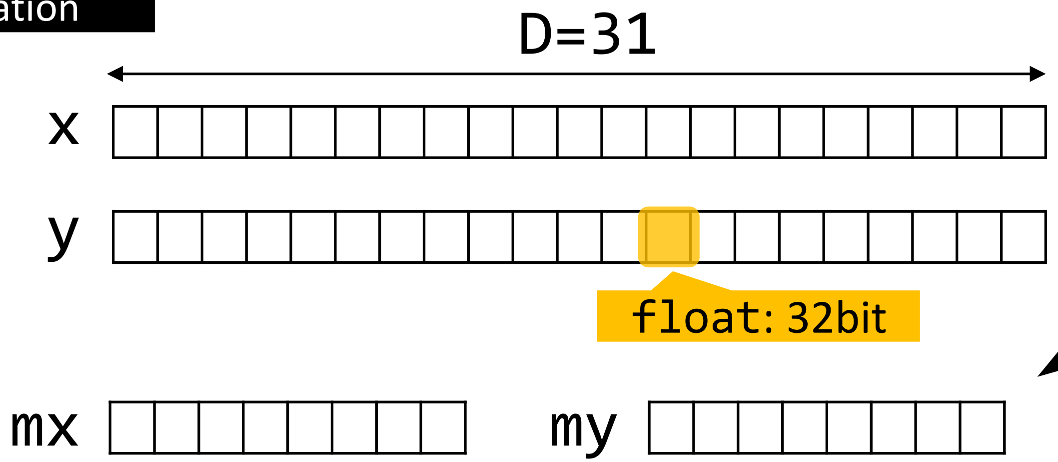
Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

$\|\boldsymbol{x}-\boldsymbol{y}\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



- 256bit SIMD Register
- Process eight floats at once

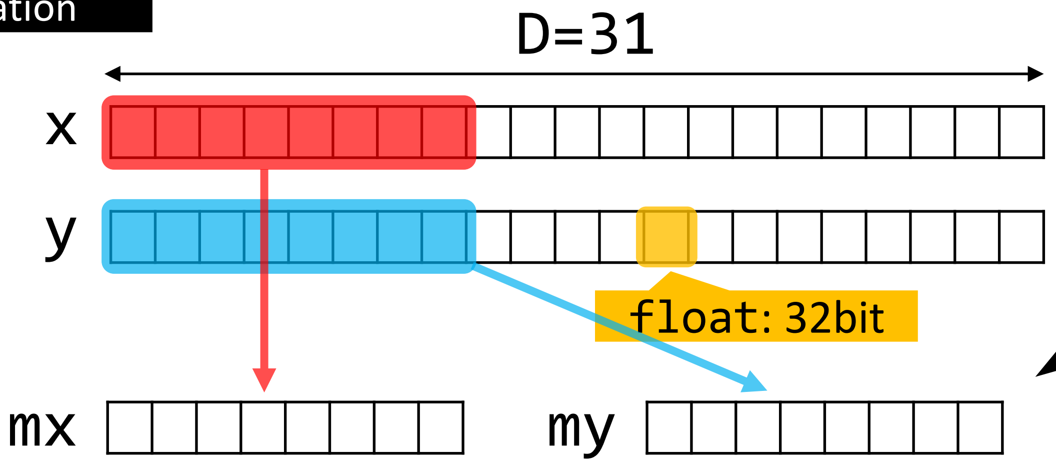
Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



- 256bit SIMD Register
- Process eight floats at once

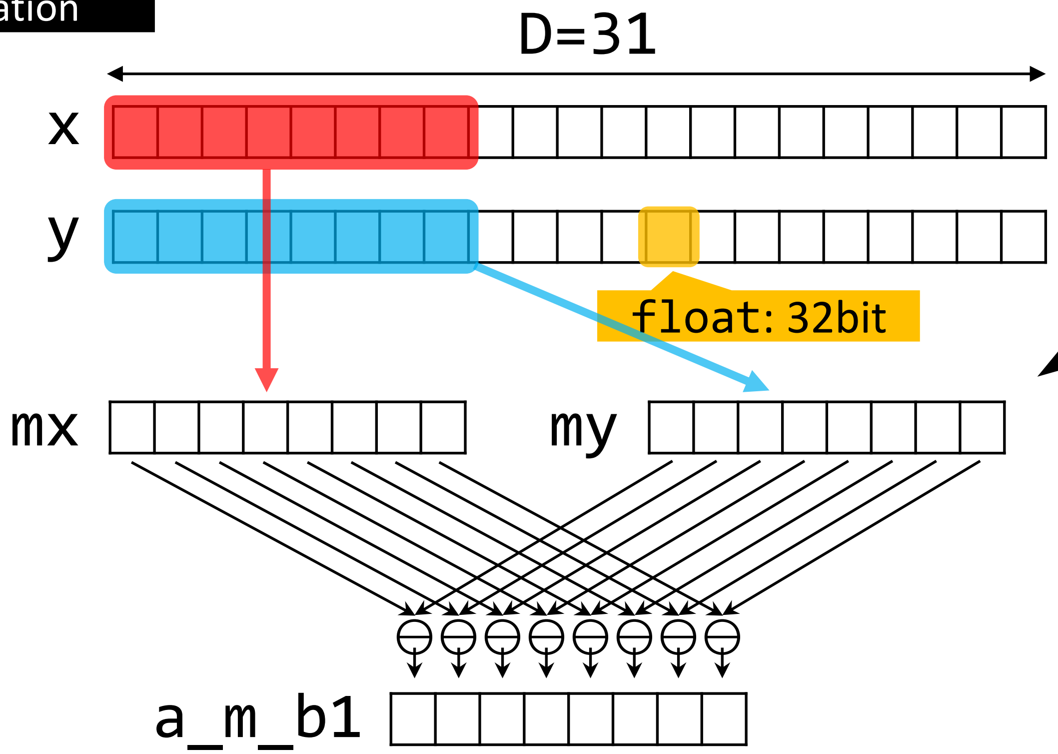
Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] – y[d])**2
  return diff
```

$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



Ref.

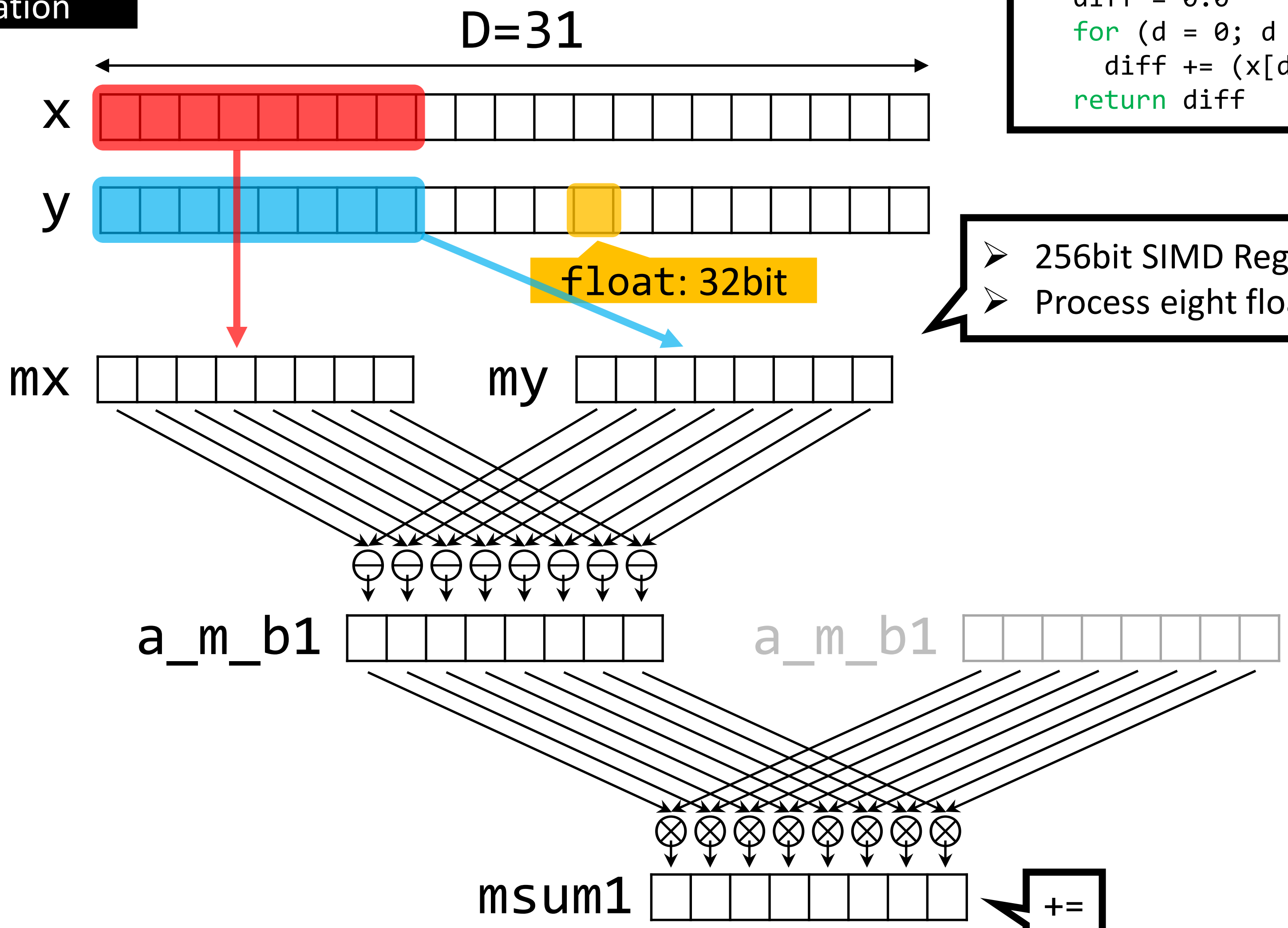
```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

- 256bit SIMD Register
- Process eight floats at once

$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



- 256bit SIMD Register
- Process eight floats at once

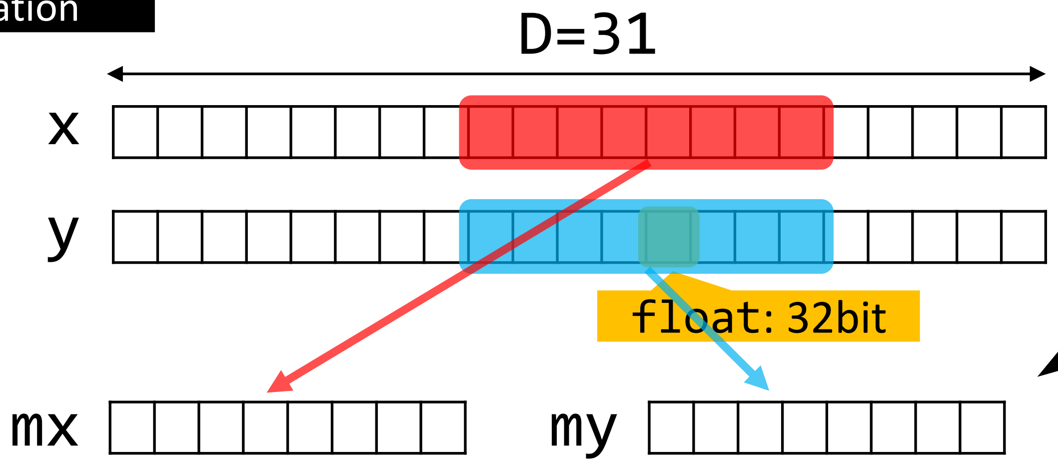
Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

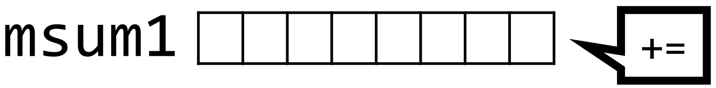
```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

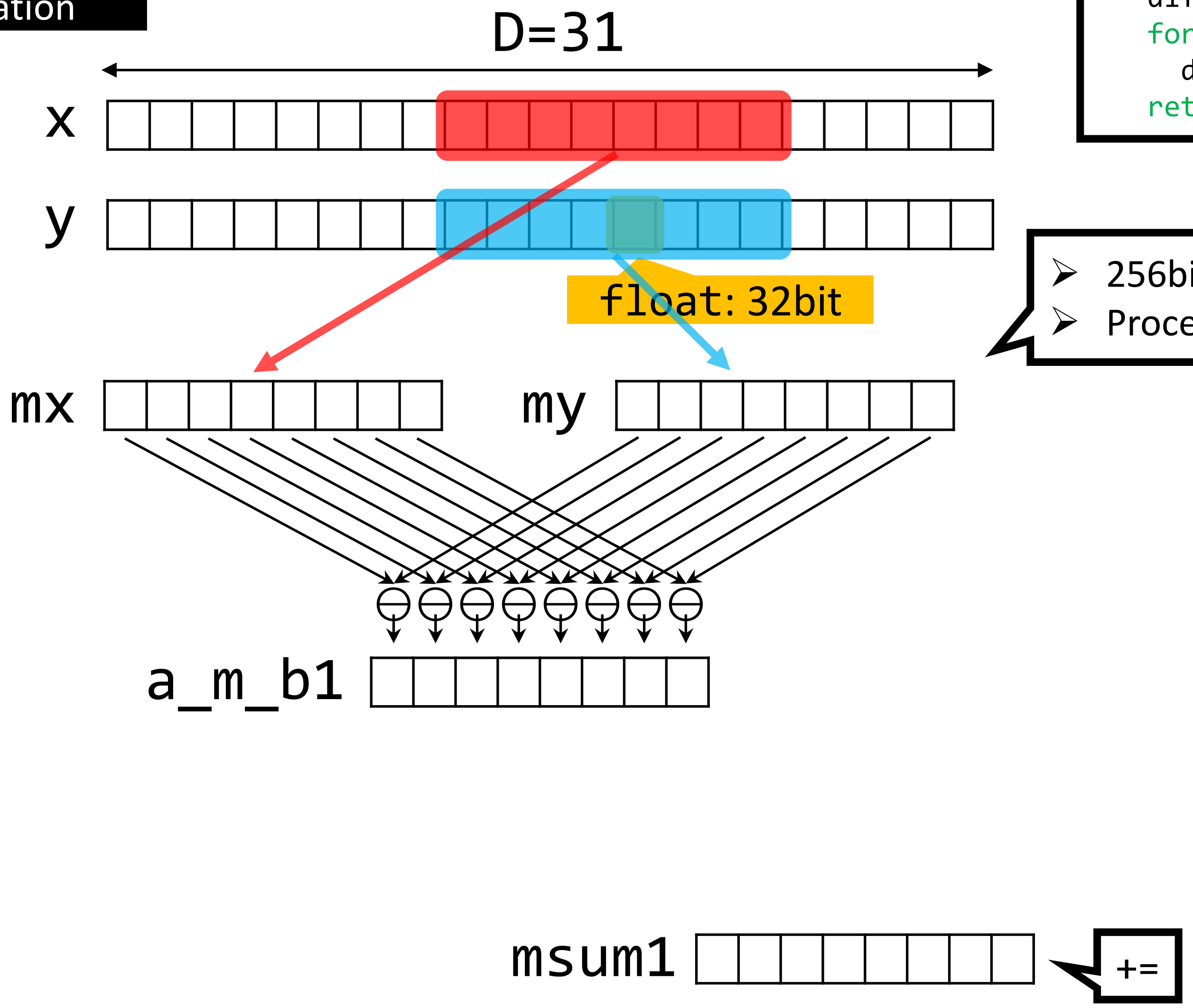
- 256bit SIMD Register
- Process eight floats at once



$\|x-y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



Ref.

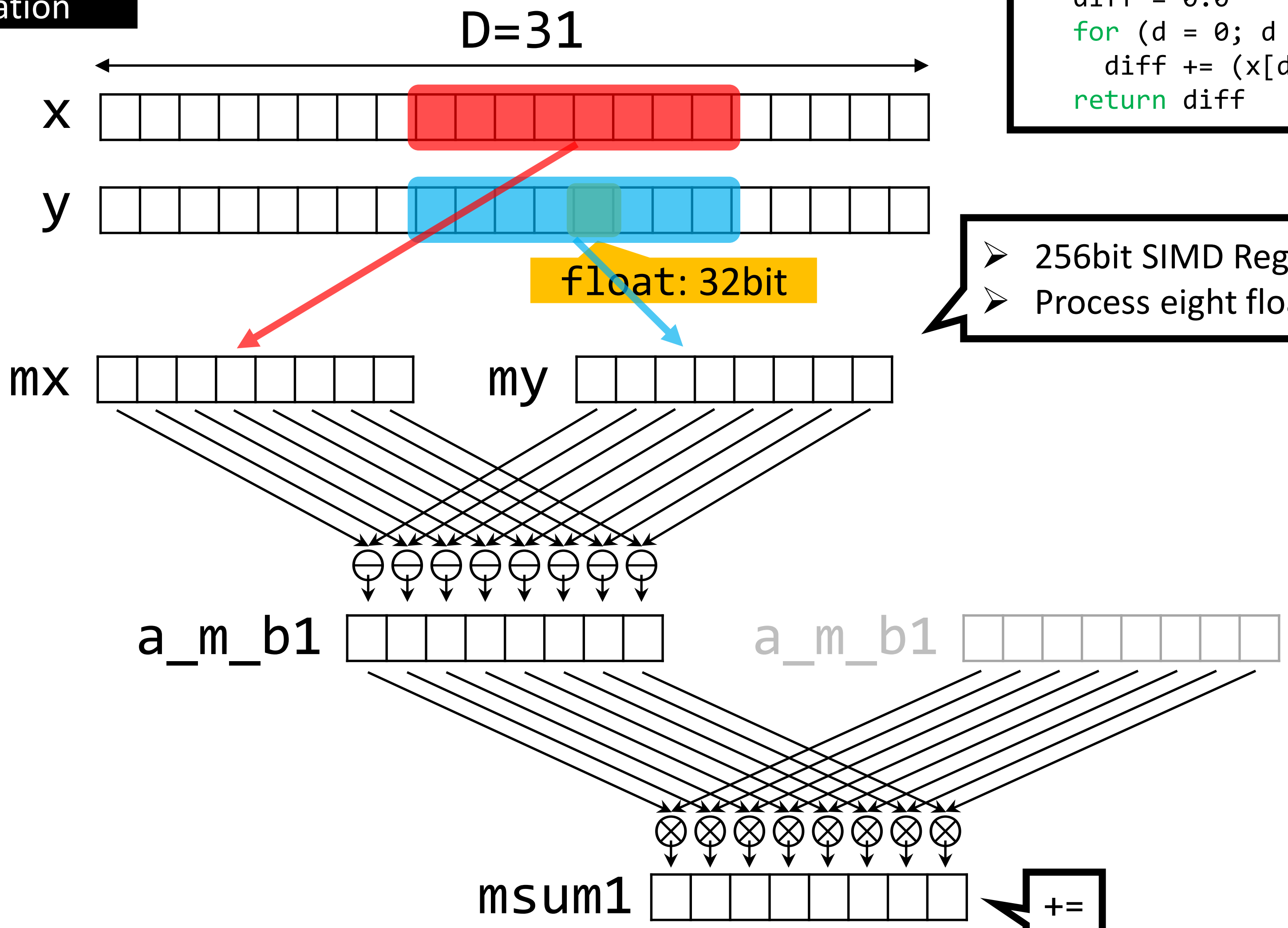
```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

- 256bit SIMD Register
- Process eight floats at once

$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

- 256bit SIMD Register
- Process eight floats at once

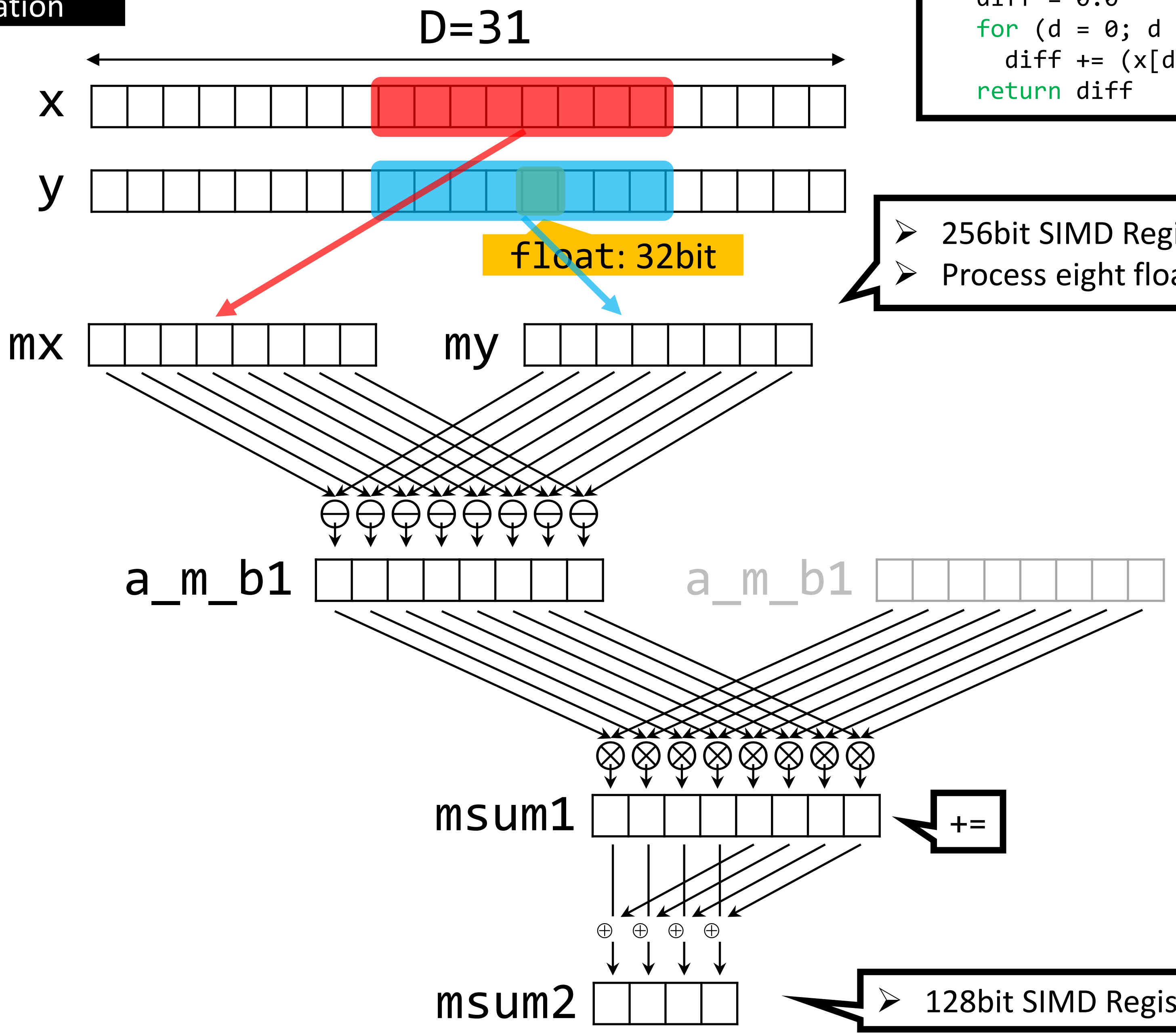
$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```

Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```



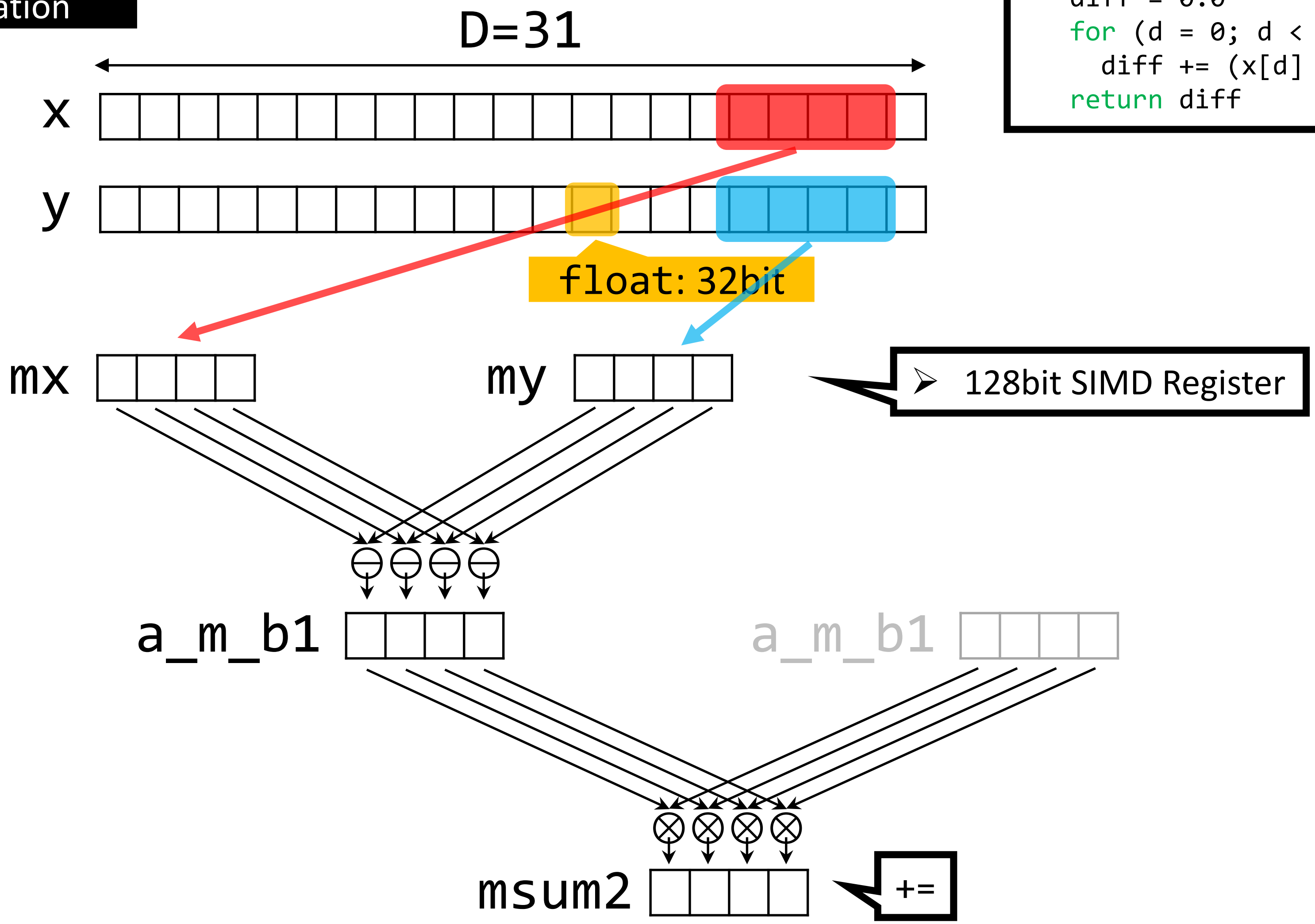
- 256bit SIMD Register
- Process eight floats at once

- 128bit SIMD Register

$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

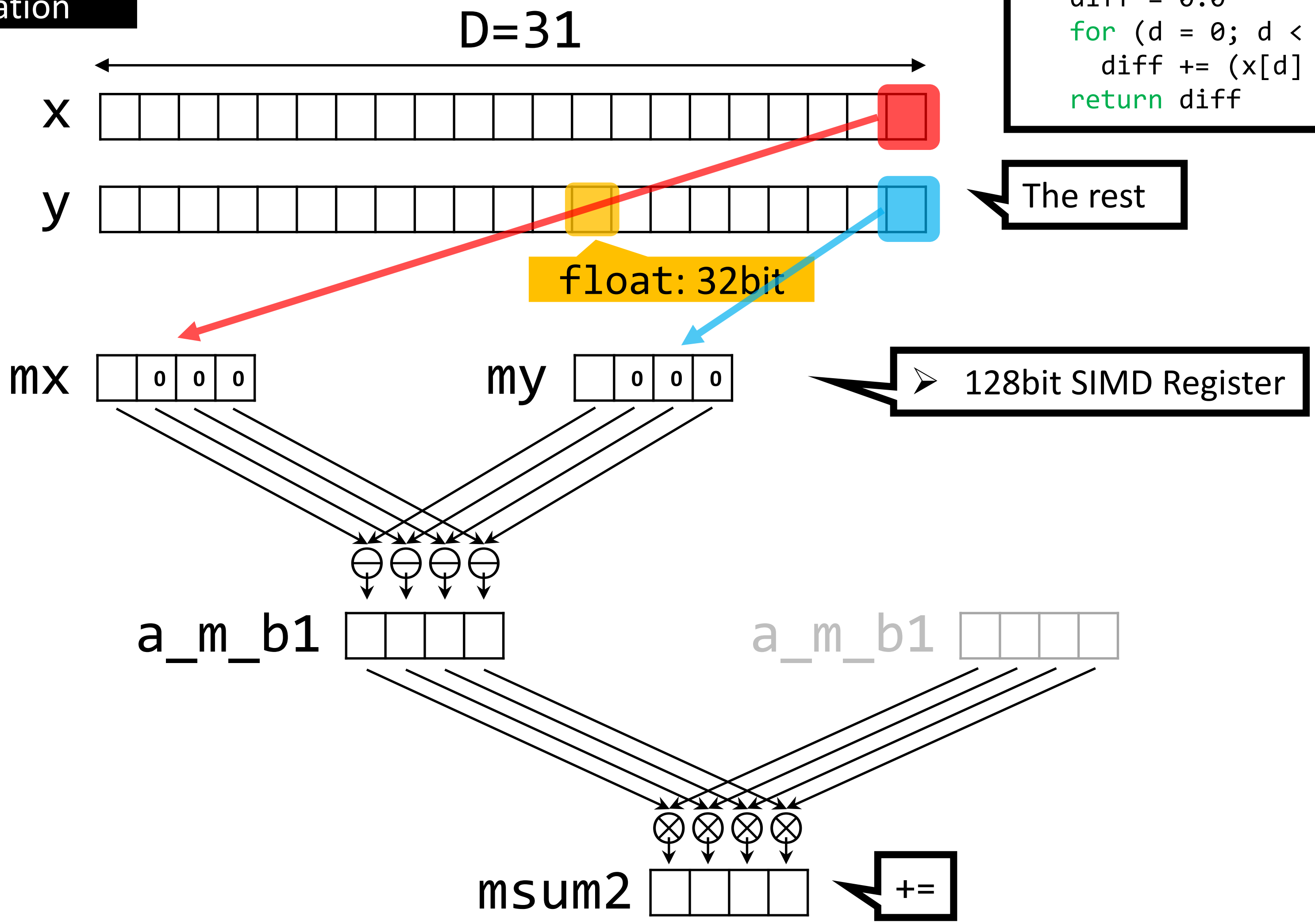
$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```

Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```



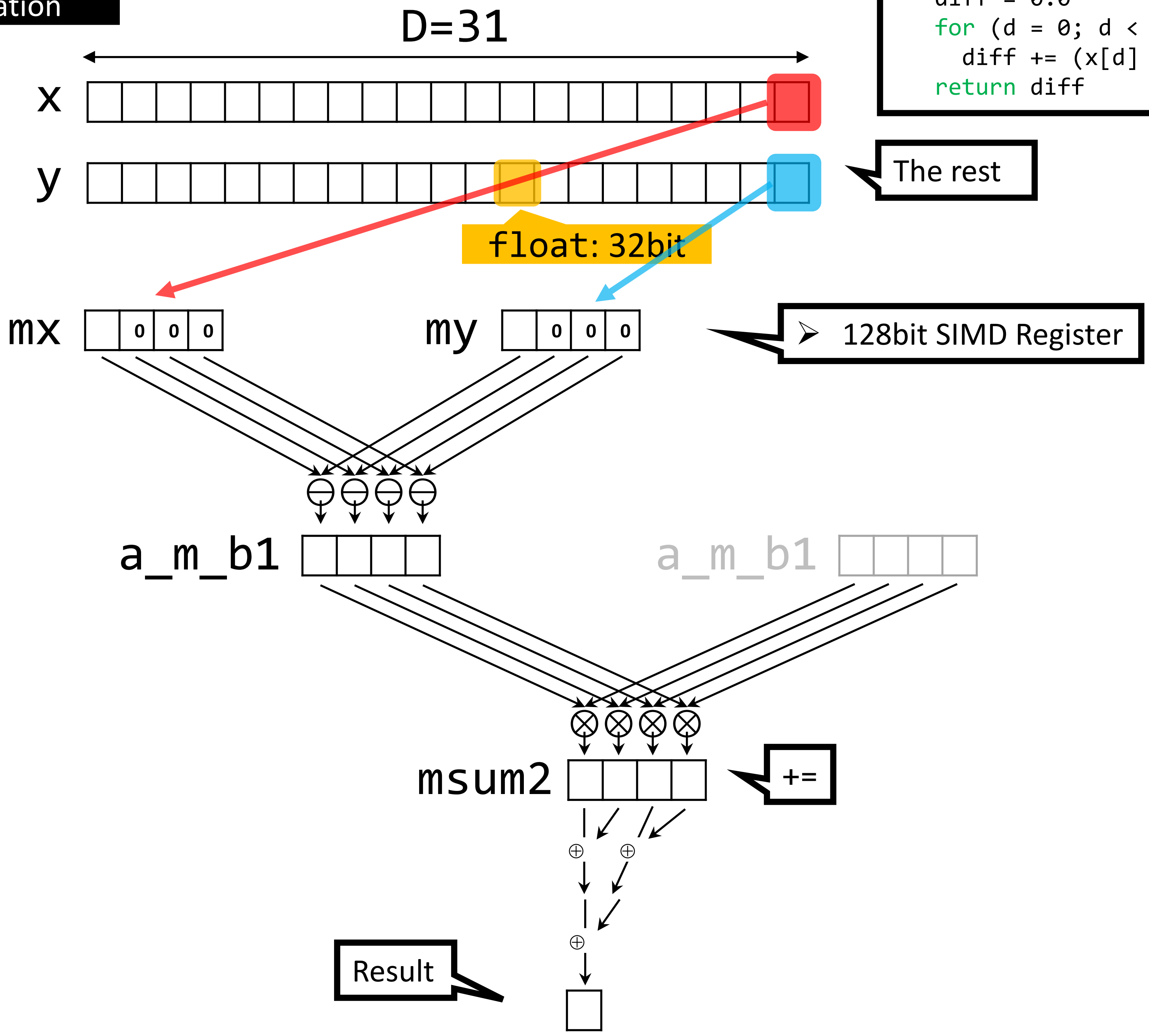
$\|x-y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();

    while (d >= 8) {
        __m256 mx = _mm256_loadu_ps (x); x += 8;
        __m256 my = _mm256_loadu_ps (y); y += 8;
        const __m256 a_m_b1 = mx - my;
        msum1 += a_m_b1 * a_m_b1;
        d -= 8;
    }

    __m128 msum2 = _mm256_extractf128_ps(msum1, 1);
    msum2 += _mm256_extractf128_ps(msum1, 0);

    if (d >= 4) {
        __m128 mx = _mm_loadu_ps (x); x += 4;
        __m128 my = _mm_loadu_ps (y); y += 4;
        const __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
        d -= 4;
    }

    if (d > 0) {
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```

Ref.

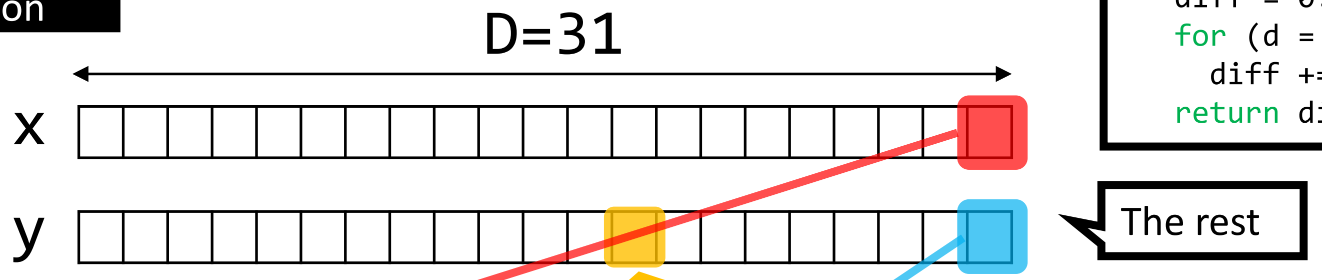
```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```



$\|x - y\|_2^2$ by SIMD

Rename variables for the sake of explanation

```
float fvec_L2sqr (const float * x,
                  const float * y,
                  size_t d)
{
    __m256 msum1 = _mm256_setzero_ps();
```

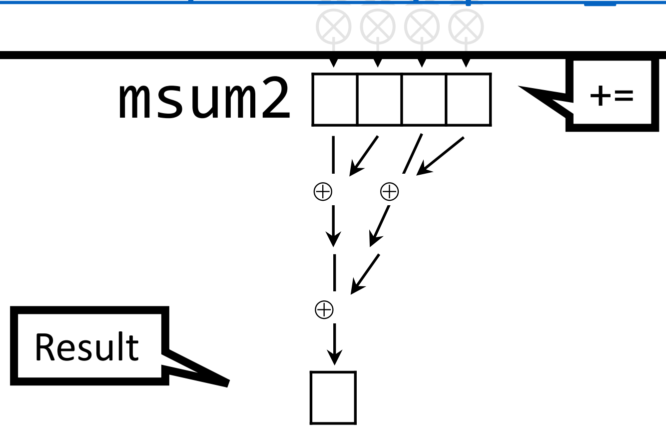


Ref.

```
def l2sqr(x, y):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (x[d] - y[d])**2
  return diff
```

- ➢ SIMD codes of faiss are simple and easy to read
- ➢ Being able to read SIMD codes comes in handy sometimes; *why this impl is super fast*
- ➢ Another example of SIMD L2sqr from HNSW: https://github.com/nmslib/hnswlib/blob/master/hnswlib/space_l2.h

```
        __m128 mx = masked_read (d, x);
        __m128 my = masked_read (d, y);
        __m128 a_m_b1 = mx - my;
        msum2 += a_m_b1 * a_m_b1;
    }

    msum2 = _mm_hadd_ps (msum2, msum2);
    msum2 = _mm_hadd_ps (msum2, msum2);
    return  _mm_cvtss_f32 (msum2);
}
```



M D-dim query vectors $\mathcal{Q} = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \dots, \boldsymbol{q}_M\}$

N D-dim database vectors $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\}$

Task：Given $\boldsymbol{q} \in \mathcal{Q}$ and $\boldsymbol{x} \in \mathcal{X}$, compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2$

Naïve impl.

```
def l2sqr(q, x):
  diff = 0.0
  for (d = 0; d < D; ++d):
    diff += (q[d] - x[d])**2
  return diff
```

```
parfor q in Q:
  for x in X:
    l2sqr(q, x)
```

Parallelize query-side

Select min by heap, but omit it now

faiss impl.

if $M < 20$:

compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2$ by SIMD

else :

compute $\|\boldsymbol{q} - \boldsymbol{x}\|_2^2 = \|\boldsymbol{q}\|_2^2 - 2\boldsymbol{q}^\top \boldsymbol{x} + \|\boldsymbol{x}\|_2^2$ by BLAS

Compute $\|\boldsymbol{q}-\boldsymbol{x}\|_2^2 = \|\boldsymbol{q}\|_2^2 - 2\boldsymbol{q}^\top\boldsymbol{x} + \|\boldsymbol{x}\|_2^2$ with BLAS

Stack M D-dim query vectors to a $D \times M$ matrix: $Q = [\boldsymbol{q}_1, \boldsymbol{q}_2, \dots, \boldsymbol{q}_M] \in \mathbb{R}^{D\times M}$

Stack N D-dim database vectors to a $D \times N$ matrix: $X = [\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N] \in \mathbb{R}^{D\times N}$

SIMD-accelerated function

```
# Compute tables
q_norms = norms(Q)        # ||q_1||_2^2, ||q_2||_2^2, ..., ||q_M||_2^2
x_norms = norms(X)        # ||x_1||_2^2, ||x_2||_2^2, ..., ||x_N||_2^2
ip = sgemm_(Q, X, …)      # Q^T X

# Scan and sum
parfor (m = 0; m < M; ++m):
  for (n = 0; n < N; ++n):
    dist = q_norms[m] + x_norms[n] - ip[m][n]
```

- Matrix multiplication by BLAS
- Dominant if Q and X are large
- The difference of the background matters:
  - ✓ Intel MKL is 30% faster than OpenBLAS

$\|\boldsymbol{q}_m - \boldsymbol{x}_n\|_2^2$ $\|\boldsymbol{q}_m\|_2^2$ $\|\boldsymbol{x}_n\|_2^2$ $(Q^\top X)_{mn}$

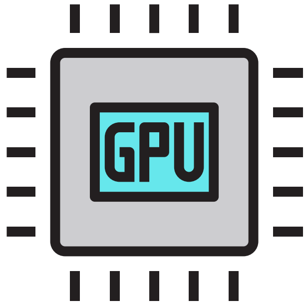
NN in GPU (faiss-gpu) is 10x faster than NN in CPU (faiss-cpu)

Benchmark: https://github.com/facebookresearch/faiss/wiki/Low-level-benchmarks

- NN-GPU always compute $\|\boldsymbol{q}\|_2^2 - 2\boldsymbol{q}^\top\boldsymbol{x} + \|\boldsymbol{x}\|_2^2$

- k-means for 1M vectors (D=256, K=20000)
  - ✓ 11 min on CPU
  - ✓ 55 sec on 1 Pascal-class P100 GPU (float32 math)
  - ✓ 34 sec on 1 Pascal-class P100 GPU (float16 math)
  - ✓ 21 sec on 4 Pascal-class P100 GPUs (float32 math)
  - ✓ 16 sec on 4 Pascal-class P100 GPUs (float16 math)

x10 faster



- If GPU is available and its memory is enough, try GPU-NN
- The behavior is little bit different (e.g., a restriction for top-k)

Reference

- Switch implementation of L2sqr in faiss: [https://github.com/facebookresearch/faiss/wiki/Implementation-notes#matrix-multiplication-to-do-many-l2-distance-computations]

- Introduction to SIMD: a lecture by Markus Püschel (ETH) [How to Write Fast Numerical Code - Spring 2019], especially [SIMD vector instructions]
  - ✓ https://acl.inf.ethz.ch/teaching/fastcode/2019/
  - ✓ https://acl.inf.ethz.ch/teaching/fastcode/2019/slides/07-simd.pdf

- SIMD codes for faiss [https://github.com/facebookresearch/faiss/blob/master/utils/distances_simd.cpp]

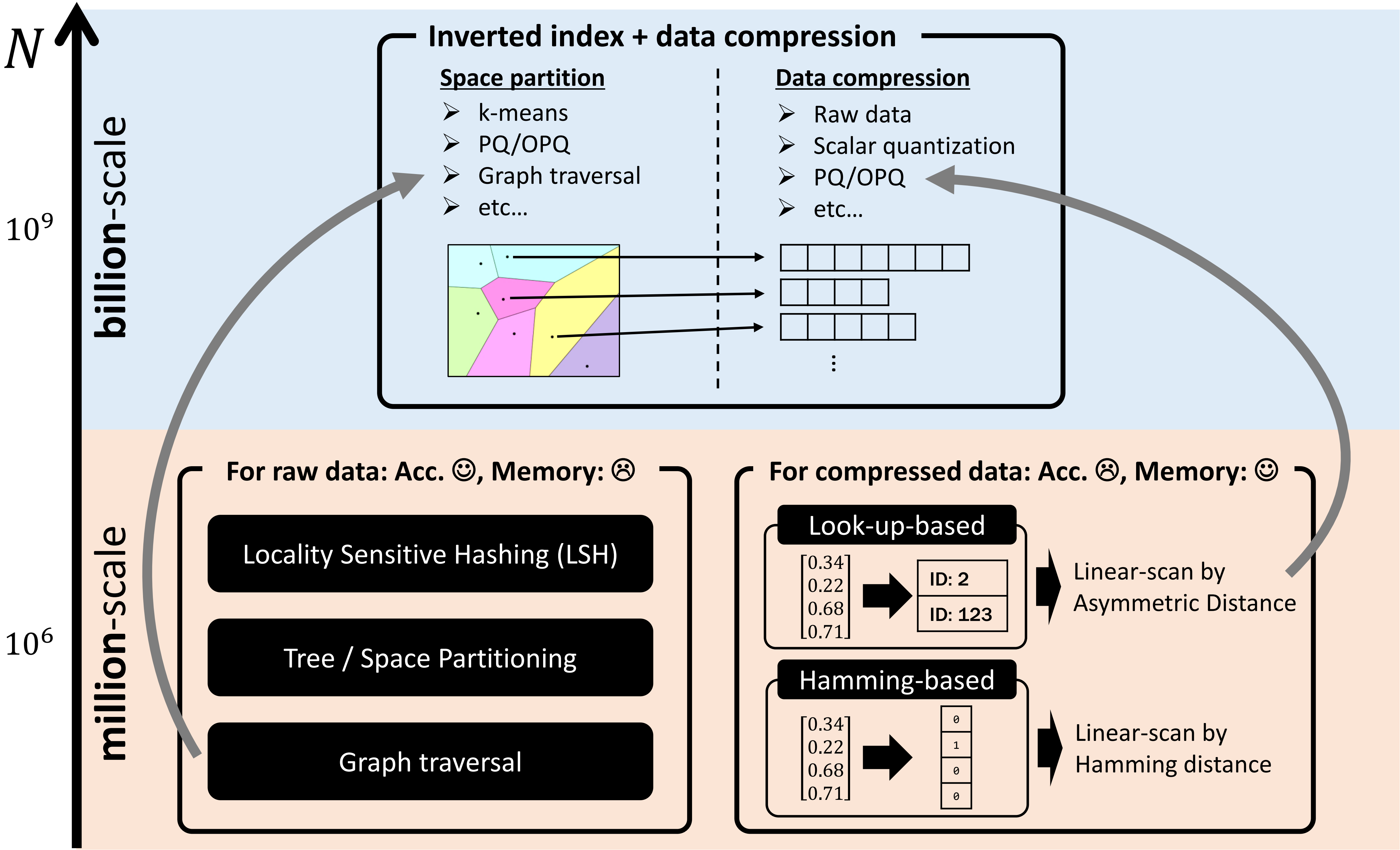
- L2sqr benchmark including AVX512 for faiss-L2sqr [https://gist.github.com/matsui528/583925f88fcb08240319030202588c74]

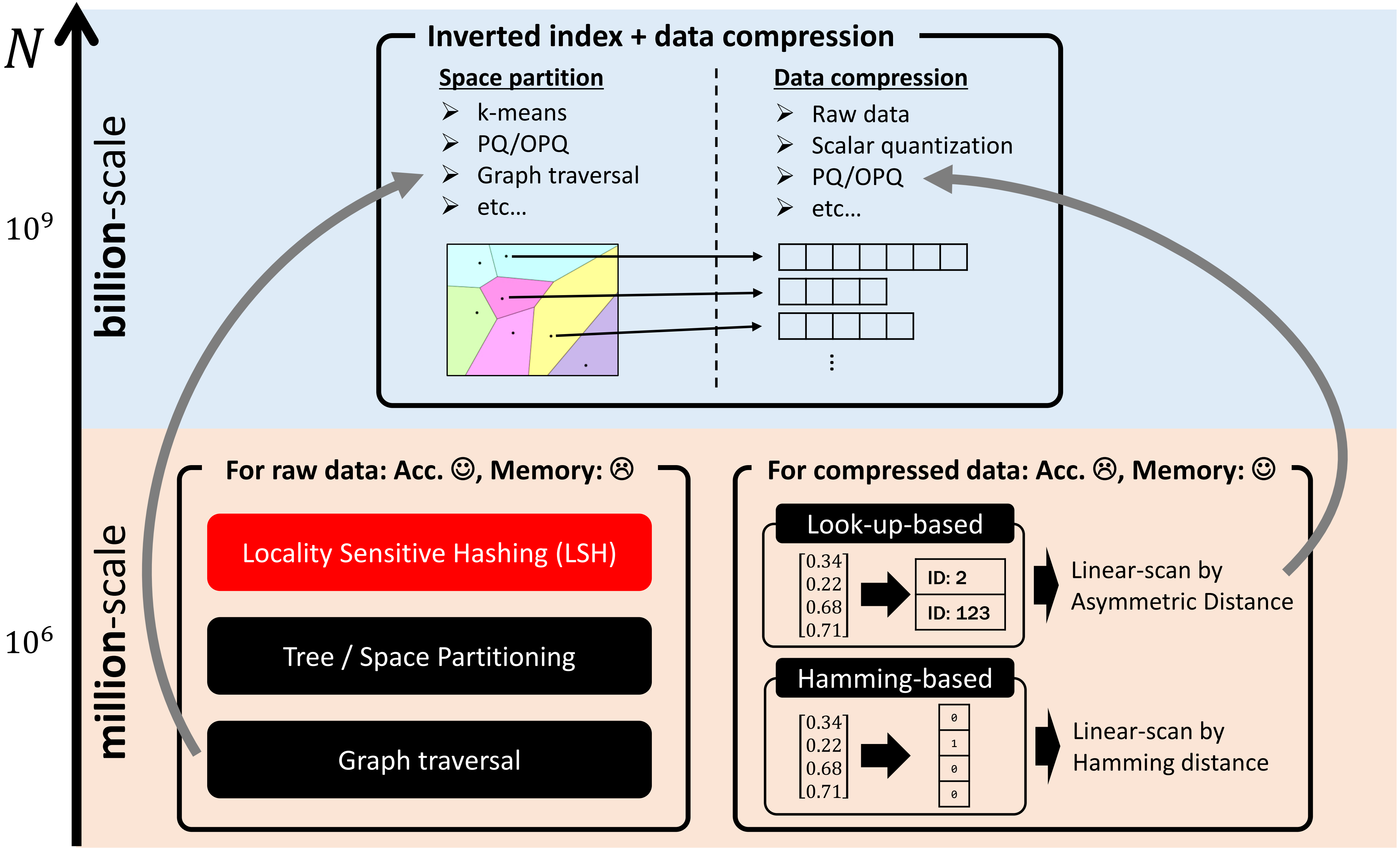
Part 1:

Nearest Neighbor Search

Part 2:

Approximate Nearest Neighbor Search

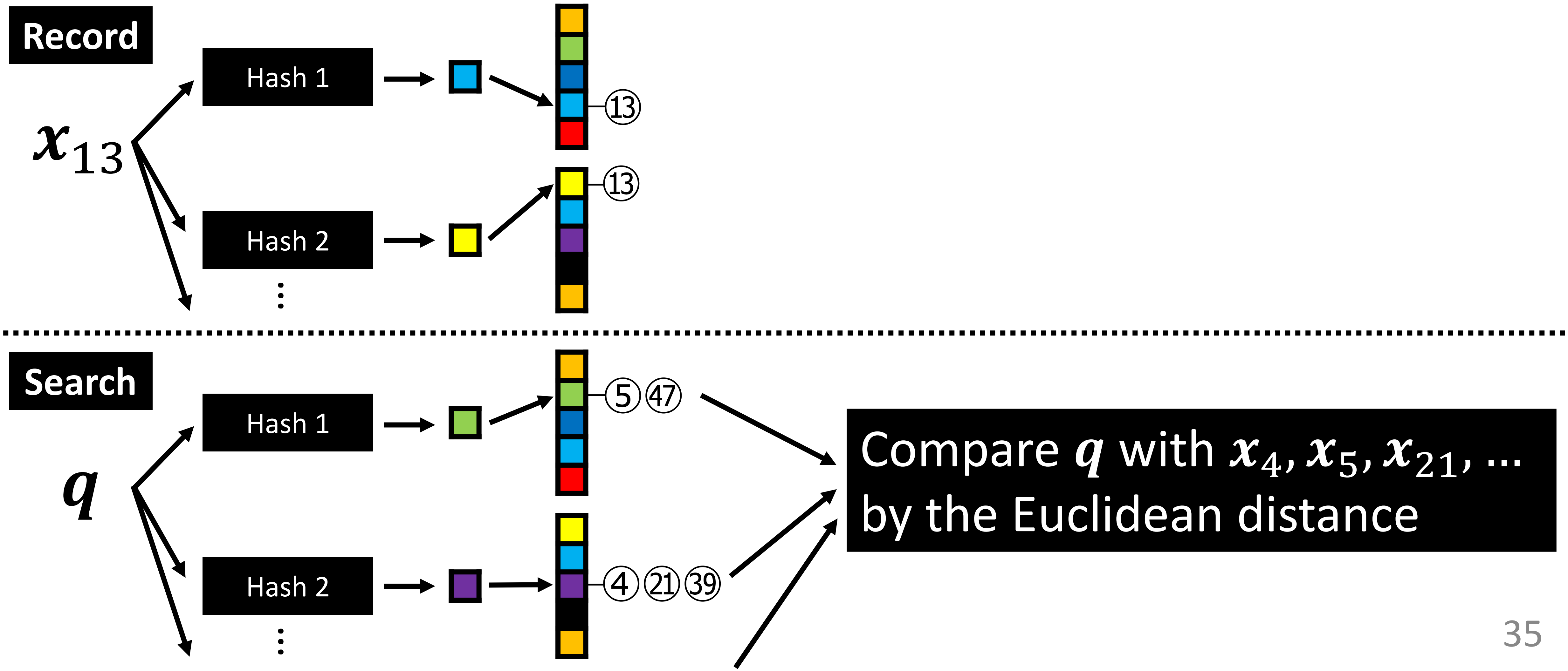

N
10^9
10^6
billion-scale
million-scale
Inverted index + data compression
Space partition
k-means
PQ/OPQ
Graph traversal
etc…
Data compression
Raw data
Scalar quantization
PQ/OPQ
etc…
For raw data: Acc. ☺, Memory: ☹
Locality Sensitive Hashing (LSH)
Tree / Space Partitioning
Graph traversal
For compressed data: Acc. ☹, Memory: ☺
Look-up-based
0.34
0.22
0.68
0.71
ID: 2
ID: 123
Linear-scan by
Asymmetric Distance
Hamming-based
0.34
0.22
0.68
0.71
0
1
0
0
Linear-scan by
Hamming distance



N
10^9
billion-scale
10^6
million-scale
Inverted index + data compression
Space partition
k-means
PQ/OPQ
Graph traversal
etc…
Data compression
Raw data
Scalar quantization
PQ/OPQ
etc…
For raw data: Acc. ☺, Memory: ☹
Locality Sensitive Hashing (LSH)
Tree / Space Partitioning
Graph traversal
For compressed data: Acc. ☹, Memory: ☺
Look-up-based
[0.34 0.22 0.68 0.71]
ID: 2
ID: 123
Linear-scan by Asymmetric Distance
Hamming-based
[0.34 0.22 0.68 0.71]
0 1 0 0
Linear-scan by Hamming distance

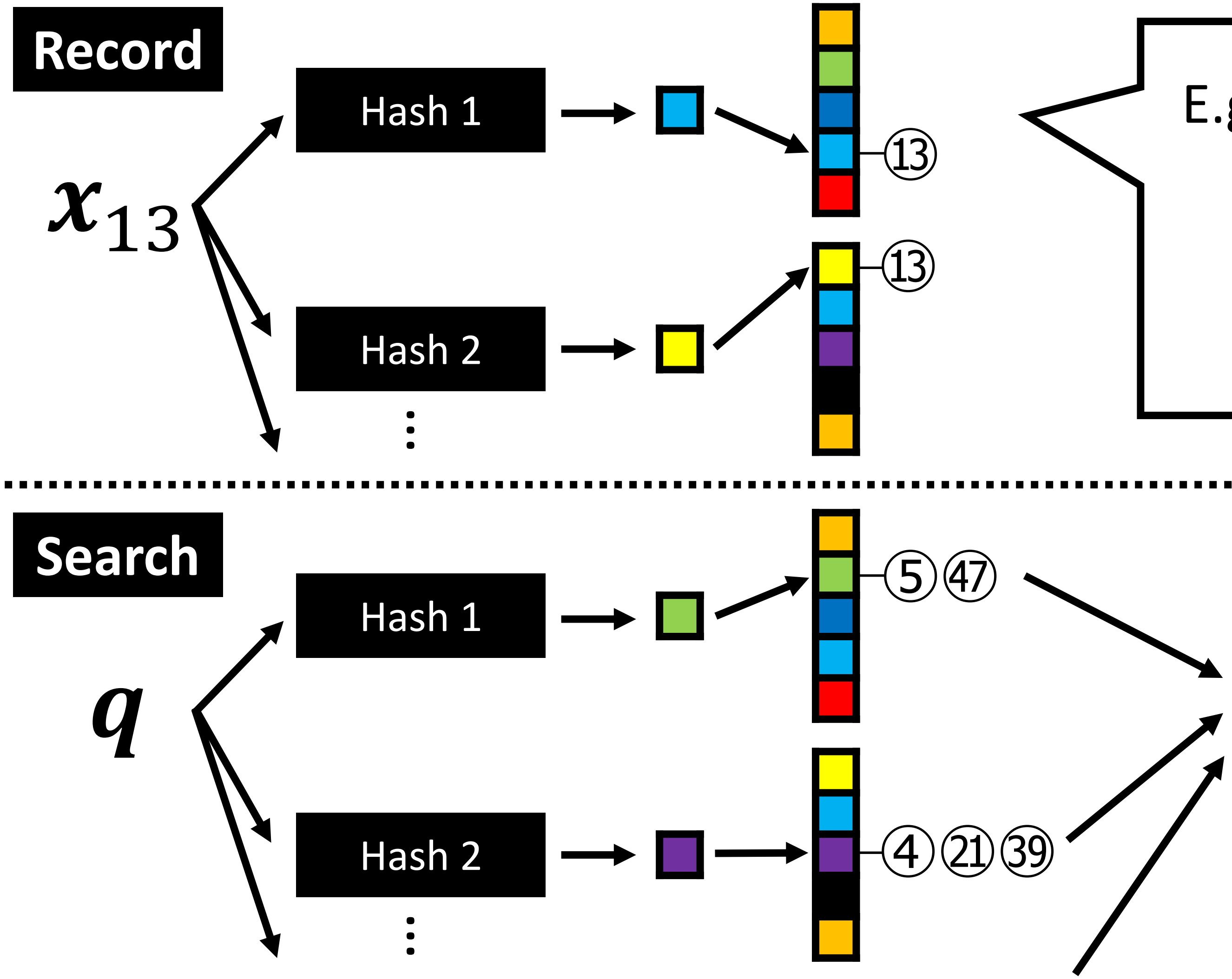
Locality Sensitive Hashing (LSH)

- LSH = Hash functions + Hash tables
- Map similar items to the same symbol with a high probability



Locality Sensitive Hashing (LSH)

- LSH = Hash functions + Hash tables
- Map similar items to the same symbol with a high probability



E.g., random projection [Dater+, SCG 04]

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

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

Compare $\boldsymbol{q}$ with $\boldsymbol{x}_4, \boldsymbol{x}_5, \boldsymbol{x}_{21}, \ldots$ by the Euclidean distance

Locality Sensitive Hashing (LSH)

➢ LSH = Hash functions + Hash tables

☺:

➢ Math-friendly

➢ Popular in the theory area (FOCS, STOC, …)

☹:

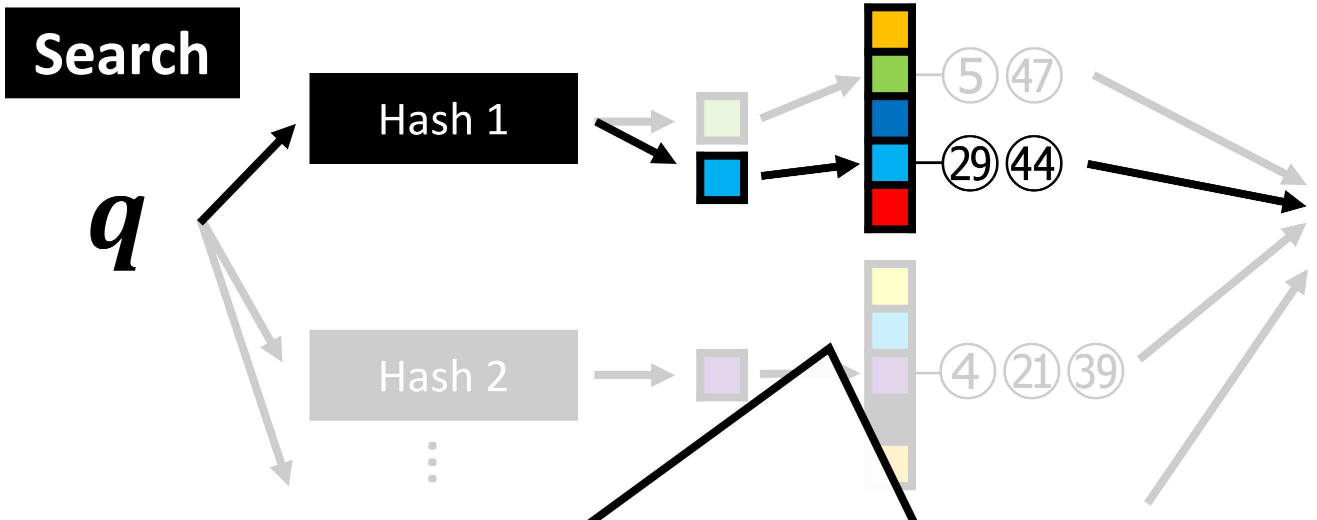
➢ Large memory cost

  ✓ Need several tables to boost the accuracy

  ✓ Need to store the original data, $\{\boldsymbol{x}_n\}_{n=1}^{N}$, on memory

➢ Data-dependent methods such as PQ are better for real-world data

➢ Thus, in recent CV papers, LSH has been treated as a classic-method ☹☹☹



Compare $\boldsymbol{q}$ with $\boldsymbol{x}_4, \boldsymbol{x}_5, \boldsymbol{x}_{21}, \ldots$ by the Euclidean distance

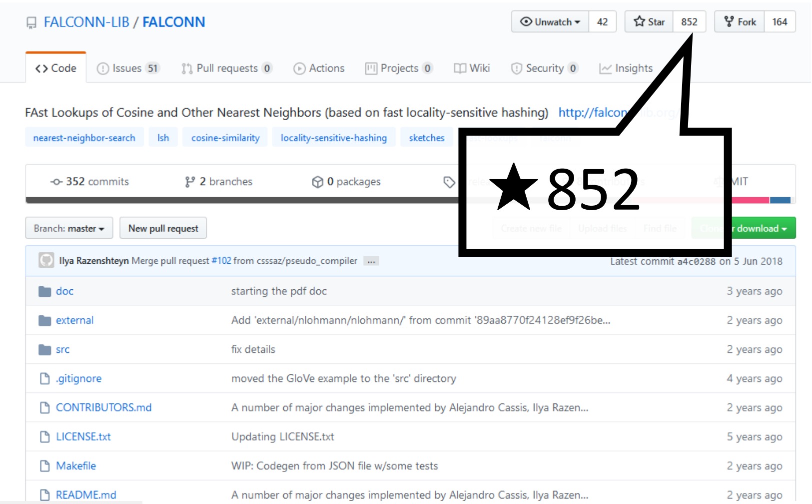
In fact:

- ➢ Consider a next candidate ➡ practical memory consumption （Multi-Probe [Lv+, VLDB 07]）
- ➢ A library based on the idea: **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)
```

☺ Faster data addition (than annoy, nmslib, ivfpq)

☺ Useful for on-the-fly addition

☹ Parameter configuration seems a bit non-intuitive

Reference

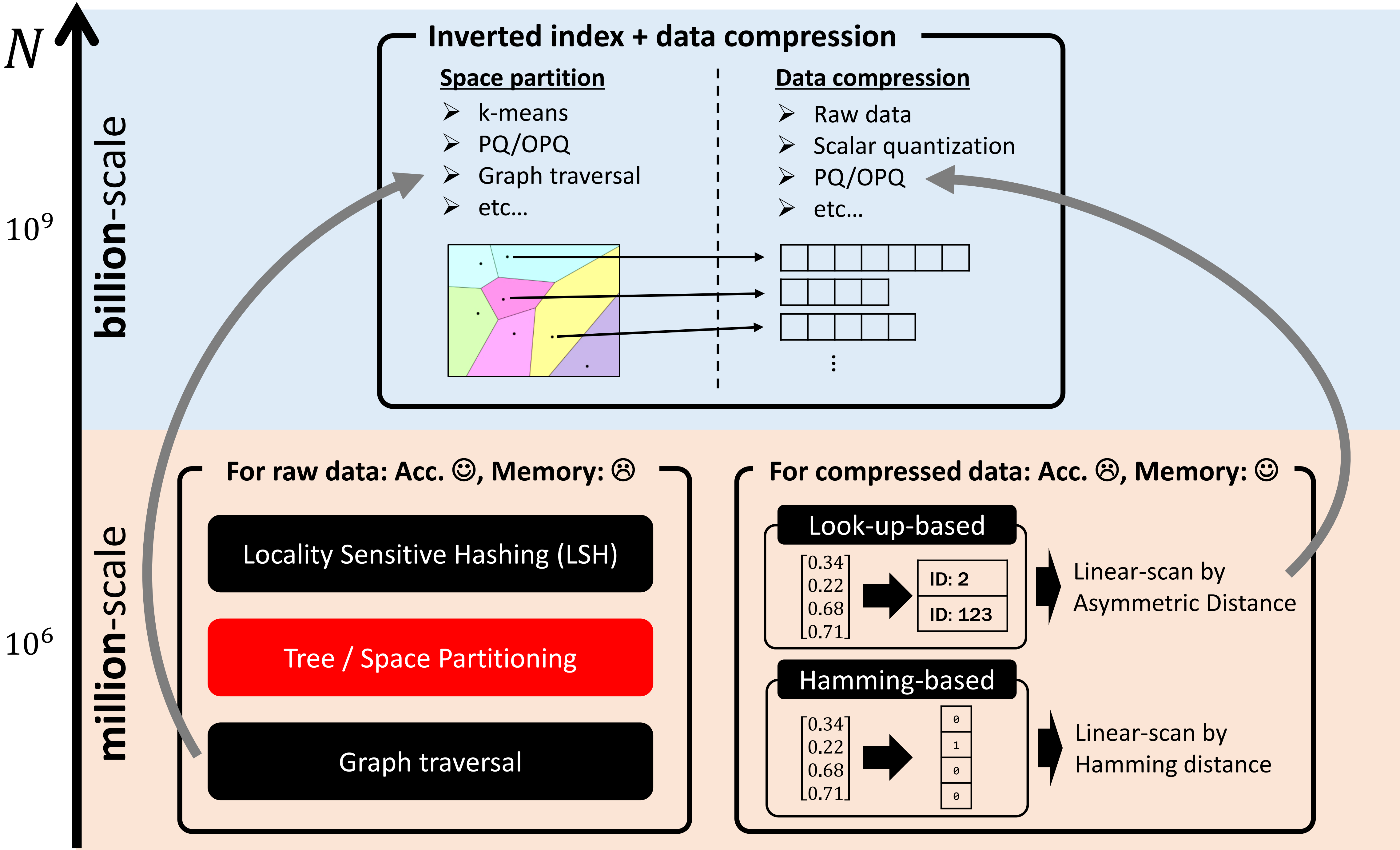
- Good summaries on this field: CVPR 2014 Tutorial on Large-Scale Visual Recognition, Part I: Efficient matching, H. Jégou [https://sites.google.com/site/lsvrtutorialcvpr14/home/efficient-matching]

- Practical Q&A: FAQ in Wiki of FALCONN [https://github.com/FALCONN-LIB/FALCONN/wiki/FAQ]

- Hash functions: 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

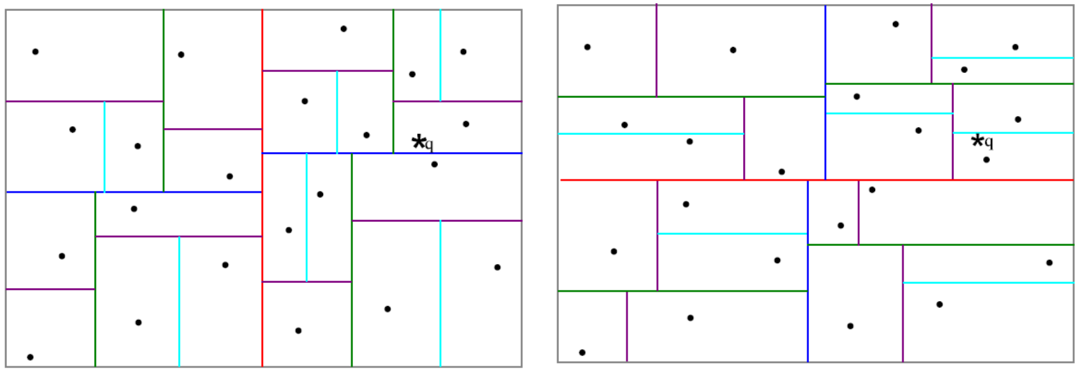
- Survey: A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," Comm. ACM 2008



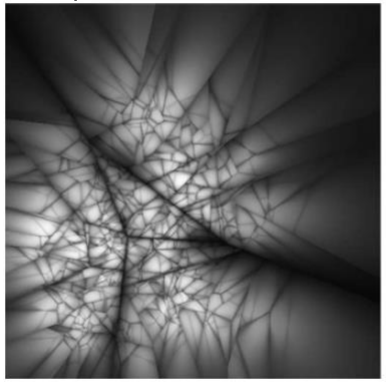
N
billion-scale
10^9
million-scale
10^6
Inverted index + data compression
Space partition
k-means
PQ/OPQ
Graph traversal
etc…
Data compression
Raw data
Scalar quantization
PQ/OPQ
etc…
For raw data: Acc. ☺, Memory: ☹
Locality Sensitive Hashing (LSH)
Tree / Space Partitioning
Graph traversal
For compressed data: Acc. ☹, Memory: ☺
Look-up-based
[0.34 0.22 0.68 0.71]
ID: 2
ID: 123
Linear-scan by Asymmetric Distance
Hamming-based
[0.34 0.22 0.68 0.71]
0 1 0 0
Linear-scan by Hamming distance

FLANN: Fast Library for Approximate Nearest Neighbors

Images are from [Muja and Lowe, TPAMI 2014]



Randomized KD Tree



k-means Tree

➢ Automatically select "Randomized KD Tree" or "k-means Tree"
https://github.com/mariusmuja/flann

☺ Good code base. Implemented in OpenCV and PCL

☺ Very popular in the late 00's and early 10's

☹ Large memory consumption. The original data need to be stored

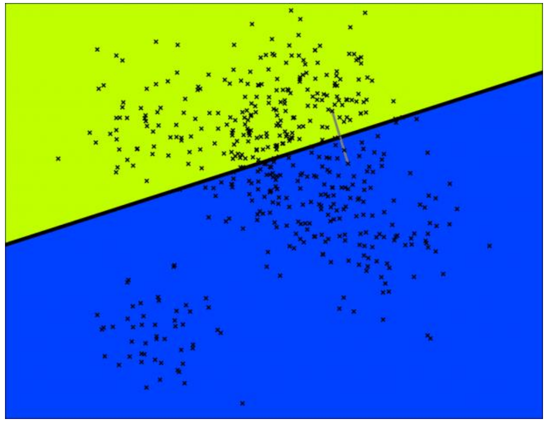
☹ Not actively maintained now

Annoy

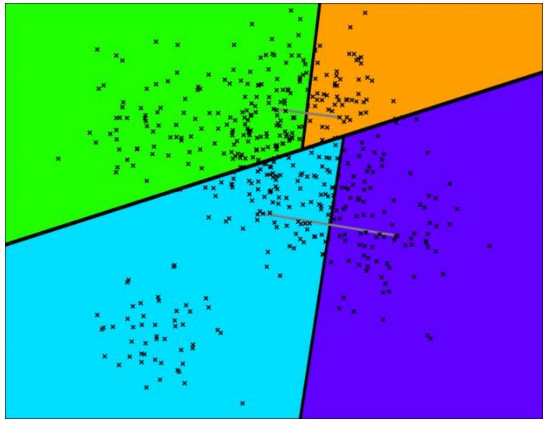
All images are cited from the author's blog post (https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html)

"2-means tree"+ "multiple-trees" + "shared priority queue"

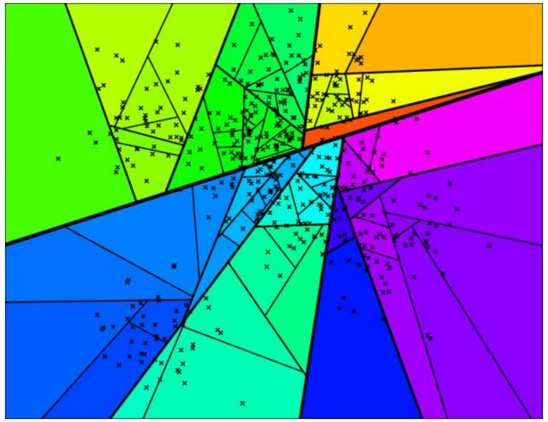
Record



Select two points randomly

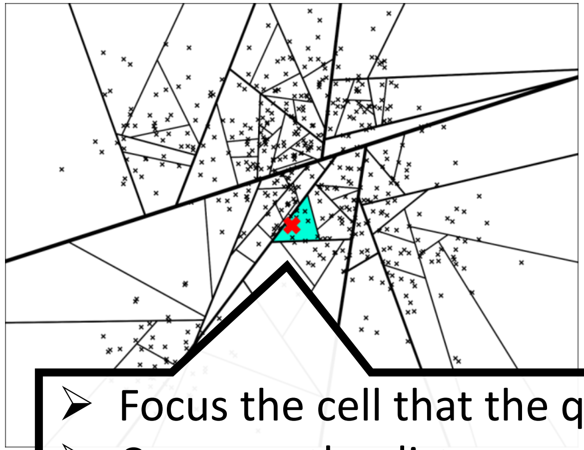


Divide up the space

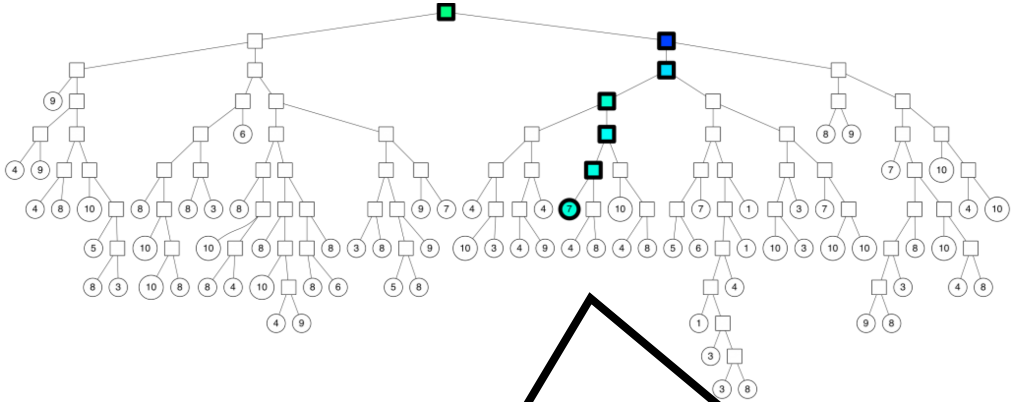


Repeat hierarchically

Search



- Focus the cell that the query lives
- Compare the distances



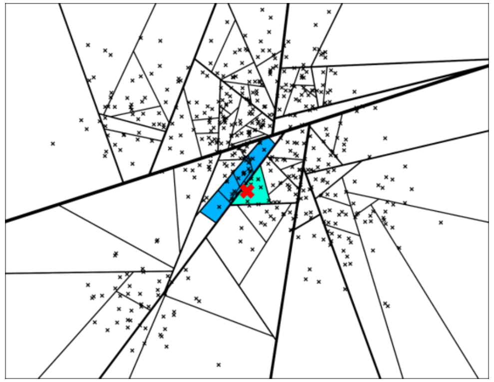
Can traverse the tree by log-times comparisons

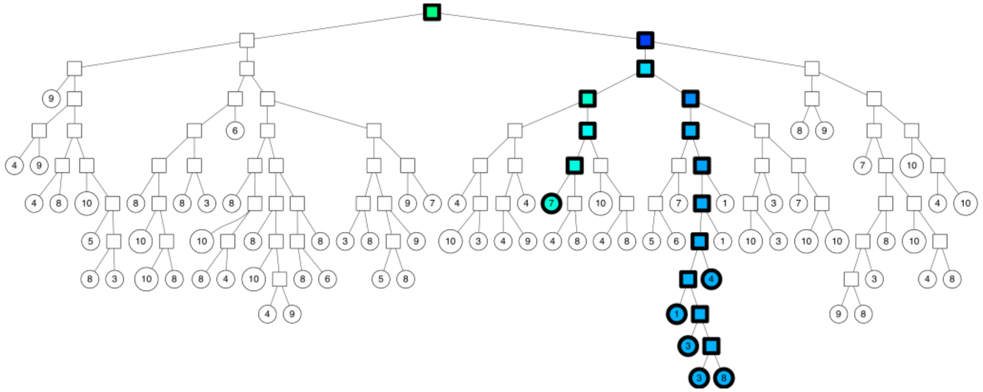
Annoy

All images are cited from the author's blog post (https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html)

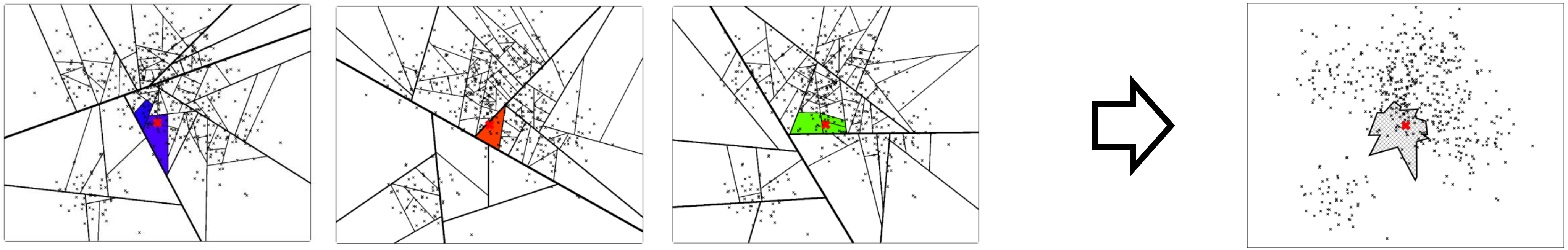
"2-means tree"+ "multiple-trees" + "shared priority queue"

Feature 1 If we need more data points, use a priority queue





Feature 2 Boost the accuracy by multi-tree with a shared priority queue

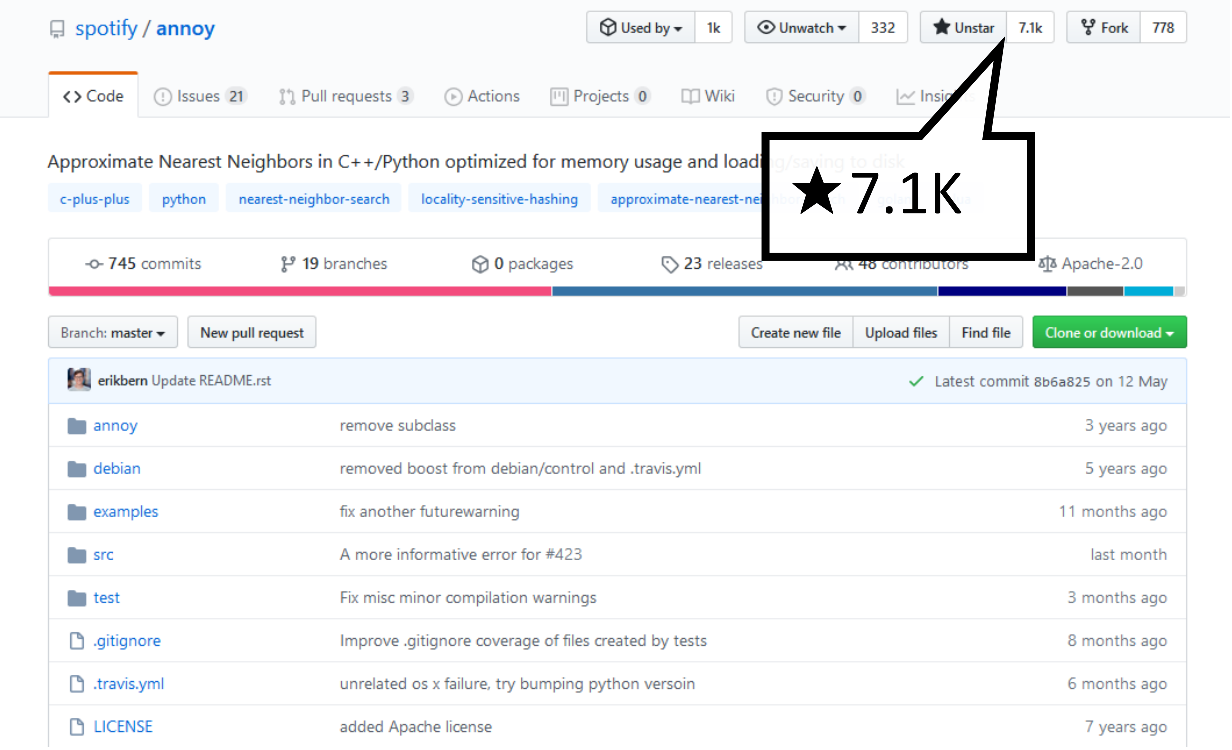


Annoy

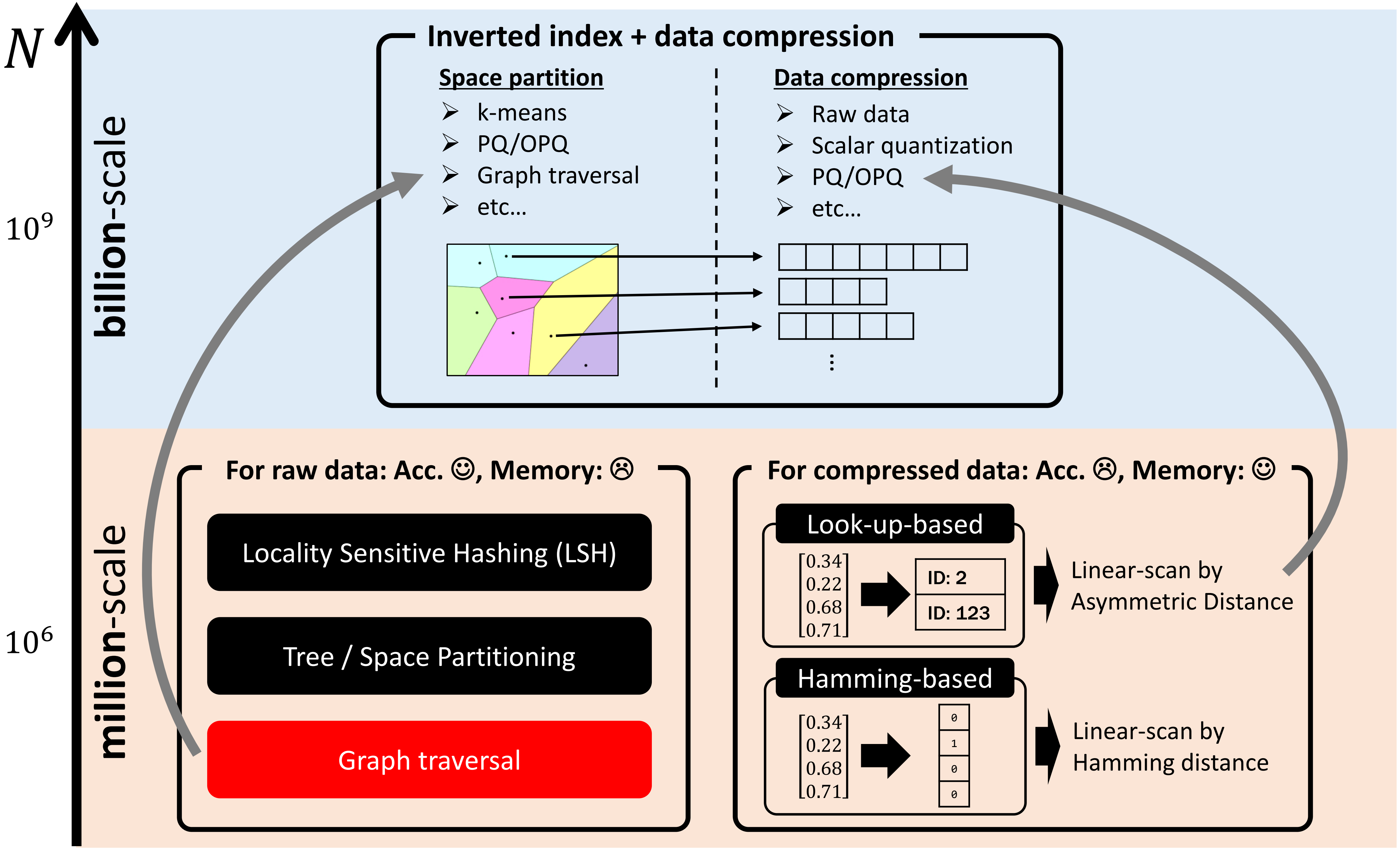
https://github.com/erikbern/annoy

```
$> 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)
```



- ☺ Developed at Spotify. Well-maintained. Stable
- ☺ Simple interface with only a few parameters
- ☺ Baseline for million-scale data
- ☺ Support mmap, i.e., can be accessed from several processes
- ☹ Large memory consumption
- ☹ Runtime itself is slower than HNSW


N
10^9
10^6
billion-scale
million-scale
Inverted index + data compression
Space partition
k-means
PQ/OPQ
Graph traversal
etc…
Data compression
Raw data
Scalar quantization
PQ/OPQ
etc…
For raw data: Acc. ☺, Memory: ☹
Locality Sensitive Hashing (LSH)
Tree / Space Partitioning
Graph traversal
For compressed data: Acc. ☹, Memory: ☺
Look-up-based
0.34
0.22
0.68
0.71
ID: 2
ID: 123
Linear-scan by
Asymmetric Distance
Hamming-based
0.34
0.22
0.68
0.71
0
1
0
0
Linear-scan by
Hamming distance

Graph traversal

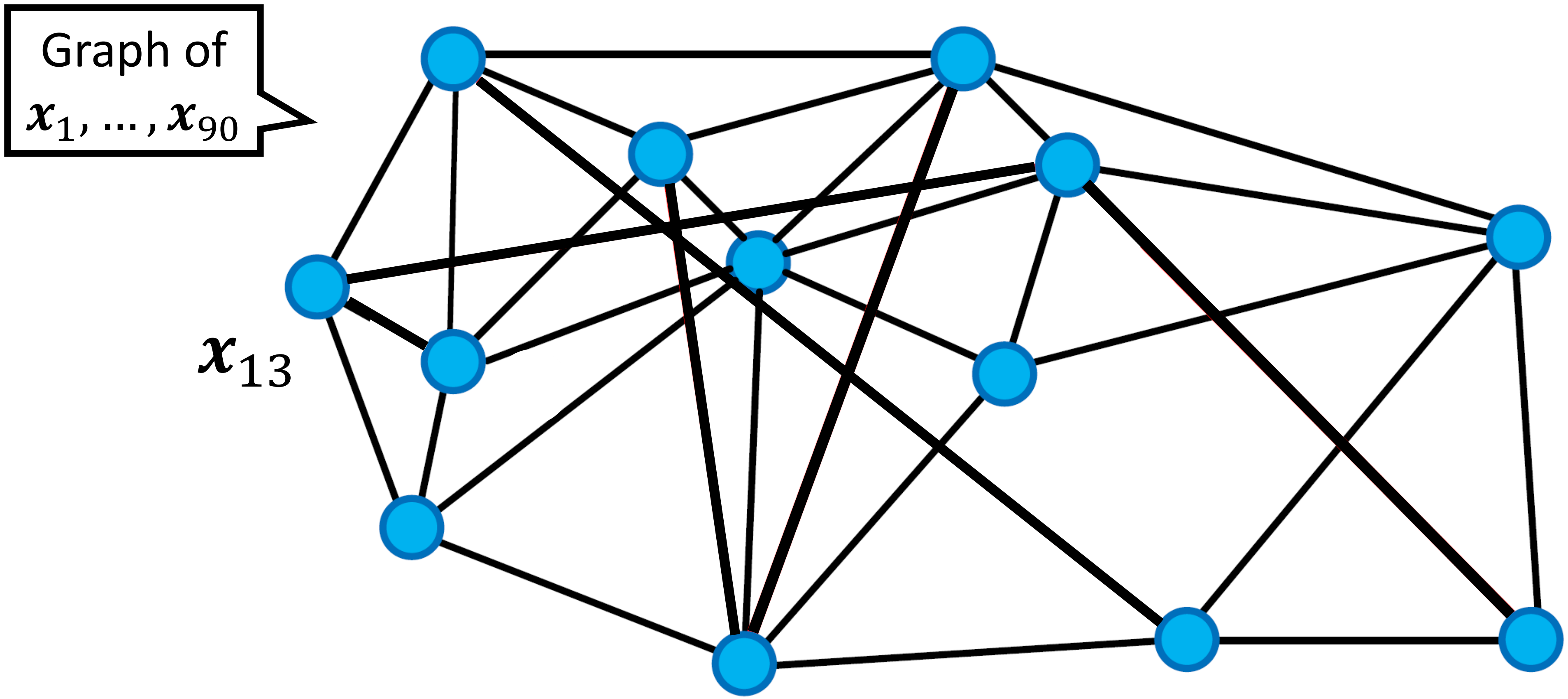
- **Very popular** in recent years

- Around 2017, it turned out that the graph-traversal-based methods work well for million-scale data

- Pioneer:
  - ✓ Navigable Small World Graphs (NSW)
  - ✓ Hierarchical NSW (**HNSW**)

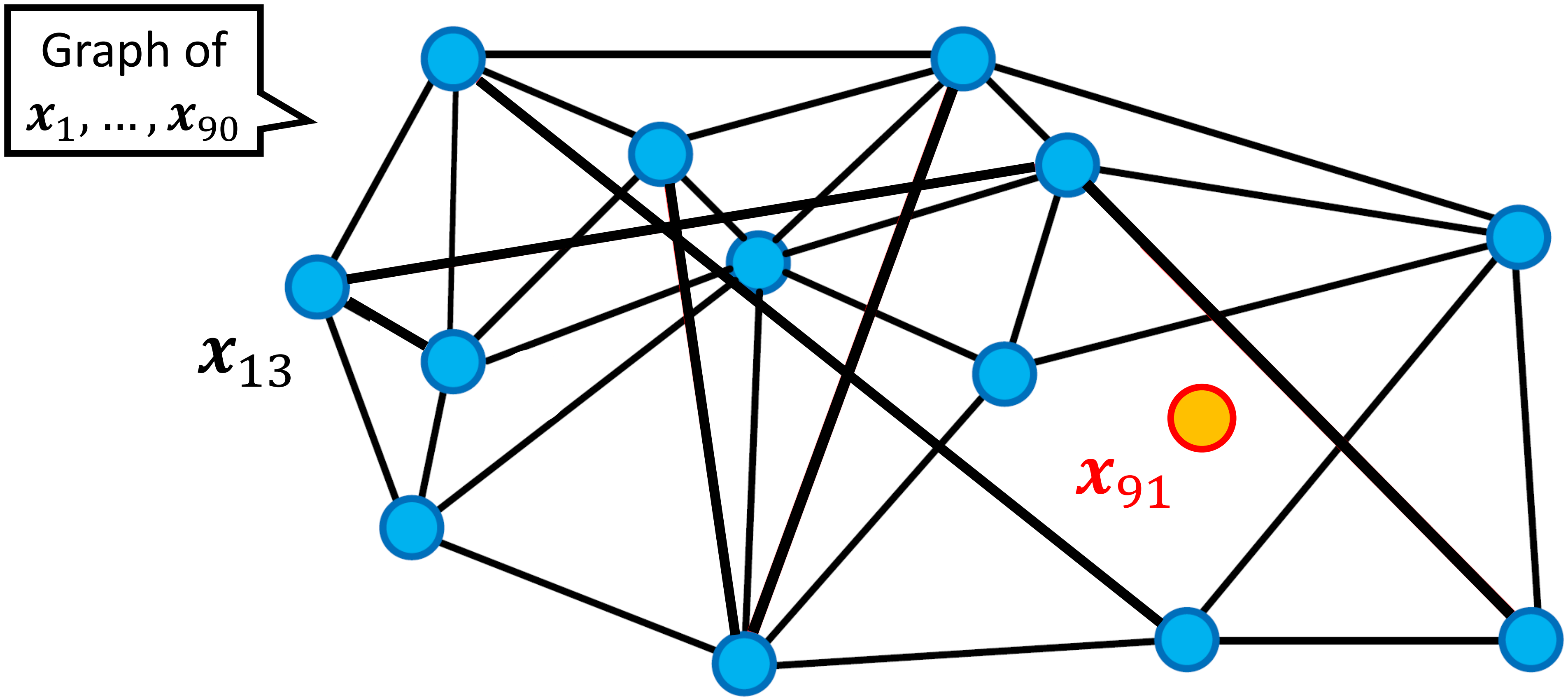
- Implementation: nmslib, hnsw, faiss

Record Images are from [Malkov+, Information Systems, 2013]



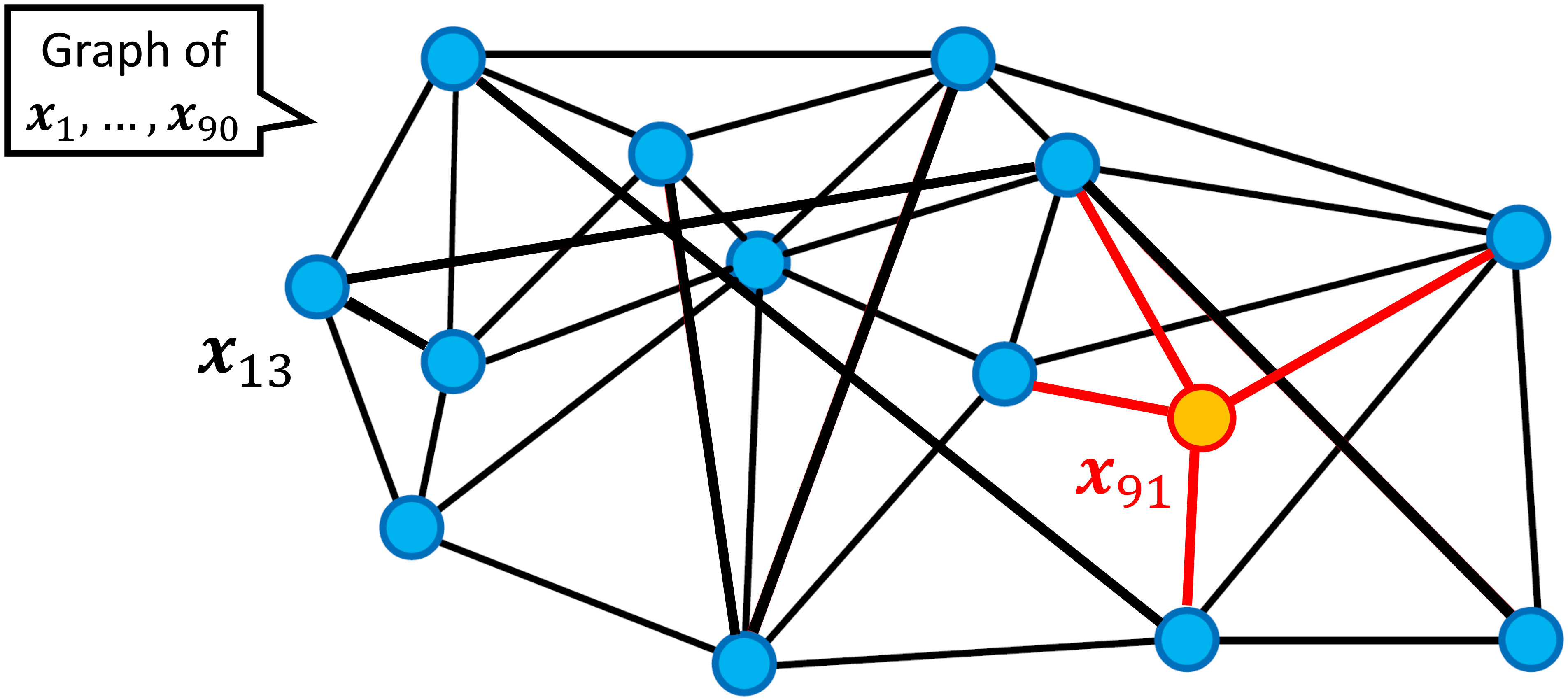
- Each node is a database vector

Record Images are from [Malkov+, Information Systems, 2013]



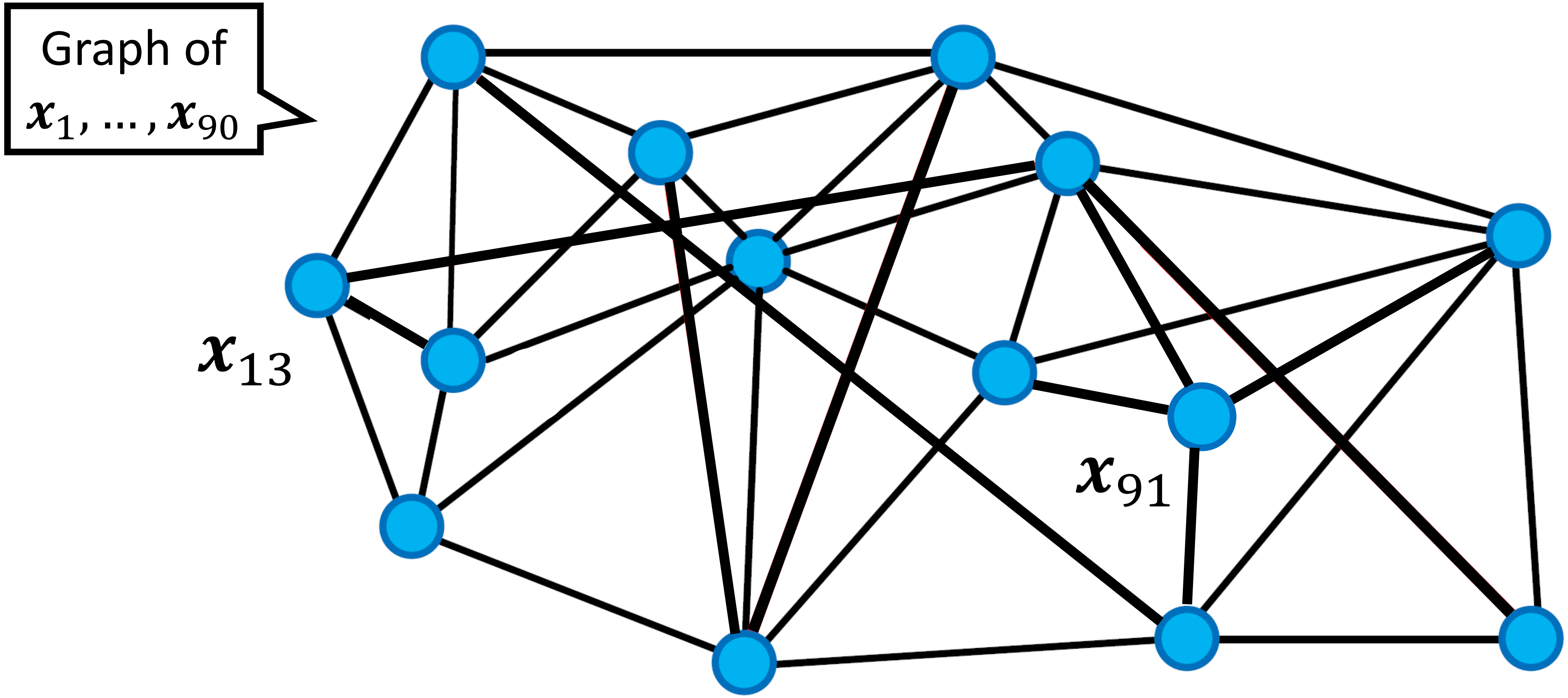
- ➢Each node is a database vector
- ➢Given a new database vector, create new edges to neighbors

Record Images are from [Malkov+, Information Systems, 2013]



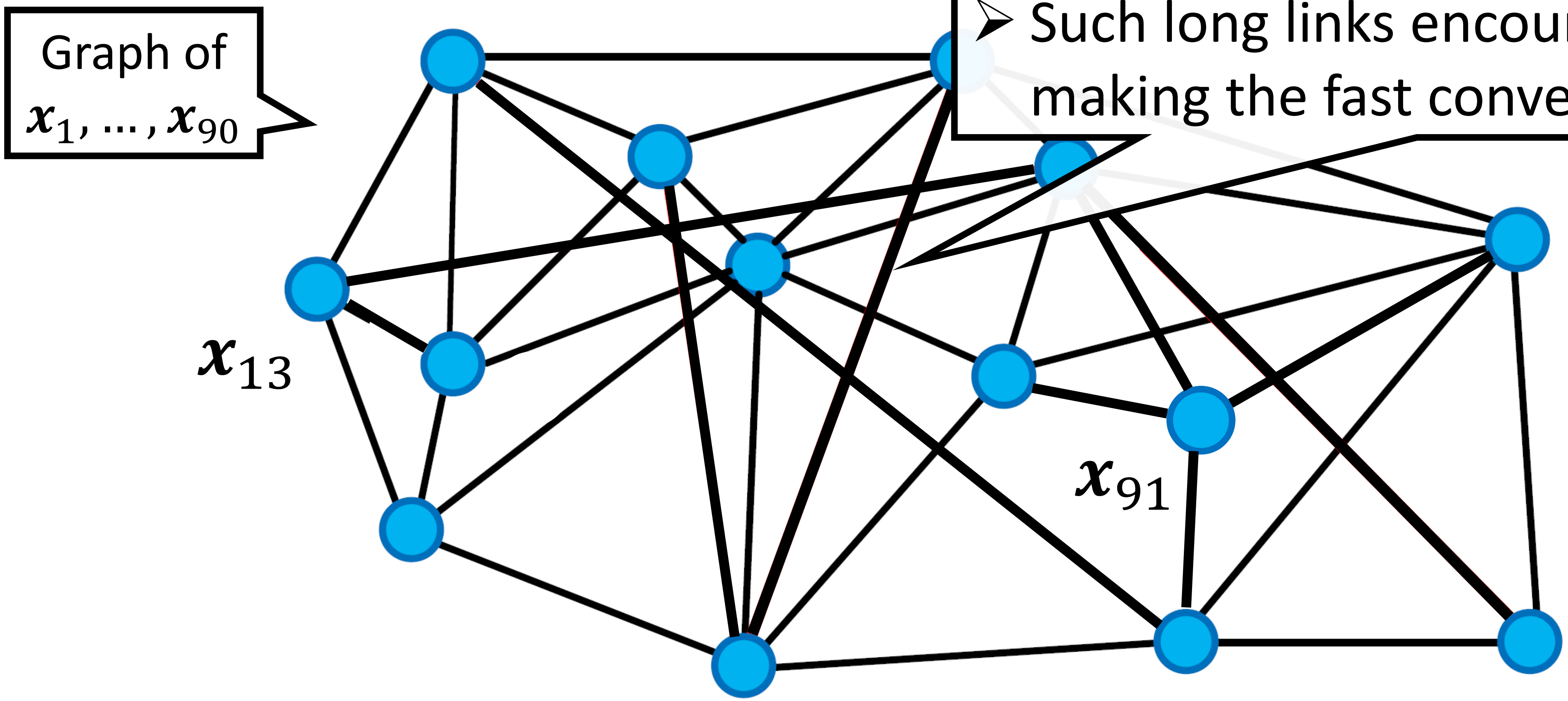
- Each node is a database vector
- Given a new database vector, create new edges to neighbors

Record Images are from [Malkov+, Information Systems, 2013]



- ➢Each node is a database vector
- ➢Given a new database vector, create new edges to neighbors

Record Images are from [Malkov+, Information Systems, 2013]

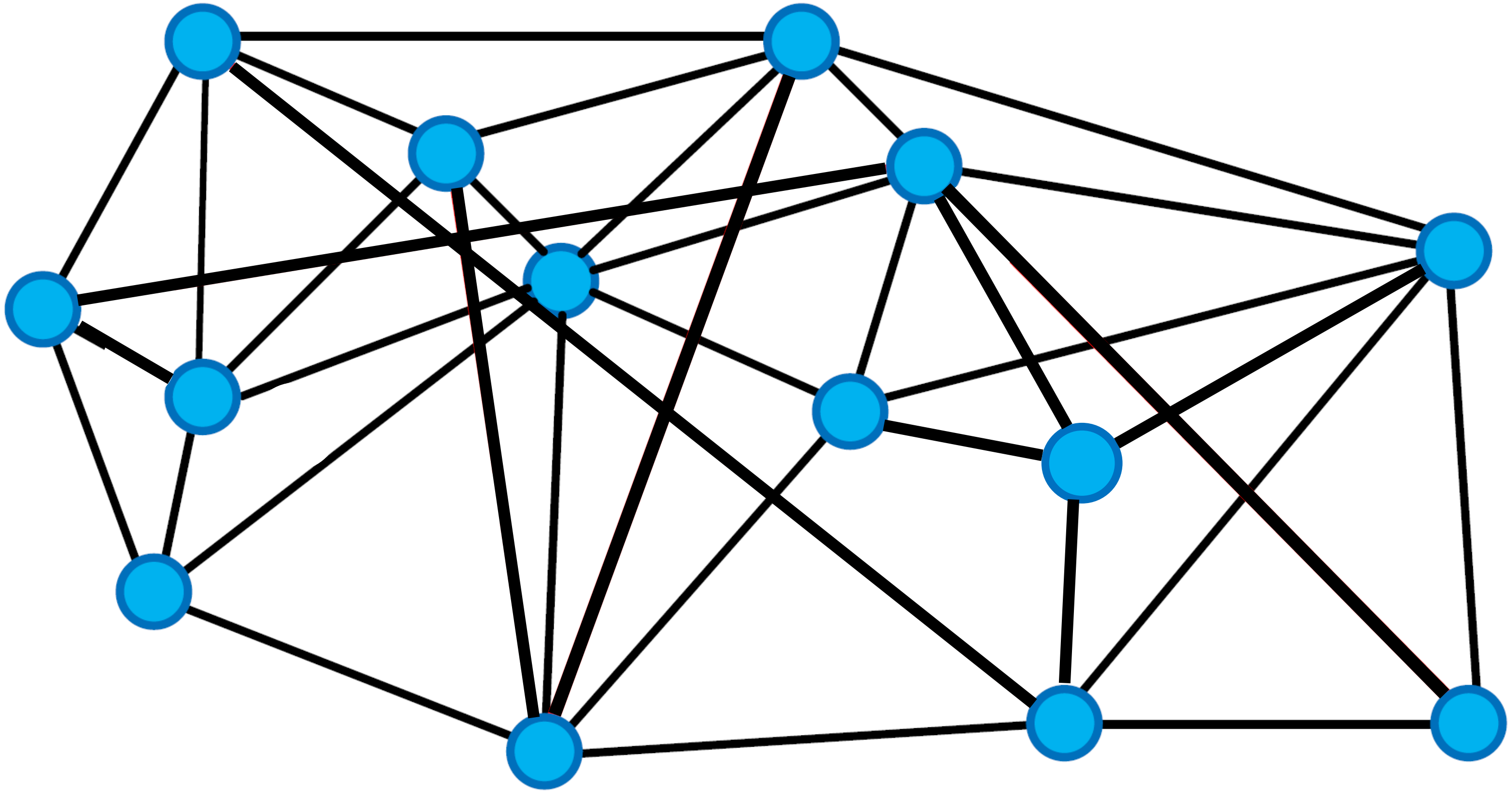


- ➢ Early links can be long
- ➢ Such long links encourage a large hop, making the fast convergence for search

- ➢Each node is a database vector
- ➢Given a new database vector, create new edges to neighbors

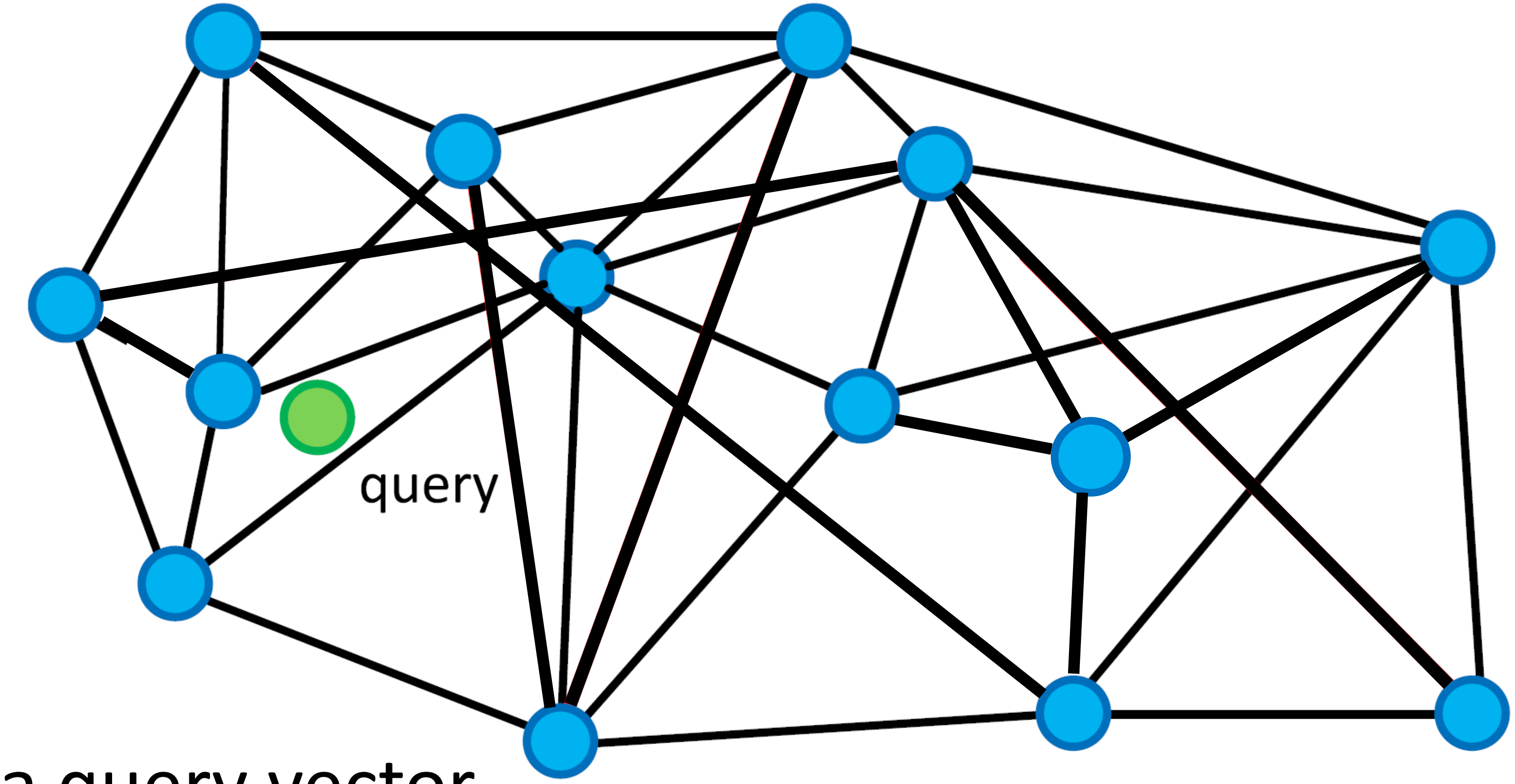
Search

Images are from [Malkov+, Information Systems, 2013]



Search

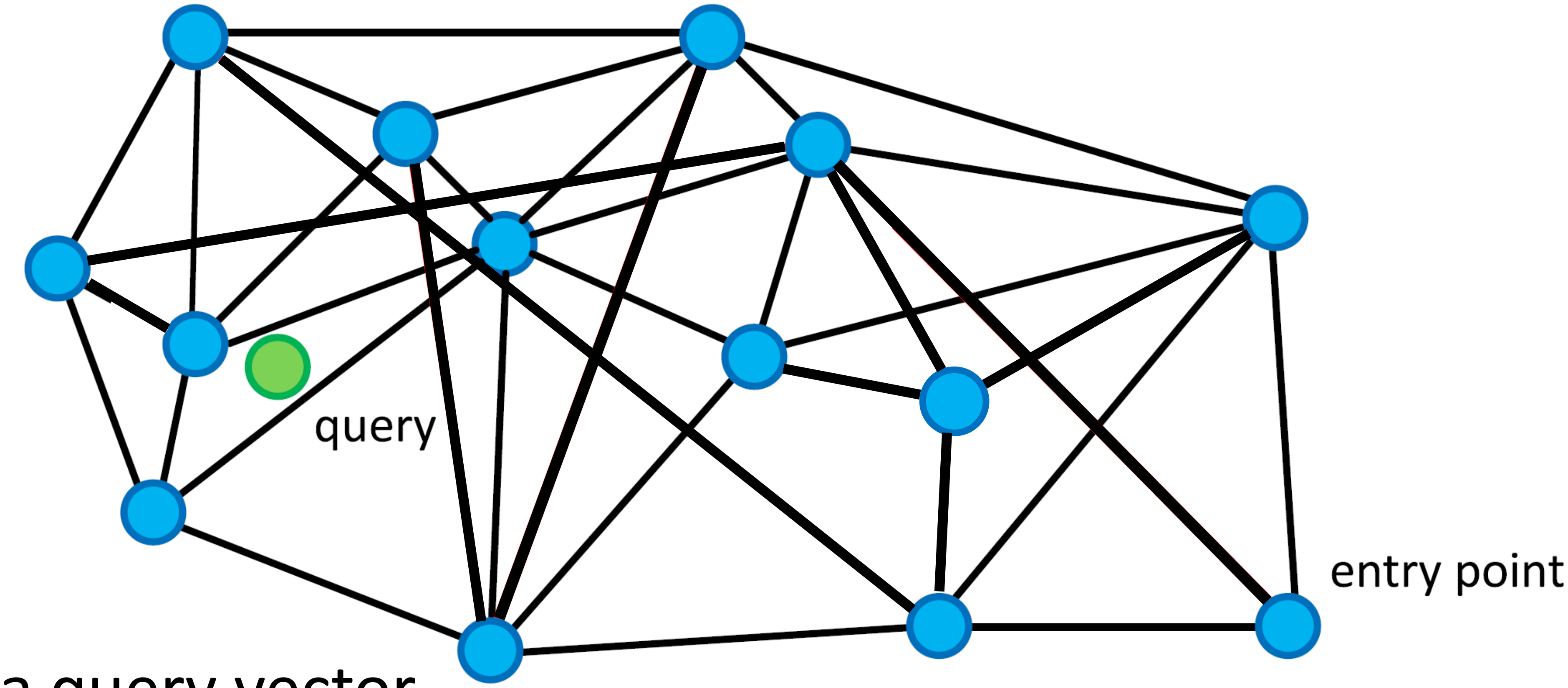
Images are from [Malkov+, Information Systems, 2013]



- Given a query vector

Search

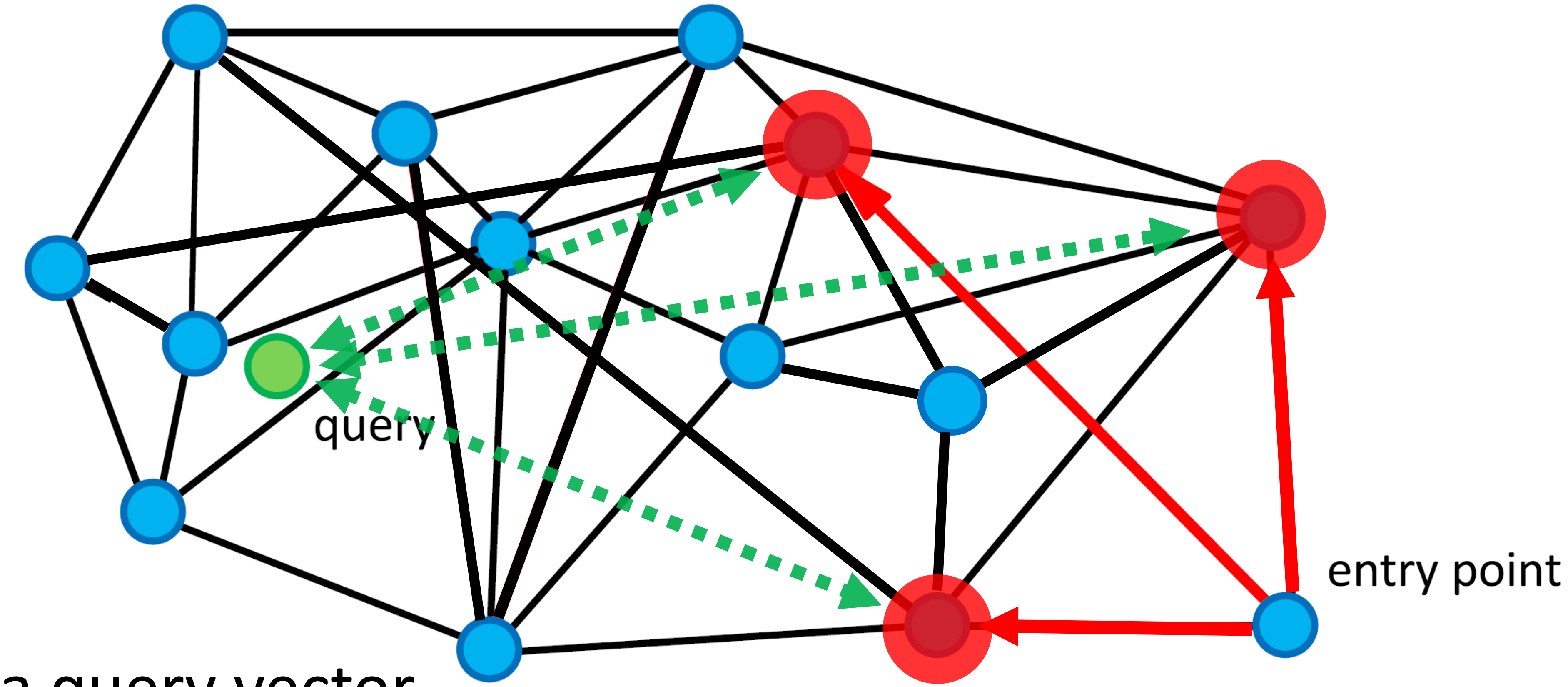
Images are from [Malkov+, Information Systems, 2013]



- Given a query vector
- Start from a random point

Search

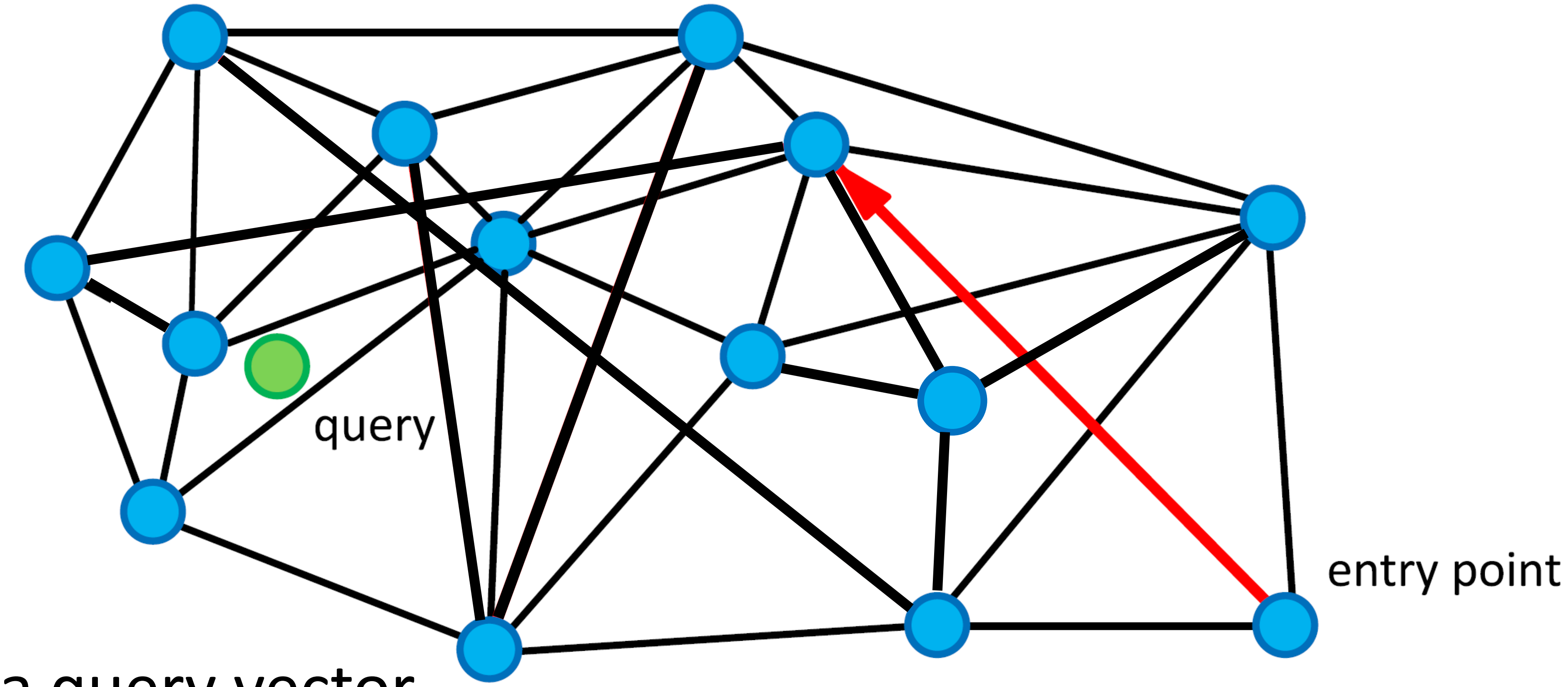
Images are from [Malkov+, Information Systems, 2013]



- Given a query vector
- Start from a random point
- From the connected nodes, find the closest one to the query

Search

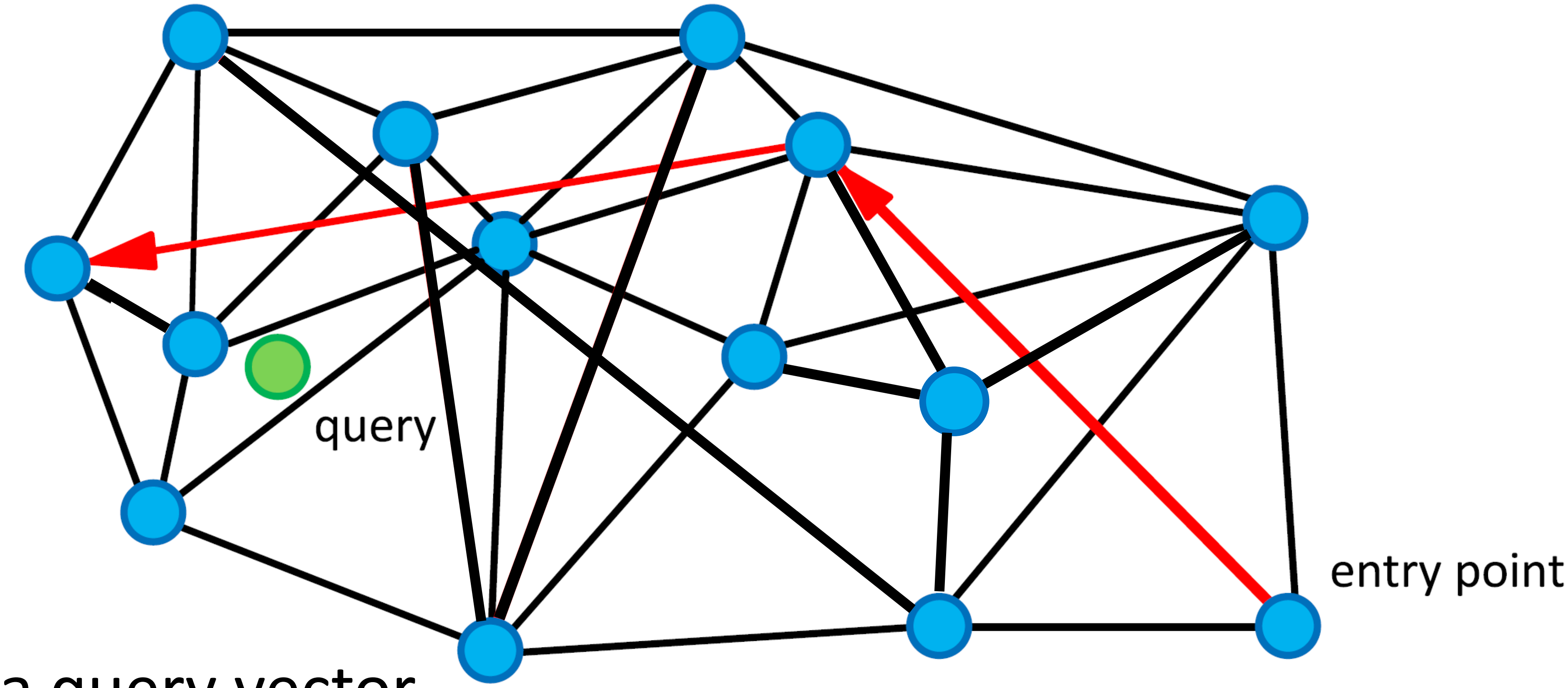
Images are from [Malkov+, Information Systems, 2013]



- Given a query vector
- Start from a random point
- From the connected nodes, find the closest one to the query

Search

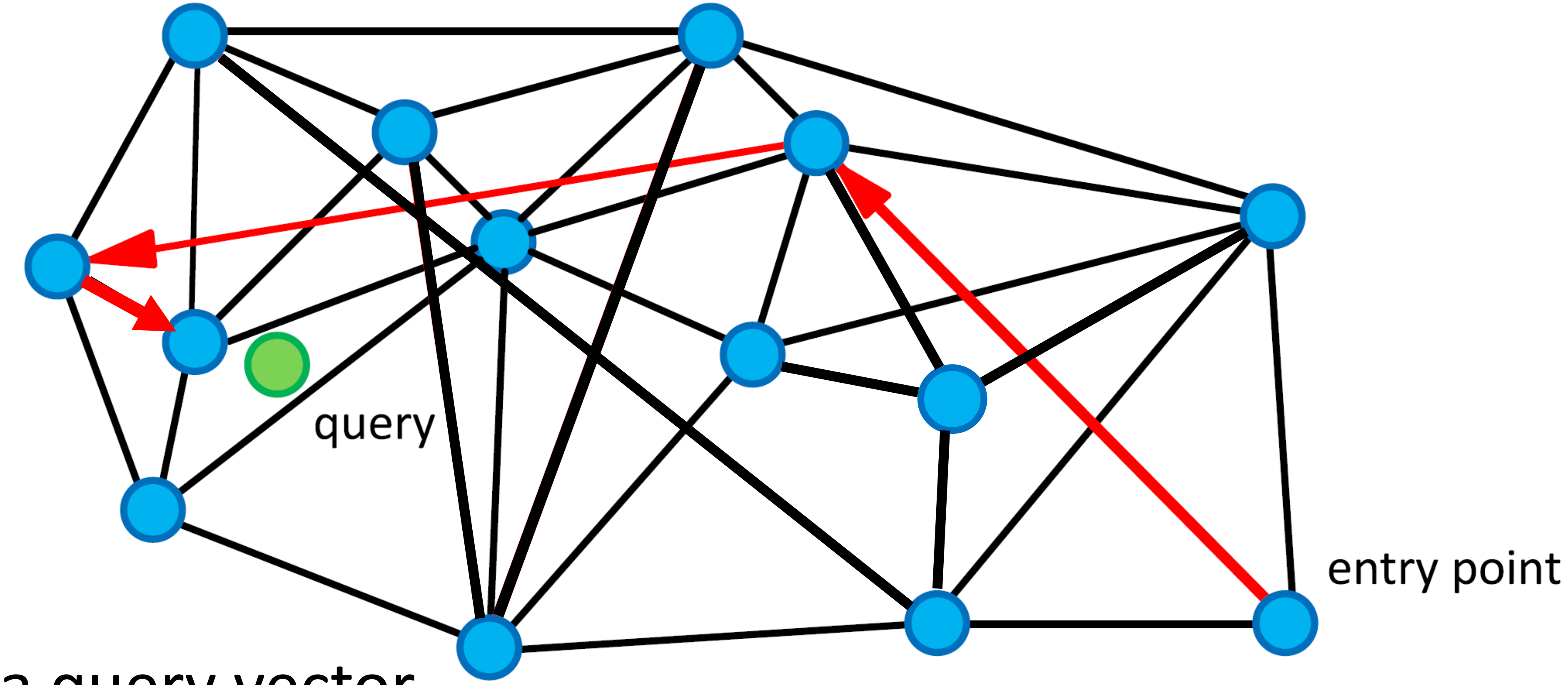
Images are from [Malkov+, Information Systems, 2013]



- Given a query vector
- Start from a random point
- From the connected nodes, find the closest one to the query
- Traverse in a greedy manner

Search

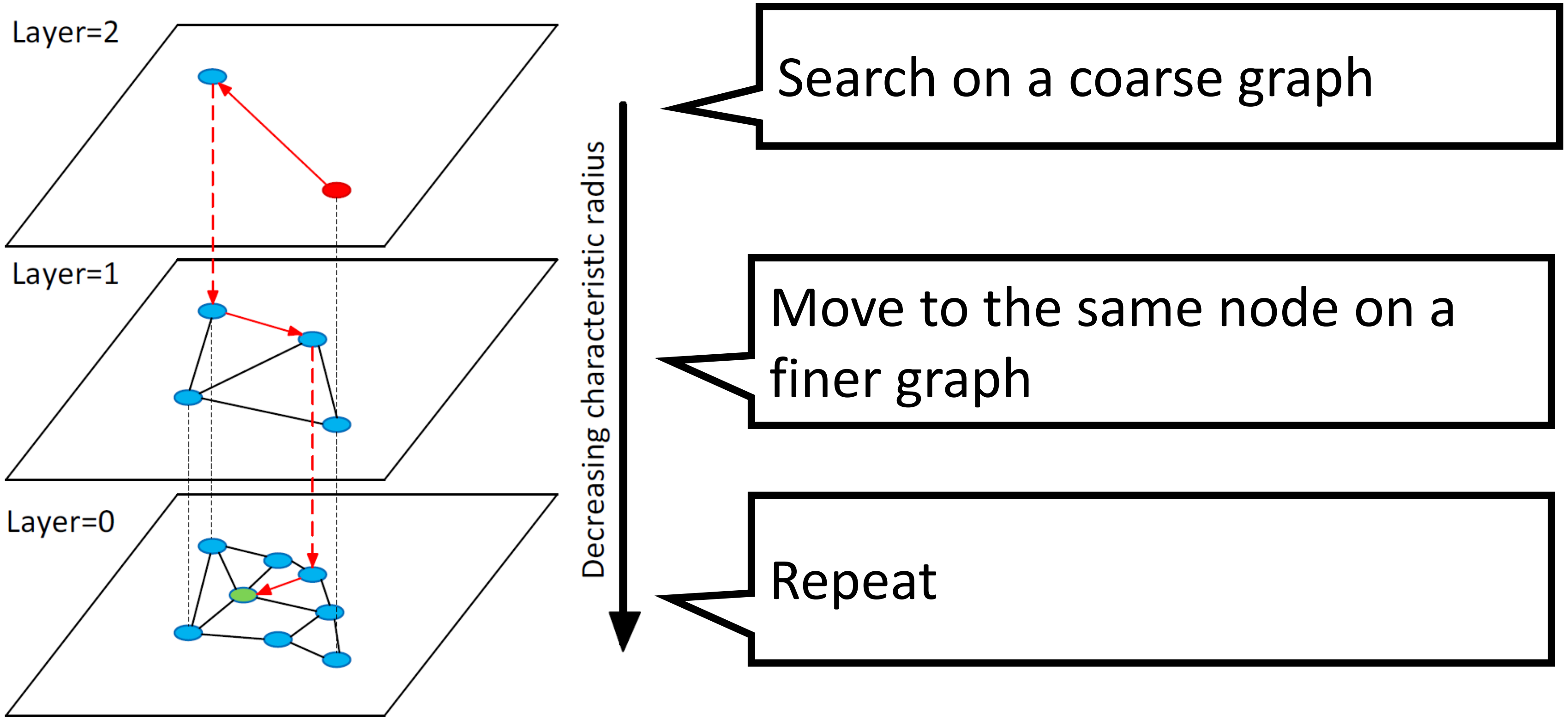
Images are from [Malkov+, Information Systems, 2013]



- Given a query vector
- Start from a random point
- From the connected nodes, find the closest one to the query
- Traverse in a greedy manner

Extension: Hierarchical NSW; HNSW

- Construct the graph hierarchically [Malkov and Yashunin, TPAMI, 2019]
- This structure works pretty well for real-world data



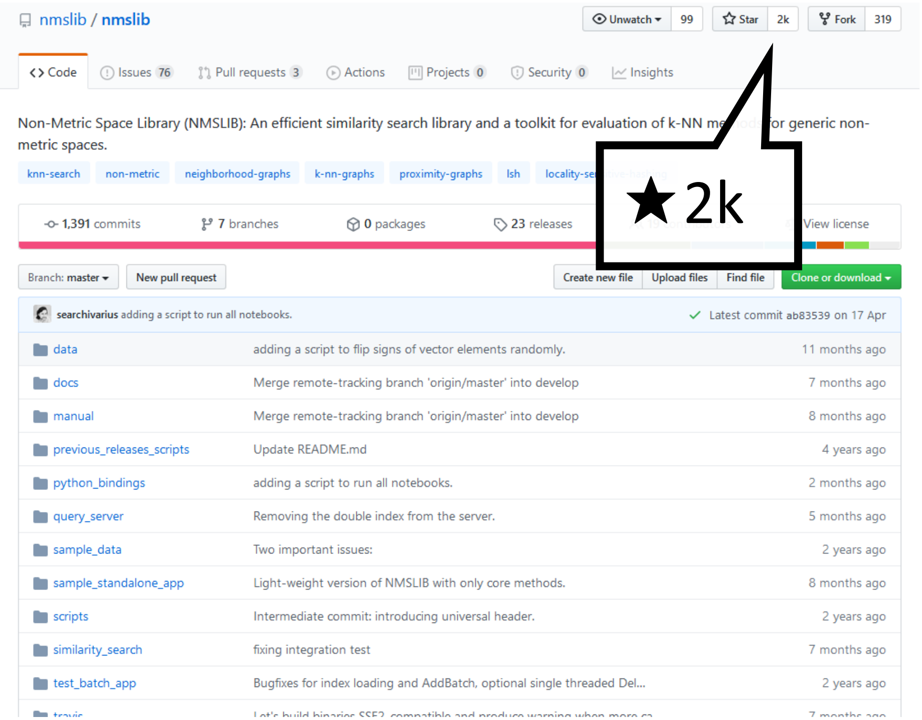
[Malkov and Yashunin, TPAMI, 2019]

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)
```



- ☺ The “hnsw” is the best method as of 2020 for million-scale data
- ☺ Simple interface
- ☺ If memory consumption is not the problem, try this
- ☹ Large memory consumption
- ☹ Data addition is not fast

Other implementations of HNSW

Hnswlib: https://github.com/nmslib/hnswlib

- Spin-off library from nmslib
- Include only hnsw
- Simpler; may be useful if you want to extend hnsw

Faiss: https://github.com/facebookresearch/faiss

- Libraries for PQ-based methods. Will Introduce later
- This lib also includes hnsw

Other graph-based approaches

- From Alibaba:

C. Fu et al., “Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph”, VLDB19
https://github.com/ZJULearning/nsg

- From Microsoft Research Asia. Used inside Bing:

J. Wang and S. Lin, “Query-Driven Iterated Neighborhood Graph Search for Large Scale Indexing”, ACMMM12 (This seems the backbone paper)
https://github.com/microsoft/SPTAG

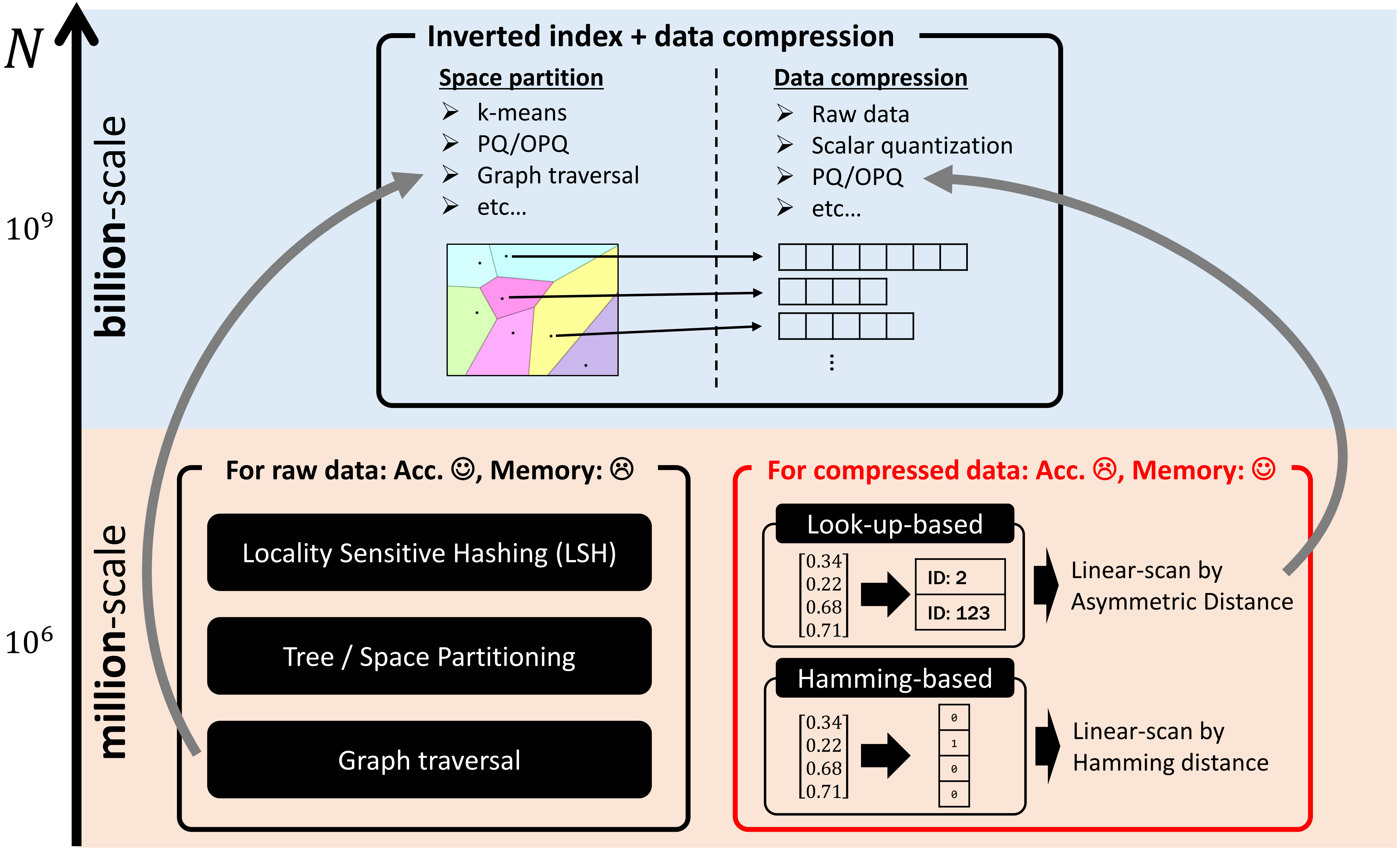
- From Yahoo Japan. Competing with NMSLIB for the 1st place of benchmark:

M. Iwasaki and D. Miyazaki, “Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data”, arXiv18
https://github.com/yahoojapan/NGT

Reference

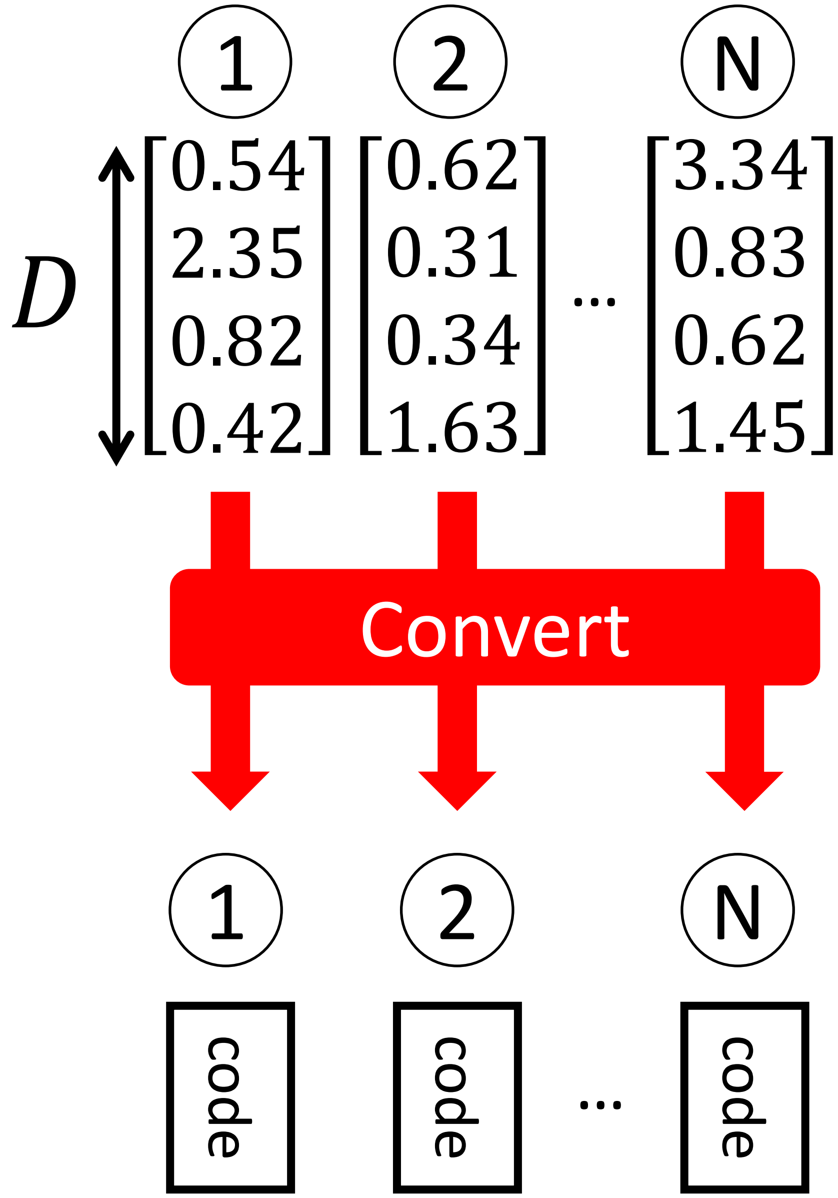
- The original paper of Navigable Small World Graph: Y. Malkov et al., “Approximate Nearest Neighbor Algorithm based on Navigable Small World Graphs,” Information Systems 2013

- The original paper of Hierarchical Navigable Small World Graph: Y. Malkov and D. Yashunin, “Efficient and Robust Approximate Nearest Neighbor search using Hierarchical Navigable Small World Graphs,” IEEE TPAMI 2019



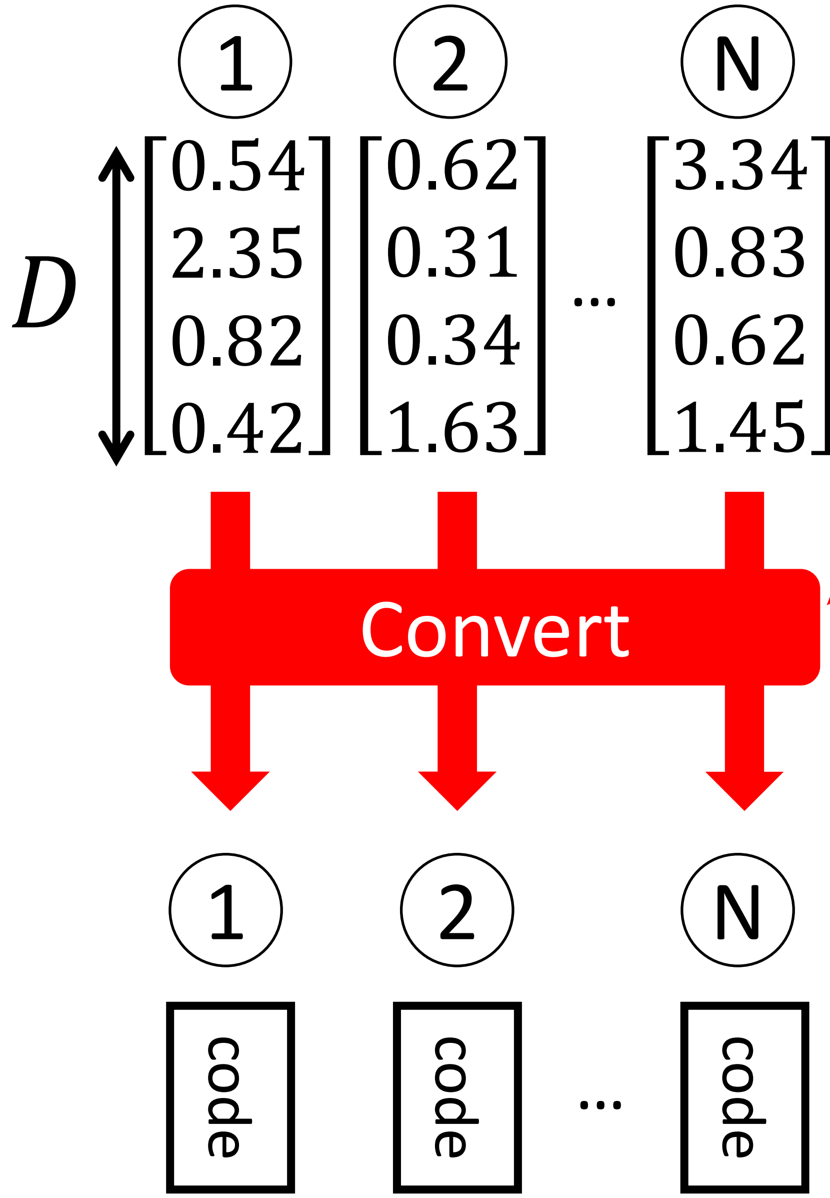
N
10^9
10^6
billion-scale
million-scale
Inverted index + data compression
Space partition
k-means
PQ/OPQ
Graph traversal
etc…
Data compression
Raw data
Scalar quantization
PQ/OPQ
etc…
For raw data: Acc. ☺, Memory: ☹
Locality Sensitive Hashing (LSH)
Tree / Space Partitioning
Graph traversal
For compressed data: Acc. ☹, Memory: ☺
Look-up-based
0.34 0.22 0.68 0.71
ID: 2
ID: 123
Linear-scan by Asymmetric Distance
Hamming-based
0.34 0.22 0.68 0.71
0 1 0 0
Linear-scan by Hamming distance

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 (e.g., Hamming-distance)
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

- Convert $\boldsymbol{x}$ to a B-bit binary vector:

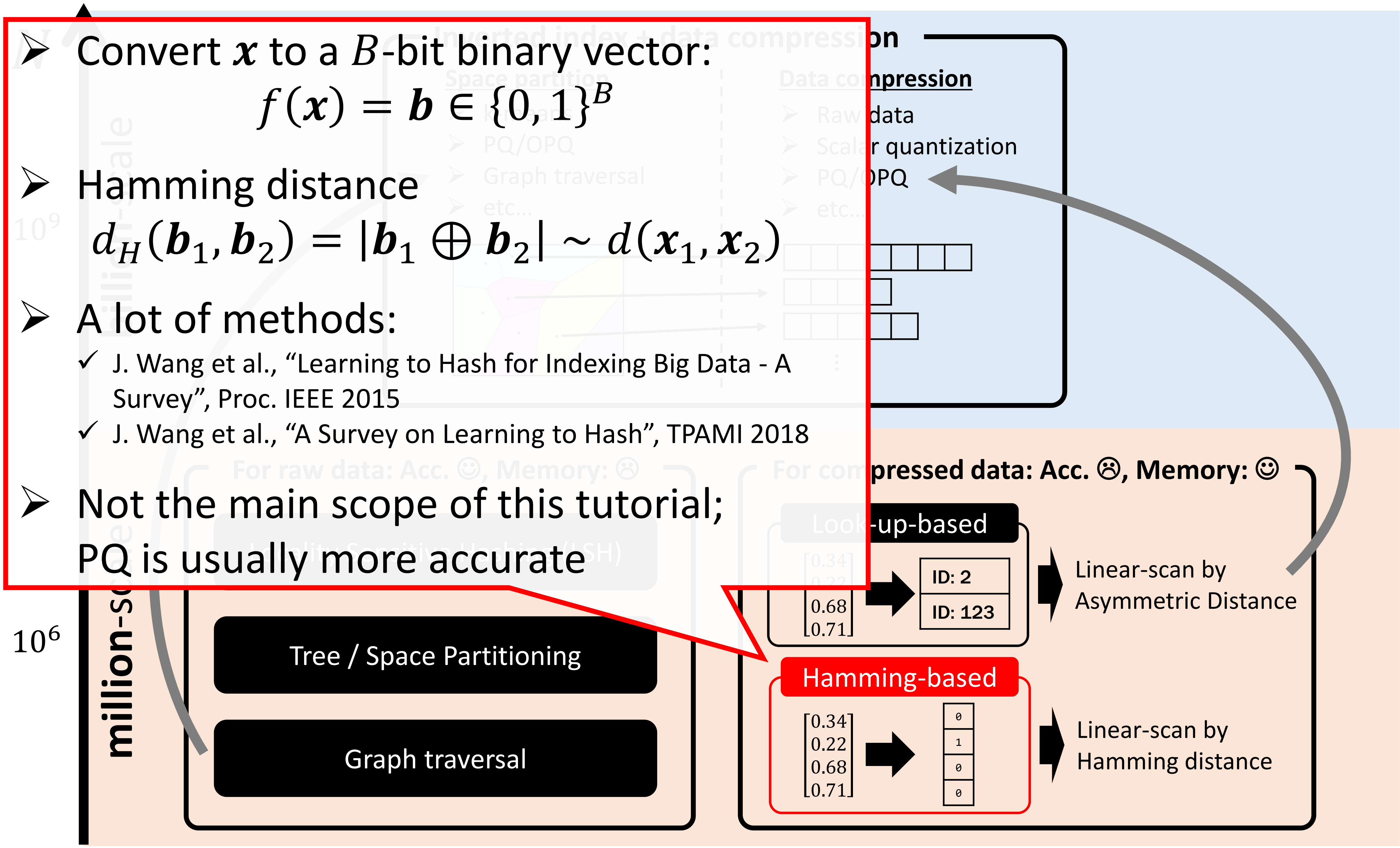
$$f(\boldsymbol{x}) = \boldsymbol{b} \in \{0,1\}^B$$

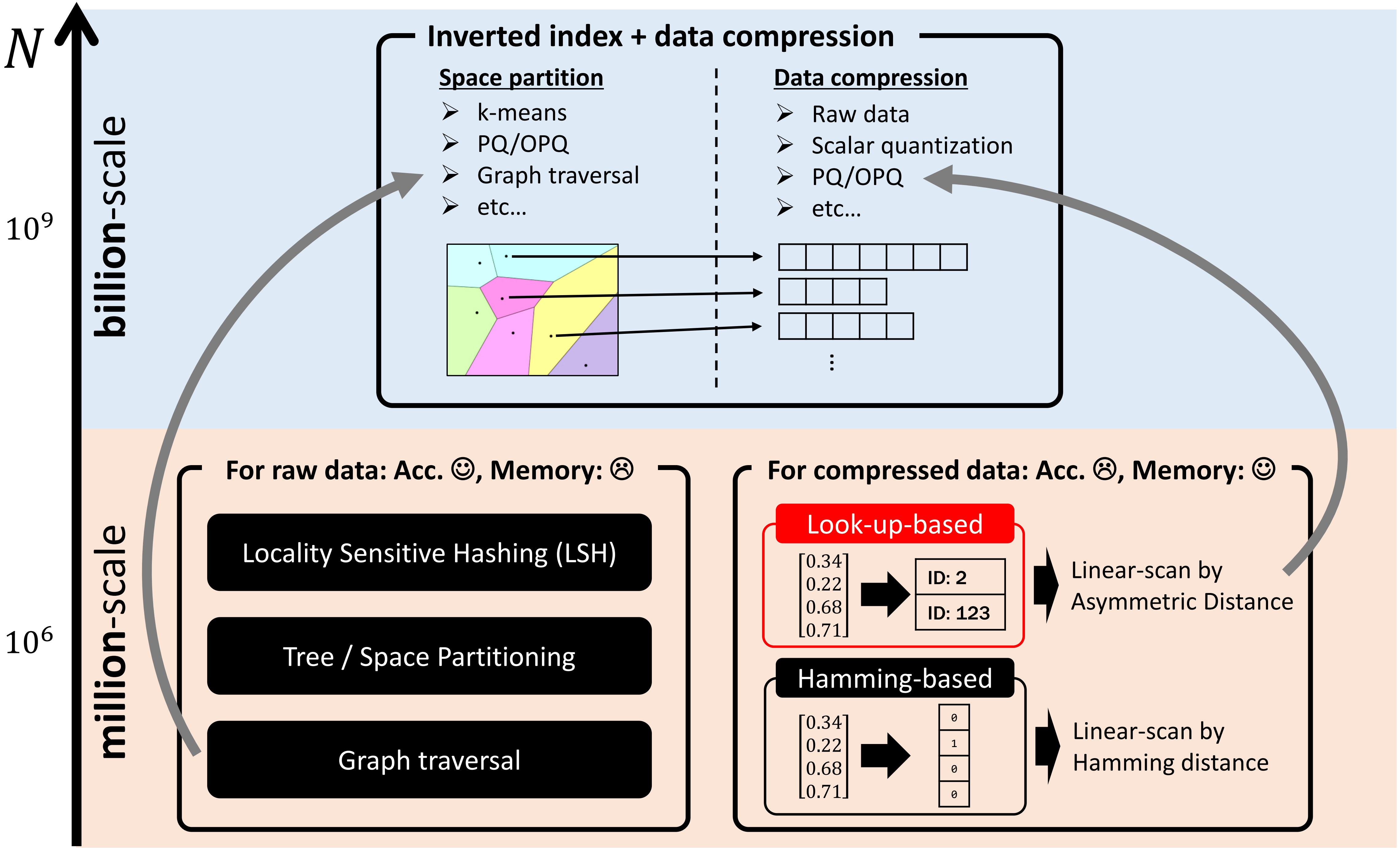
- Hamming distance

$$d_H(\boldsymbol{b}_1, \boldsymbol{b}_2) = |\boldsymbol{b}_1 \oplus \boldsymbol{b}_2| \sim d(\boldsymbol{x}_1, \boldsymbol{x}_2)$$

- A lot of methods:
  - ✓ J. Wang et al., "Learning to Hash for Indexing Big Data - A Survey", Proc. IEEE 2015
  - ✓ J. Wang et al., "A Survey on Learning to Hash", TPAMI 2018

- Not the main scope of this tutorial; PQ is usually more accurate





N
10^9
10^6
billion-scale
million-scale
Inverted index + data compression
Space partition
k-means
PQ/OPQ
Graph traversal
etc…
Data compression
Raw data
Scalar quantization
PQ/OPQ
etc…
For raw data: Acc. ☺, Memory: ☹
Locality Sensitive Hashing (LSH)
Tree / Space Partitioning
Graph traversal
For compressed data: Acc. ☹, Memory: ☺
Look-up-based
0.34
0.22
0.68
0.71
ID: 2
ID: 123
Linear-scan by Asymmetric Distance
Hamming-based
0.34
0.22
0.68
0.71
0
1
0
0
Linear-scan by Hamming distance

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

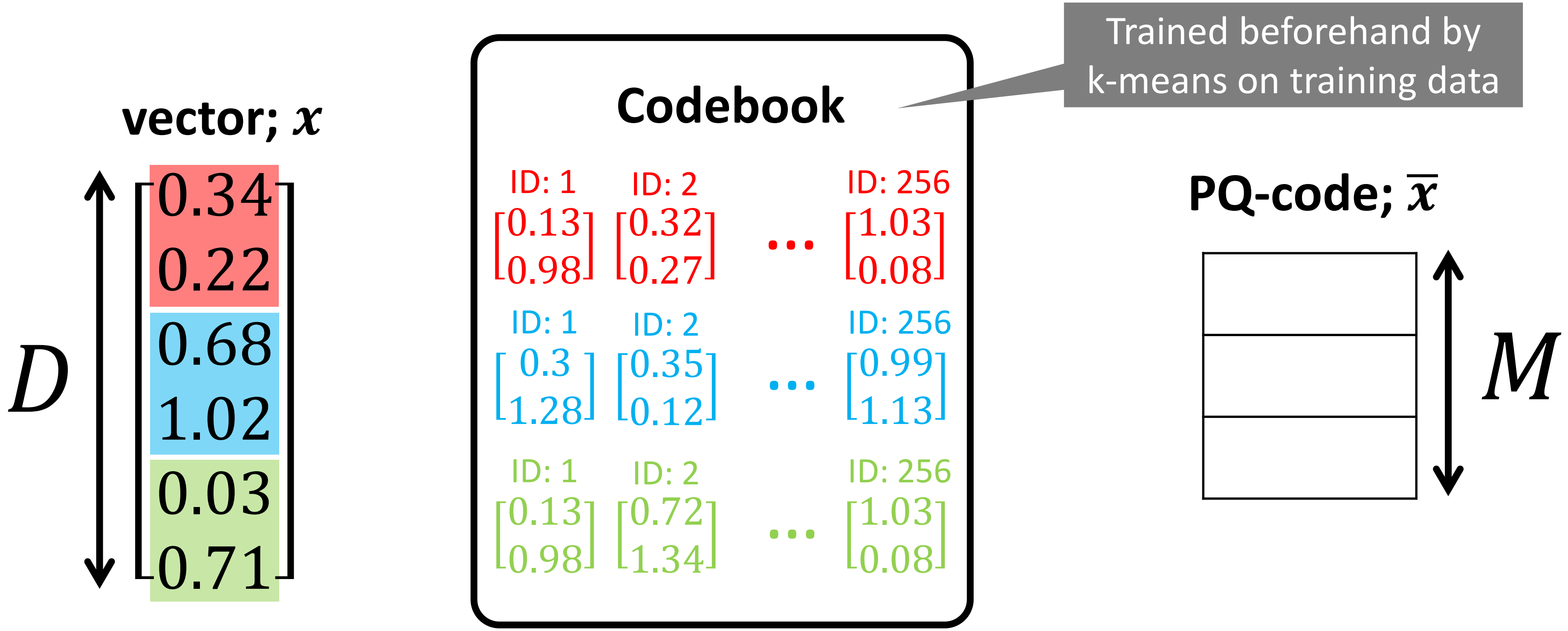
ID: 1	ID: 2		ID: 256
$\begin{bmatrix} 0.13 \\ 0.98 \end{bmatrix}$	$\begin{bmatrix} 0.32 \\ 0.27 \end{bmatrix}$	…	$\begin{bmatrix} 1.03 \\ 0.08 \end{bmatrix}$
ID: 1	ID: 2		ID: 256
$\begin{bmatrix} 0.3 \\ 1.28 \end{bmatrix}$	$\begin{bmatrix} 0.35 \\ 0.12 \end{bmatrix}$	…	$\begin{bmatrix} 0.99 \\ 1.13 \end{bmatrix}$
ID: 1	ID: 2		ID: 256
$\begin{bmatrix} 0.13 \\ 0.98 \end{bmatrix}$	$\begin{bmatrix} 0.72 \\ 1.34 \end{bmatrix}$	…	$\begin{bmatrix} 1.03 \\ 0.08 \end{bmatrix}$

PQ-code; $\bar{x}$

M

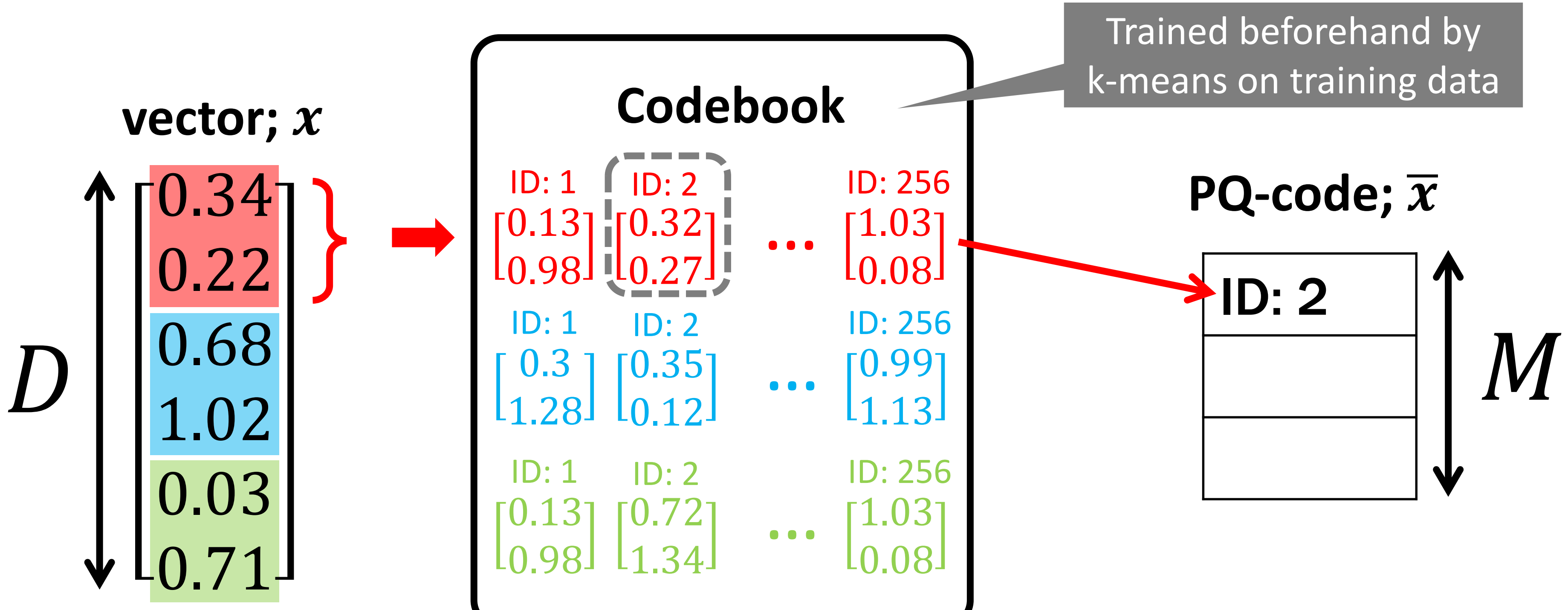
Product Quantization; PQ [Jégou, TPAMI 2011]

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



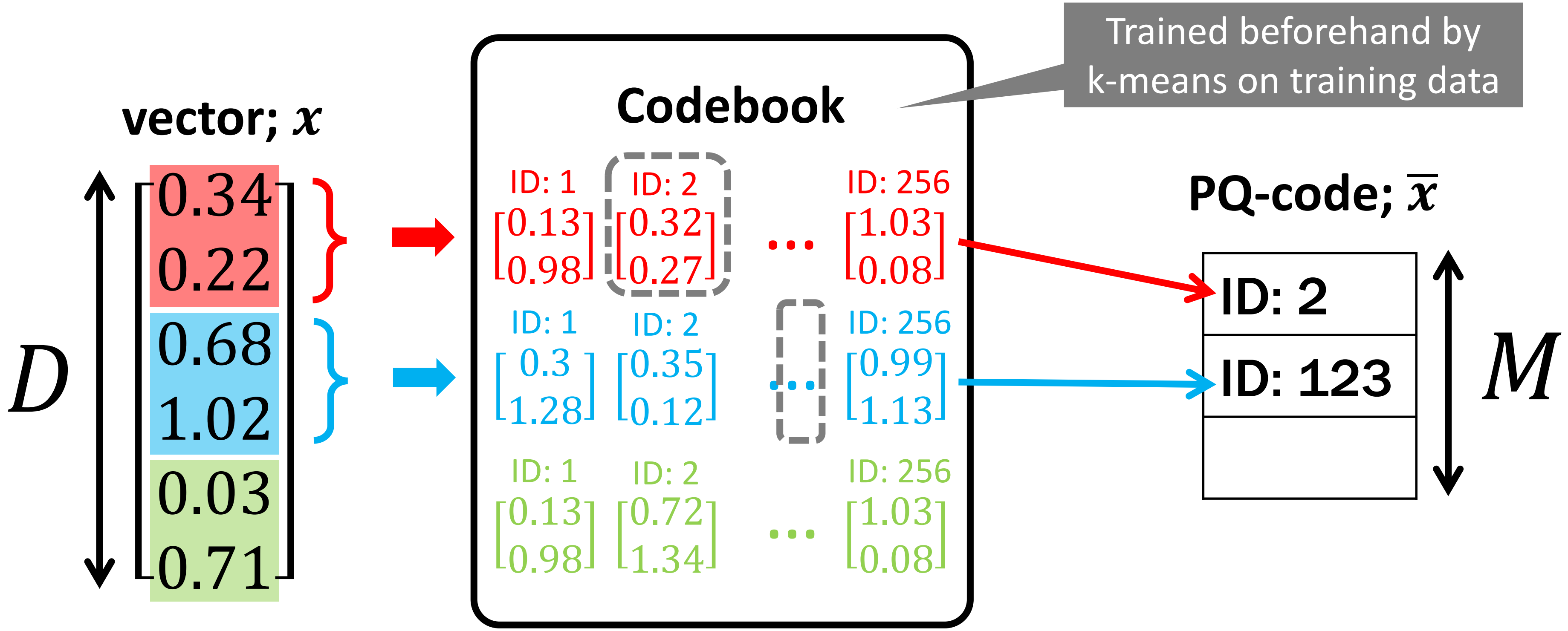
Product Quantization; PQ [Jégou, TPAMI 2011]

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



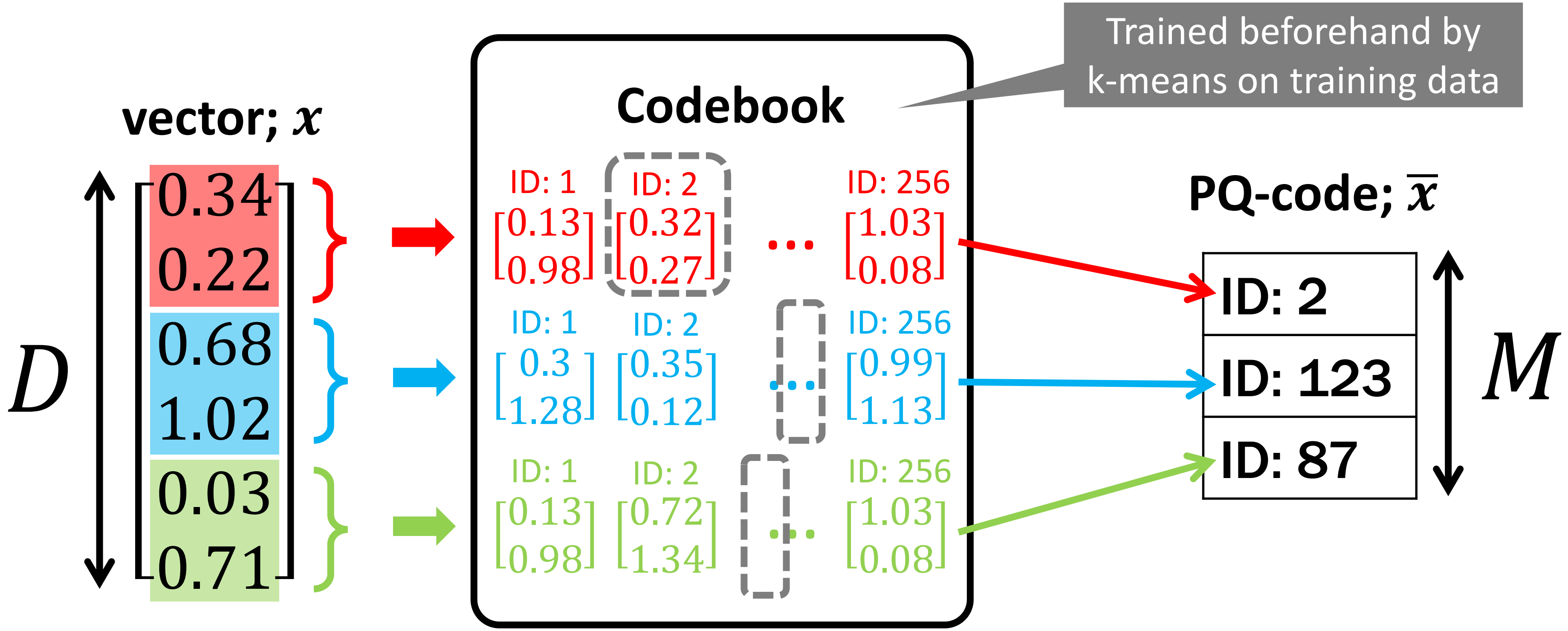
Product Quantization; PQ [Jégou, TPAMI 2011]

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



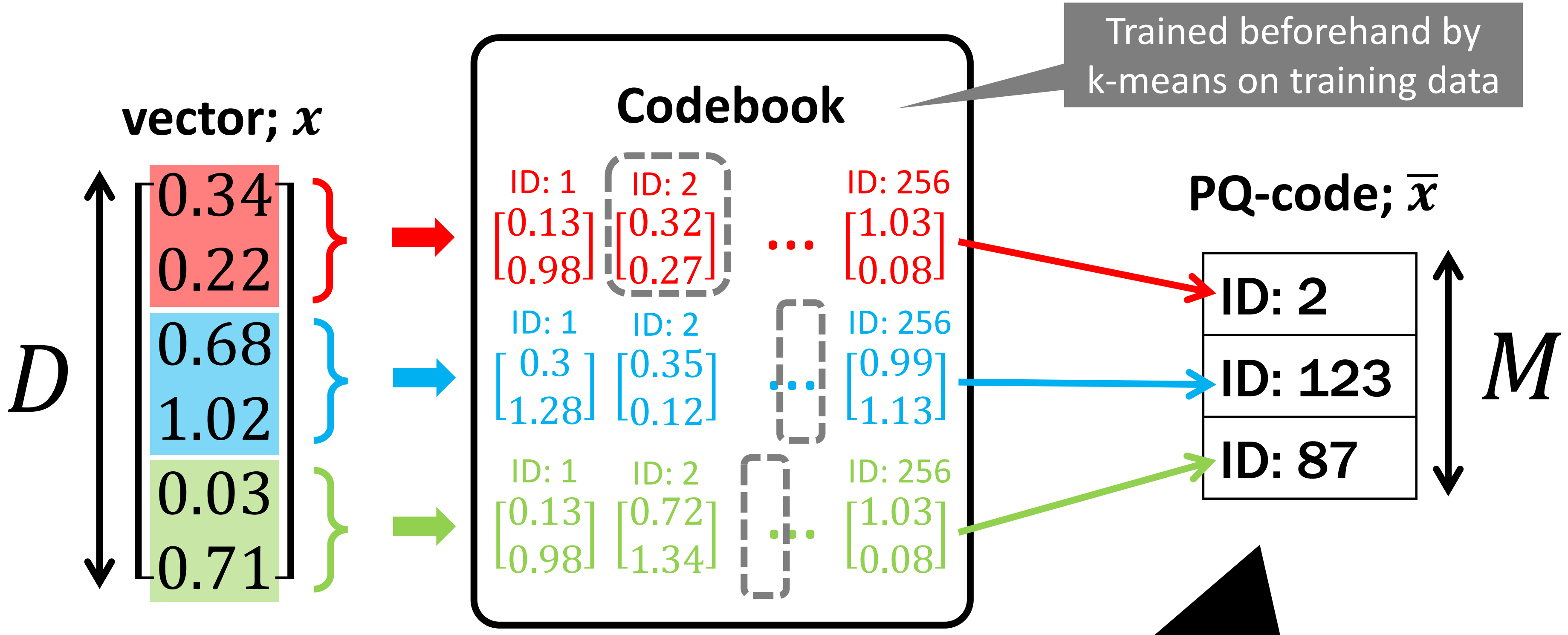
Product Quantization; PQ [Jégou, TPAMI 2011]

➢ 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

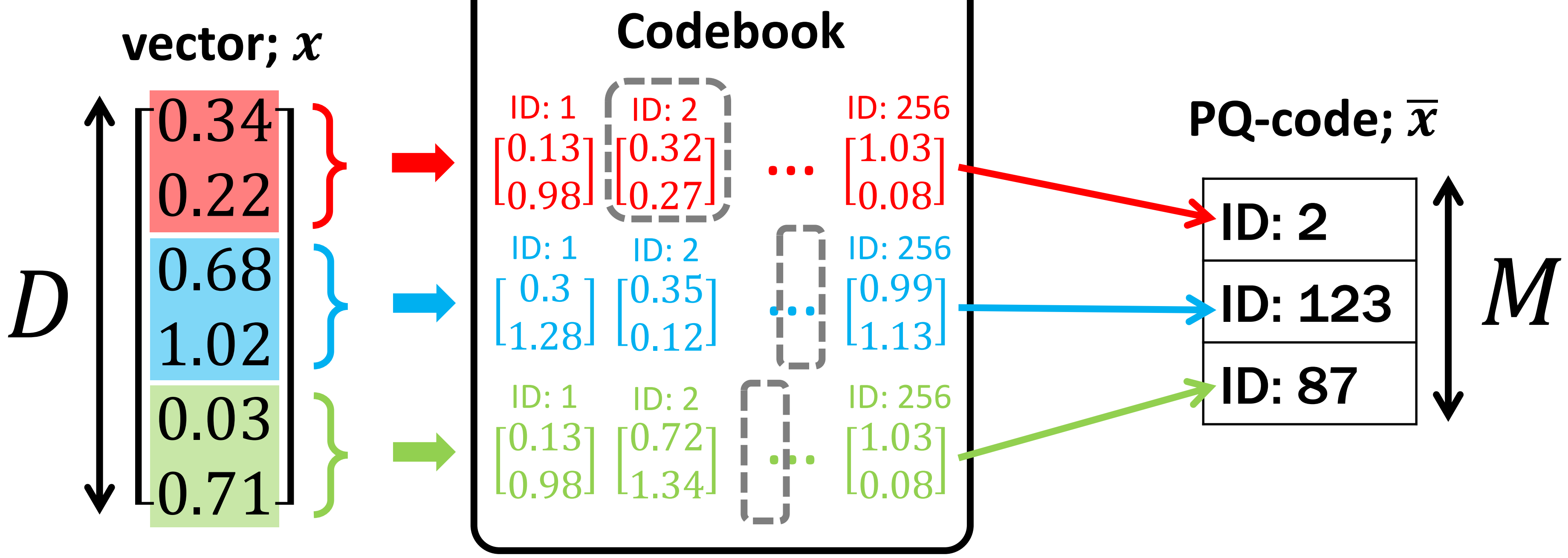


- Simple
- Memory efficient
- Distance can be esimated

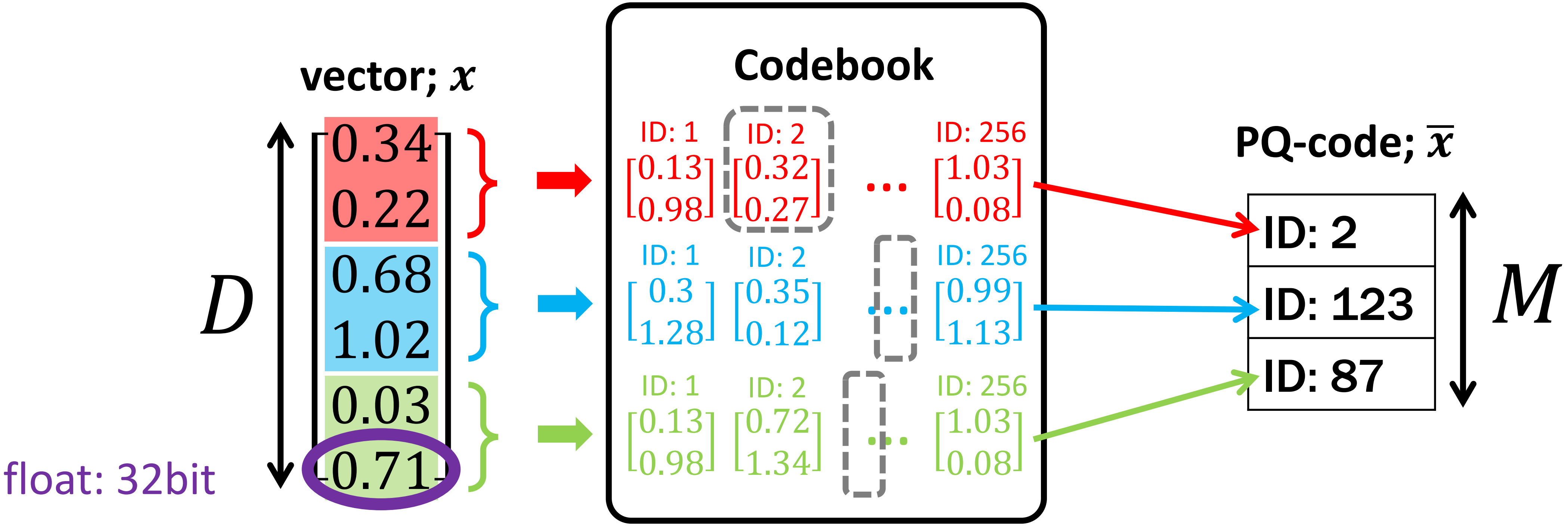
Bar notation for PQ-code in this tutorial:

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

Product Quantization: Memory efficient



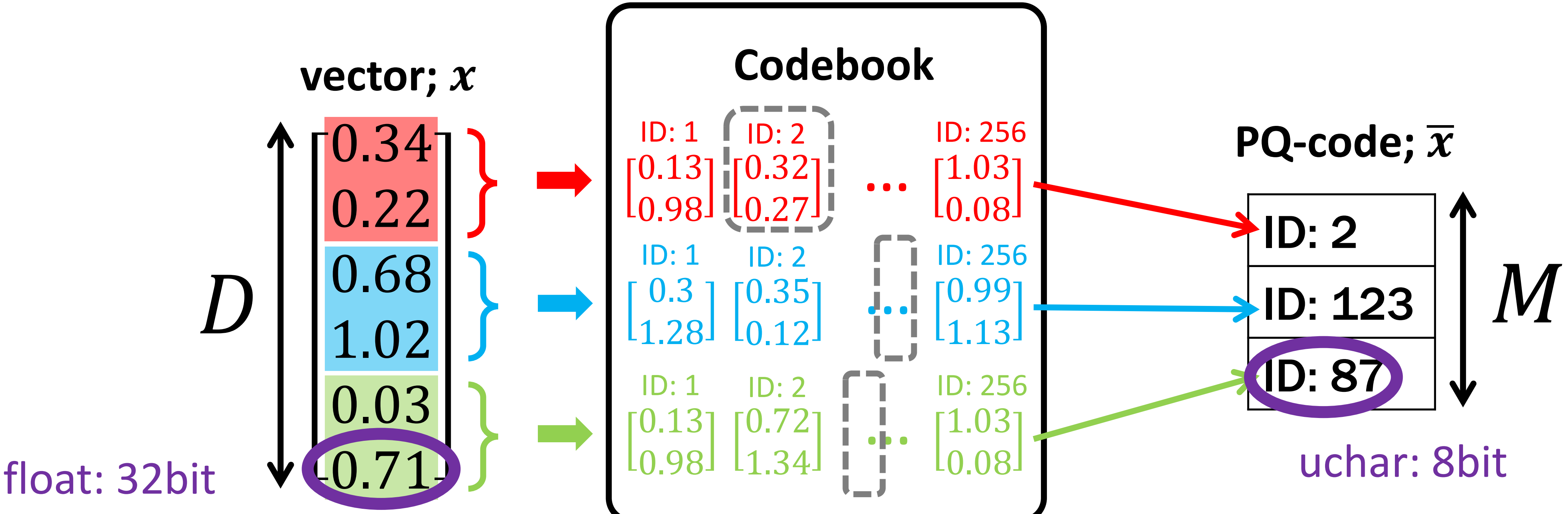
Product Quantization: Memory efficient



e.g., $D = 128$

$128 \times 32 = 4096$ [bit]

Product Quantization: Memory efficient



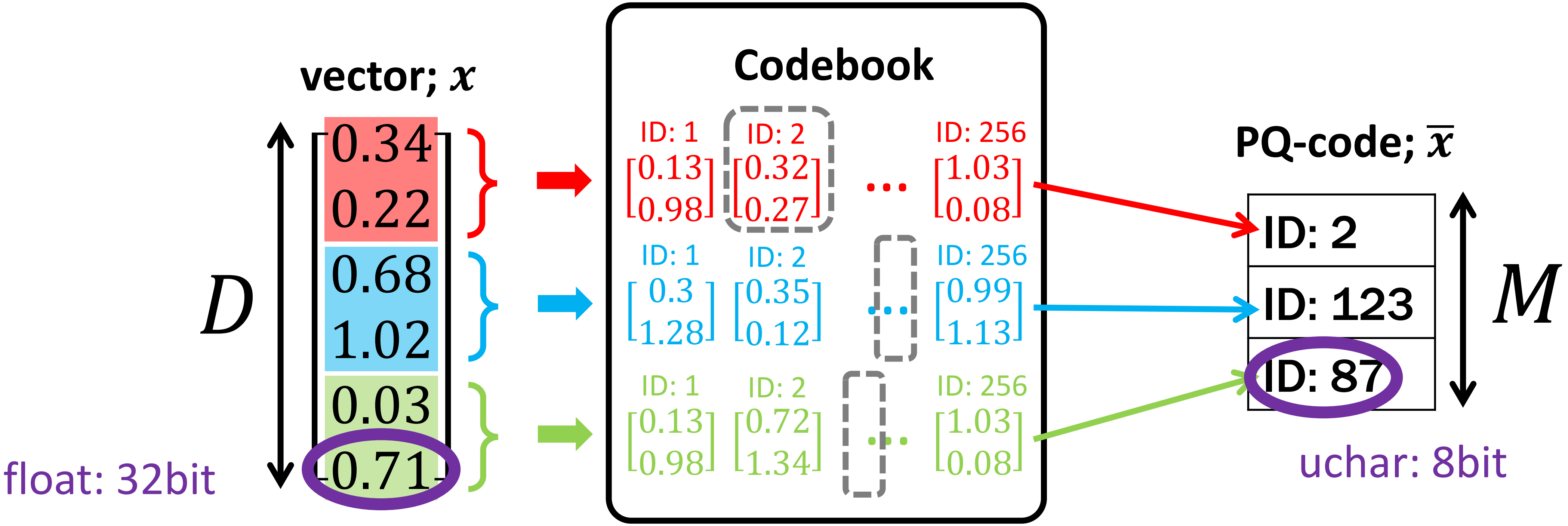
e.g., $D = 128$

$128 \times 32 =$ 4096 [bit]

e.g., $M = 8$

$8 \times 8 =$ 64 [bit]

Product Quantization: Memory efficient



e.g., $D = 128$
$128 \times 32 = 4096$ [bit]

e.g., $M = 8$
$8 \times 8 = 64$ [bit]

1/64

Product Quantization: Distance estimation

Database vectors

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

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

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

Product Quantization: Distance estimation

Database vectors

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

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

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

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

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

$\overline{\boldsymbol{x}}_1$	$\overline{\boldsymbol{x}}_2$	…	$\overline{\boldsymbol{x}}_N$
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
- Linear-scan by d_A

Not pseudo codes

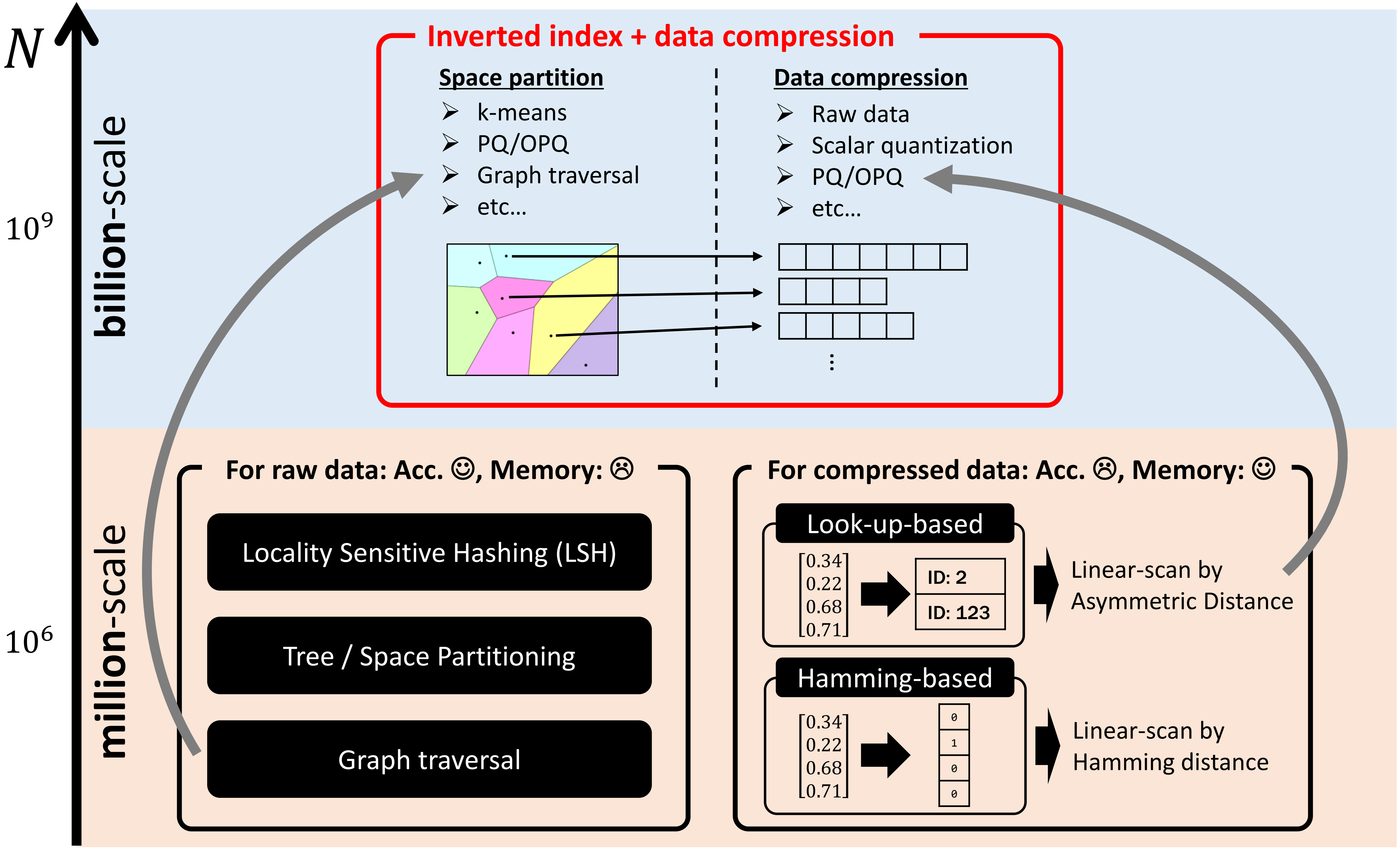
```python
import numpy as np
from scipy.cluster.vq import vq, kmeans2
from scipy.spatial.distance import cdist

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
```

```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)
```

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

Hamming-based vs Look-up-based

	Hamming-based	Look-up-based
	$\begin{bmatrix}0.34\\0.22\\0.68\\1.02\\0.03\\0.71\end{bmatrix} \Rightarrow \begin{bmatrix}0\\1\\0\\1\\0\\0\end{bmatrix}$	$\begin{bmatrix}0.34\\0.22\\0.68\\1.02\\0.03\\0.71\end{bmatrix} \Rightarrow$ ID: 2 / ID: 123 / ID: 87
Representation	Binary code : $\{0,1\}^B$	PQ code : $\{1,\ldots,256\}^M$
Distance	Hamming distance	Asymmetric distance
Approximation	☺	☺☺
Runtime	☺☺	☺
Pros	No auxiliary structure	Can reconstruct the original vector
Cons	Cannot reconstruct the original vector	Require an auxiliary structure (codebook)


N
10^9
10^6
billion-scale
million-scale
Inverted index + data compression
Space partition
k-means
PQ/OPQ
Graph traversal
etc…
Data compression
Raw data
Scalar quantization
PQ/OPQ
etc…
For raw data: Acc. ☺, Memory: ☹
Locality Sensitive Hashing (LSH)
Tree / Space Partitioning
Graph traversal
For compressed data: Acc. ☹, Memory: ☺
Look-up-based
0.34
0.22
0.68
0.71
ID: 2
ID: 123
Linear-scan by
Asymmetric Distance
Hamming-based
0.34
0.22
0.68
0.71
0
1
0
0
Linear-scan by
Hamming distance

Inverted index + PQ: Recap the notation

$$D \quad \begin{bmatrix} 0.34 \\ 0.22 \\ 0.68 \\ 1.02 \\ 0.03 \\ 0.71 \end{bmatrix} \xrightarrow{\text{Product quantization}}$$

$\boldsymbol{x} \in \mathbb{R}^D$

PQ code

ID: 2
ID: 123
ID: 87

M

Bar-notation = PQ-code

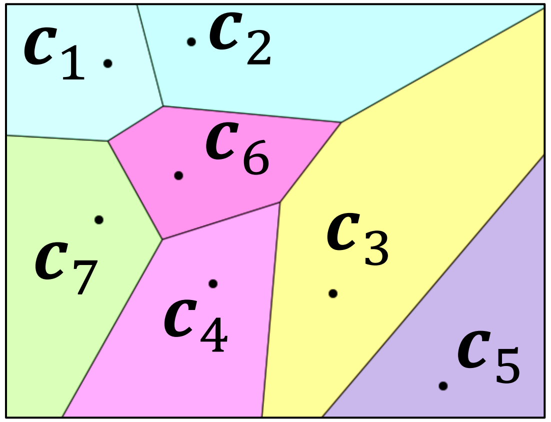
$\bar{\boldsymbol{x}} \in \{1, \ldots, 256\}^M$

- Suppose $\boldsymbol{q}, \boldsymbol{x} \in \mathbb{R}^D$, where $\boldsymbol{x}$ is quantized to $\bar{\boldsymbol{x}}$
- $d(\boldsymbol{q}, \boldsymbol{x})^2$ can be efficiently approximated by $\bar{\boldsymbol{x}}$:

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

Just by a PQ-code.
Not the original vector $\boldsymbol{x}$

Inverted index + PQ: Record



Coarse quantizer

$k = 1$

$k = 2$

$\vdots$

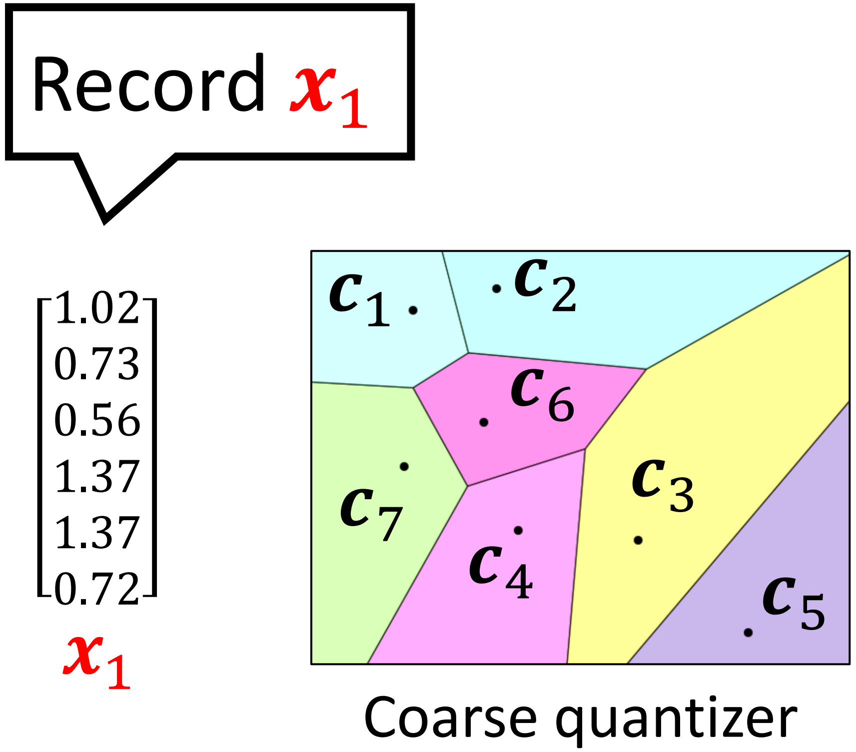
$k = K$

Prepare a **coarse quantizer**

✓ Split the space into K sub-spaces

✓ $\{\boldsymbol{c}_k\}_{k=1}^{K}$ are created by running k-means on training data

Inverted index + PQ: Record



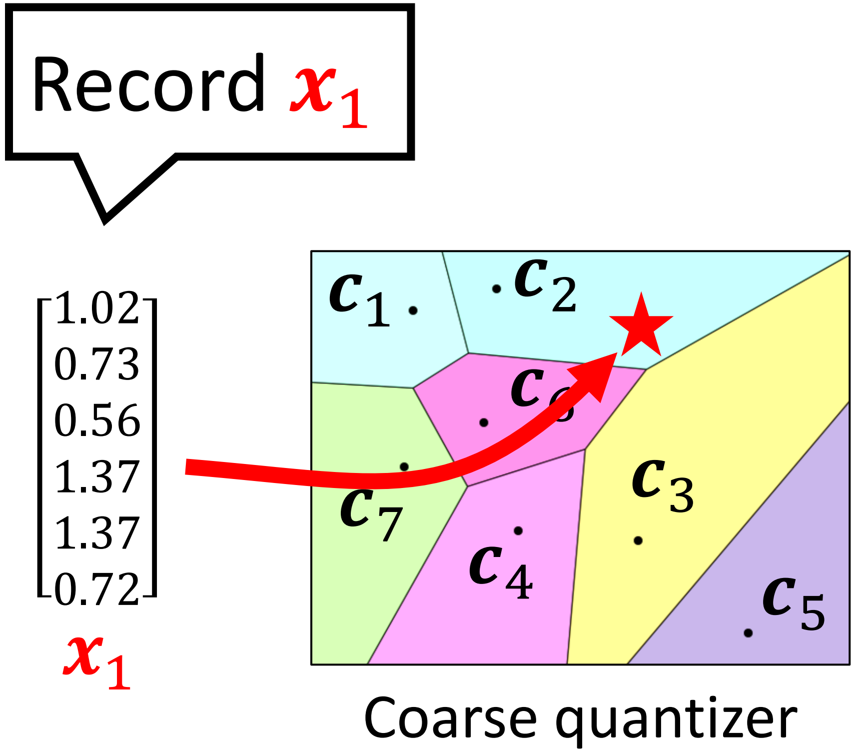
$k = 1$

$k = 2$

$\vdots$

$k = K$

Inverted index + PQ: Record



$k = 1$

$k = 2$

$\vdots$

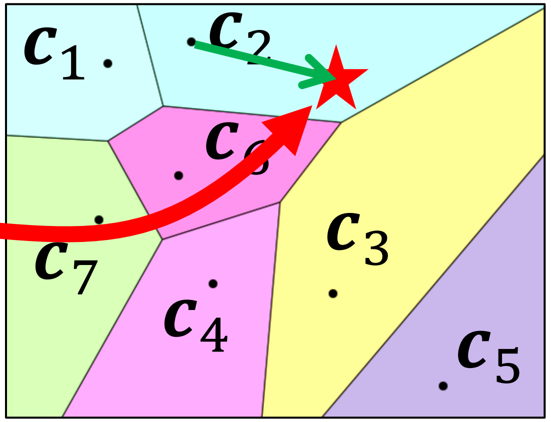
$k = K$

Inverted index + PQ: Record

Record $\boldsymbol{x}_1$

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

$\boldsymbol{x}_1$



Coarse quantizer

$k = 1$

$k = 2$

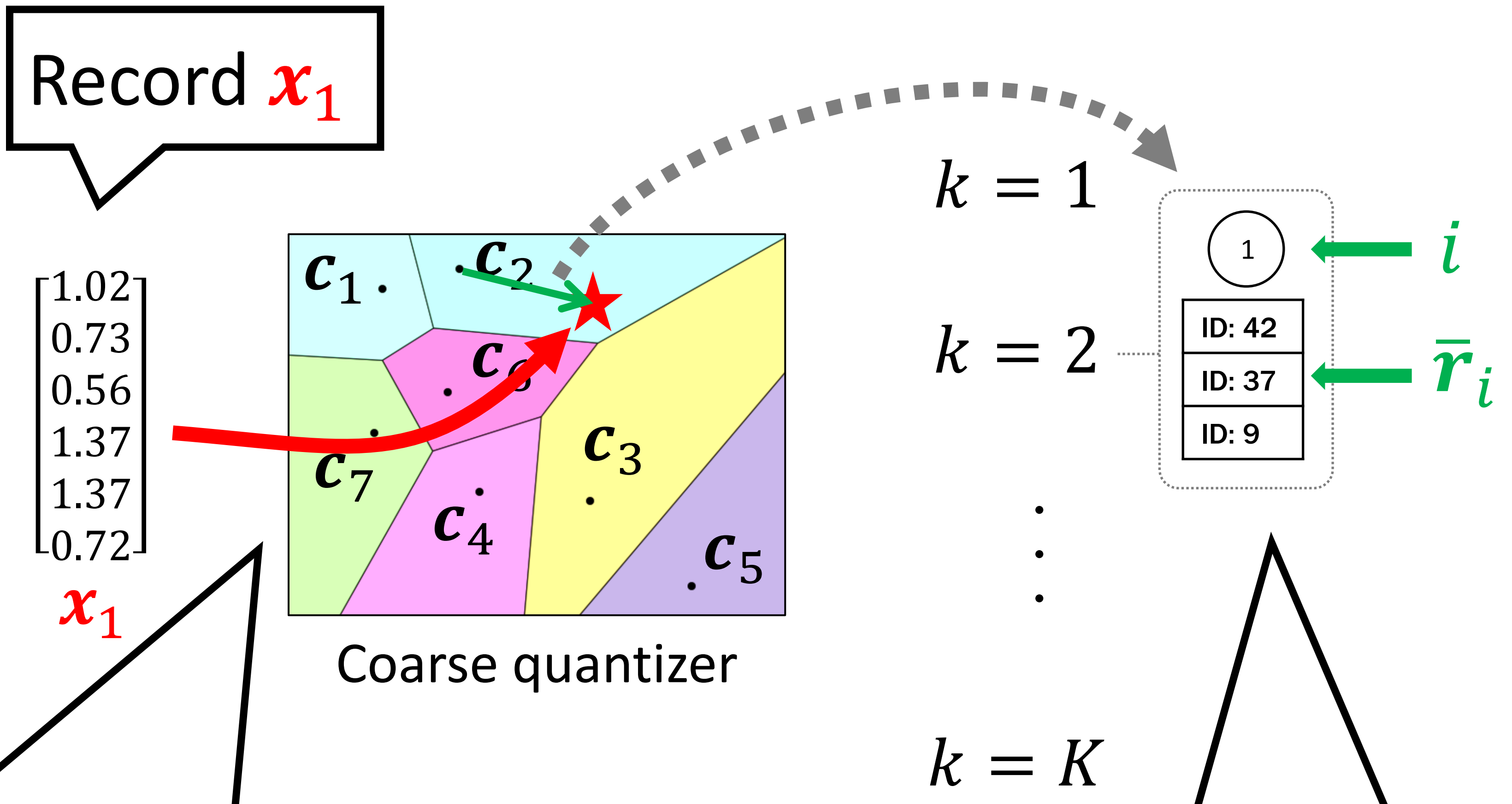
$\vdots$

$k = K$

- $\boldsymbol{c}_2$ is closest to $\boldsymbol{x}_1$
- Compute a **residual** r_1 between $\boldsymbol{x}_1$ and $\boldsymbol{c}_2$:

  $\boldsymbol{r}_1 = \boldsymbol{x}_1 - \boldsymbol{c}_2$ ($\rightarrow$)

Inverted index + PQ: Record

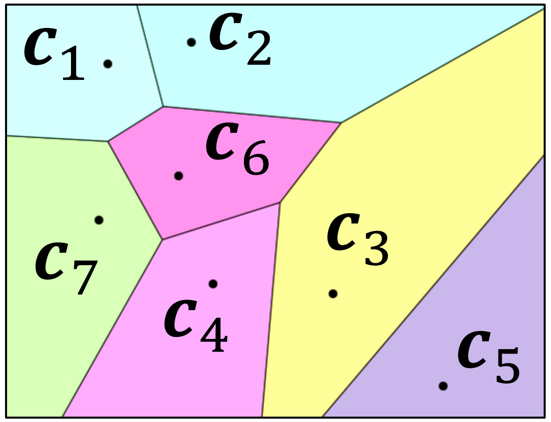


- $\boldsymbol{c}_2$ is closest to $\boldsymbol{x}_1$
- Compute a **residual** r_1 between $\boldsymbol{x}_1$ and $\boldsymbol{c}_2$:
  $\boldsymbol{r}_1 = \boldsymbol{x}_1 - \boldsymbol{c}_2 \quad (\longrightarrow)$

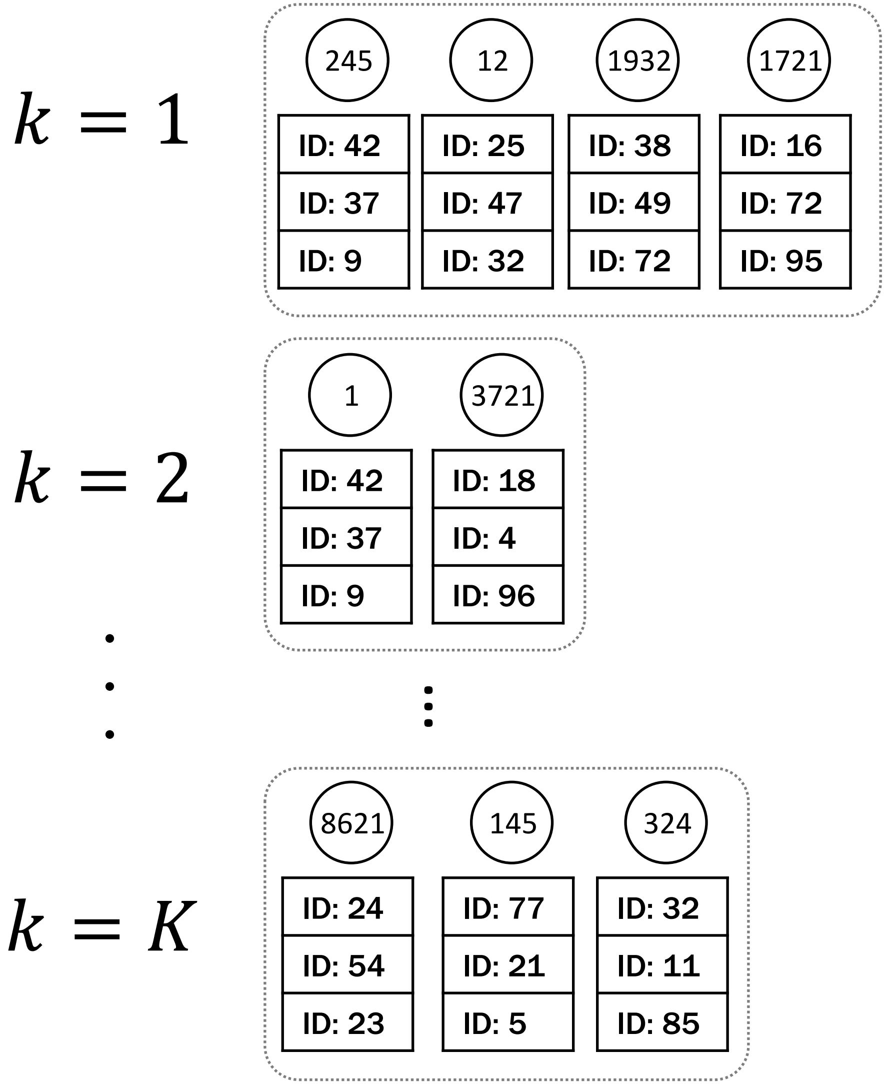
- Quantize $\boldsymbol{r}_1$ to $\bar{\boldsymbol{r}}_1$ by PQ
- Record it with the ID, "1"
- i.e., record $(i, \bar{\boldsymbol{r}}_i)$

Inverted index + PQ: Record

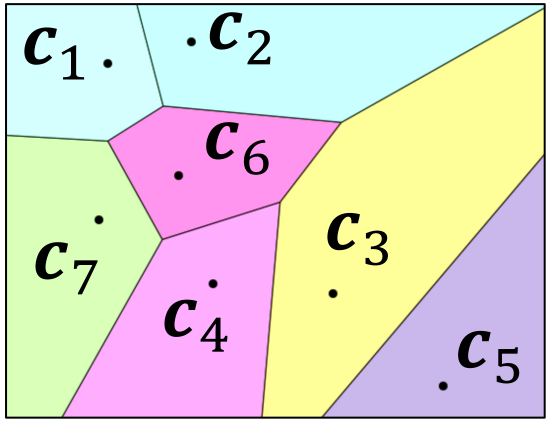
- For all database vectors, record [ID + PQ(res)] as pointing lists



coarse quantizer



Inverted index + PQ: Search



coarse quantizer

$k = 1$

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

$k = 2$

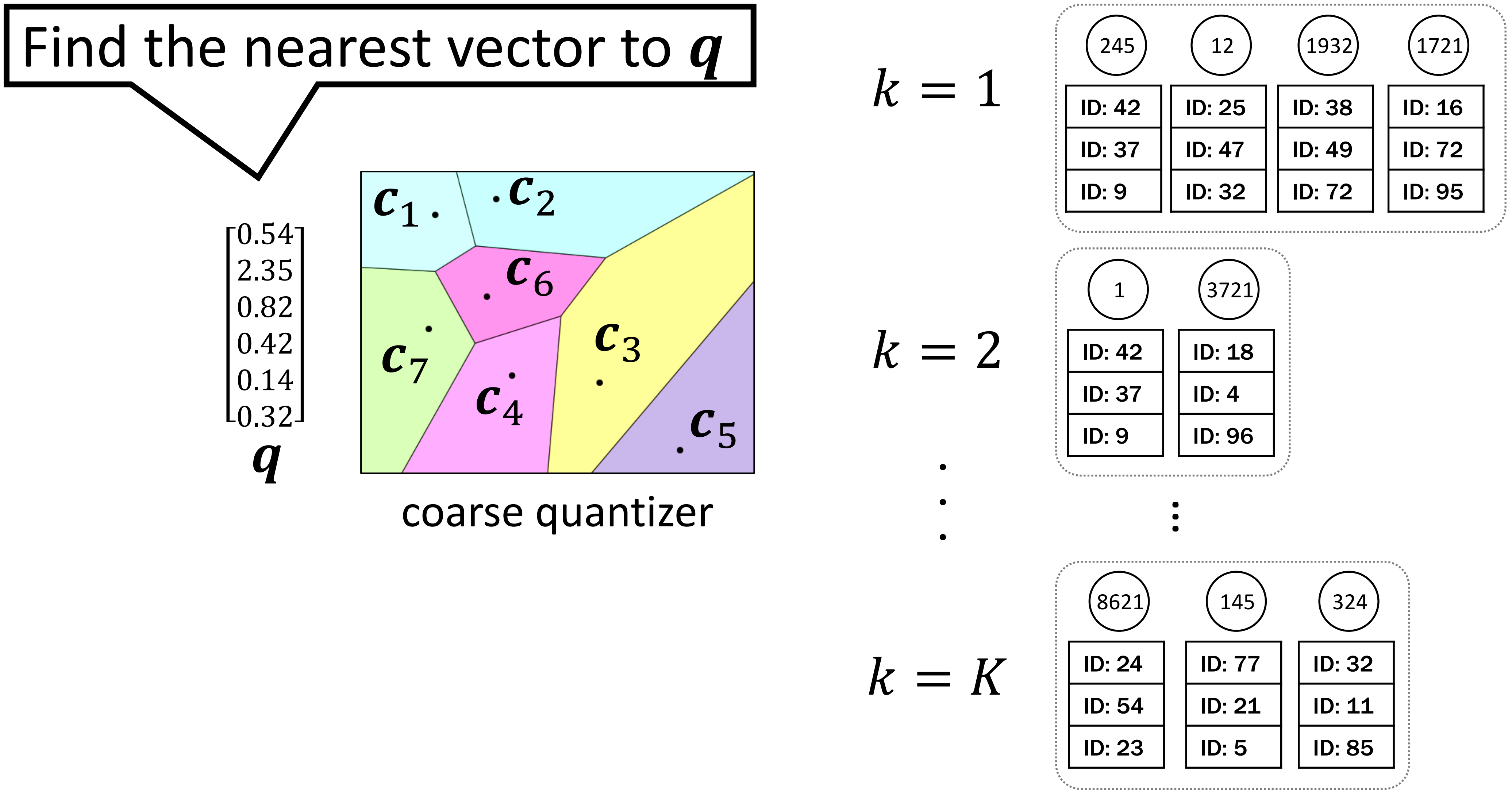
1	3721
ID: 42	ID: 18
ID: 37	ID: 4
ID: 9	ID: 96

$\vdots$

$k = K$

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

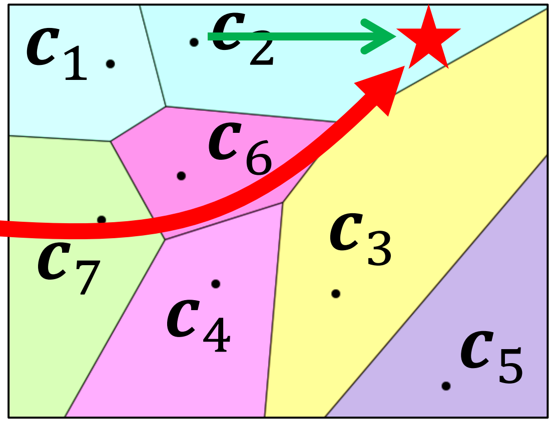
Inverted index + PQ: Search



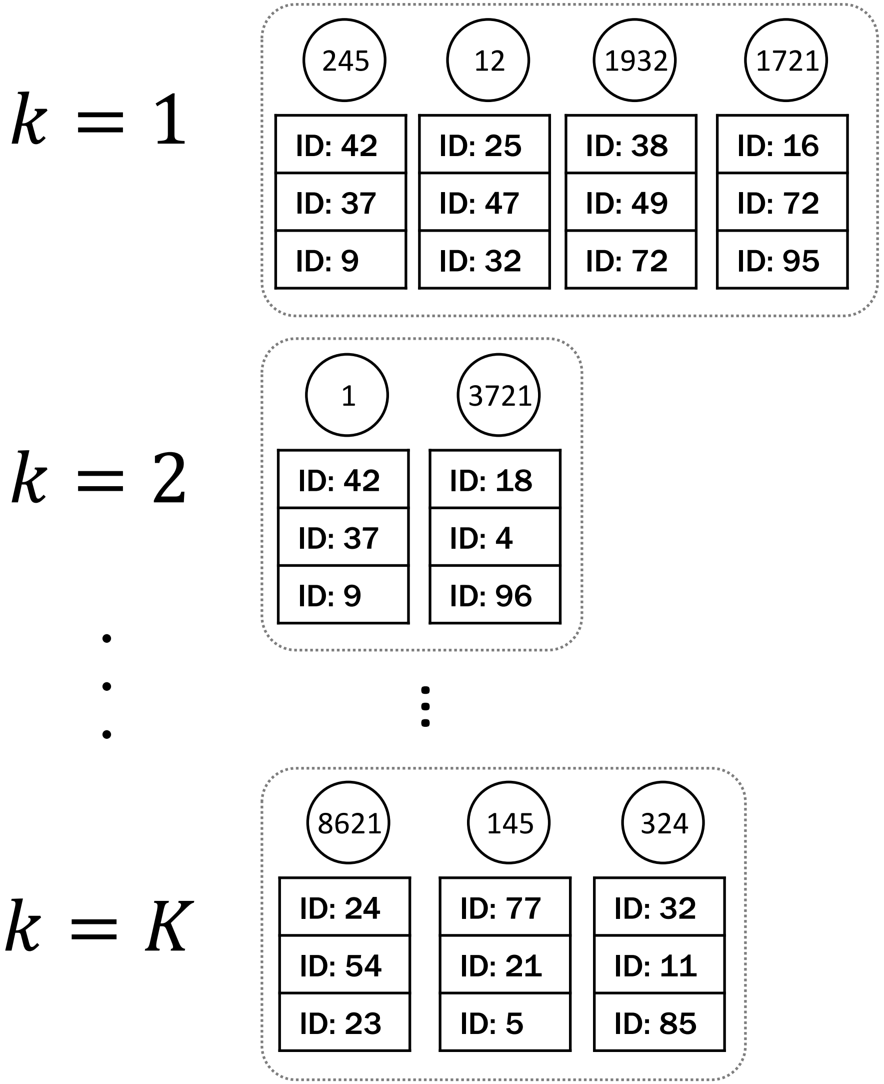
Inverted index + PQ: Search

Find the nearest vector to $\boldsymbol{q}$

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



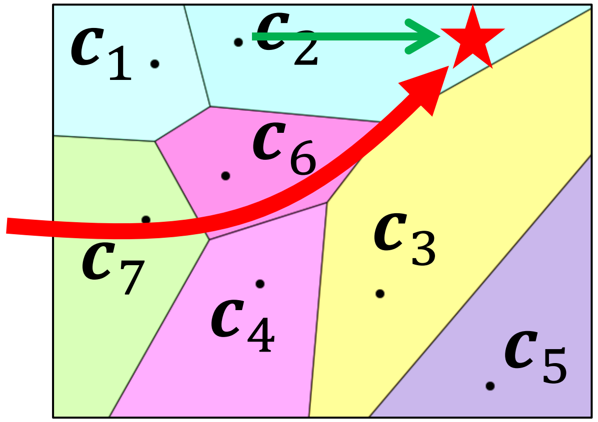
coarse quantizer



Inverted index + PQ: Search

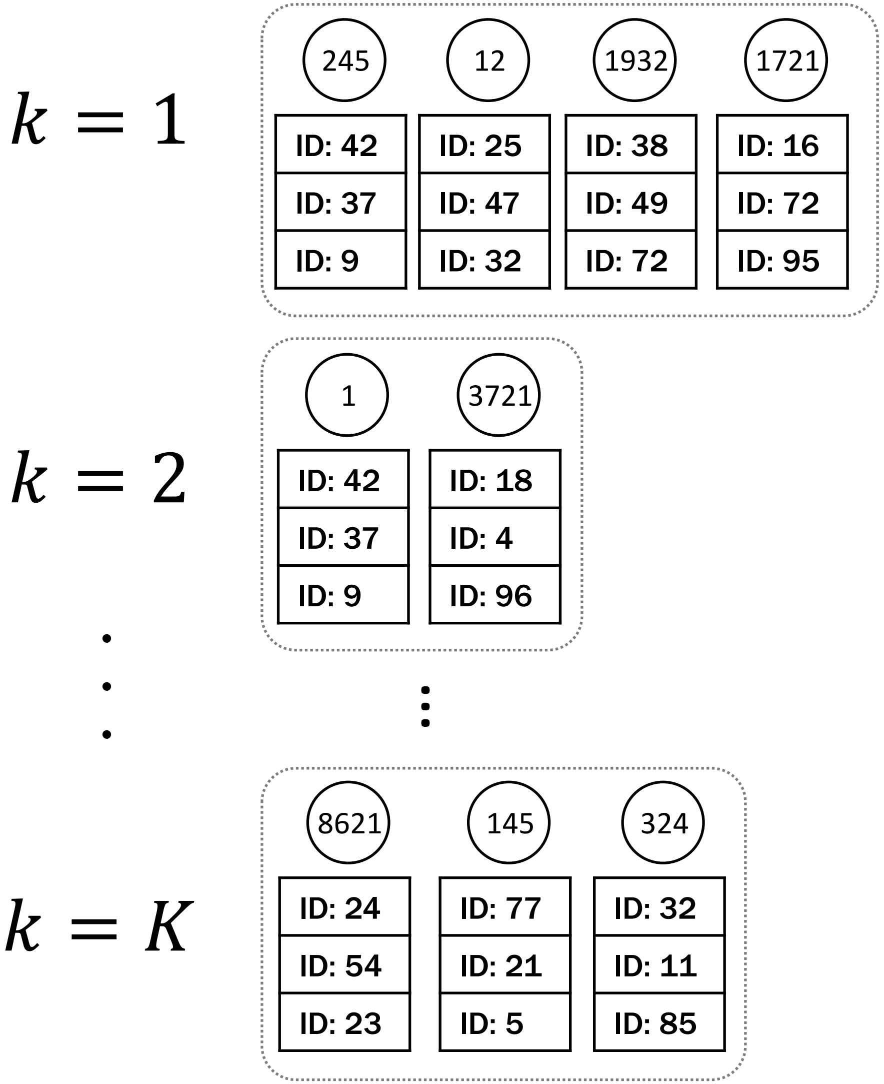
Find the nearest vector to $\boldsymbol{q}$

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



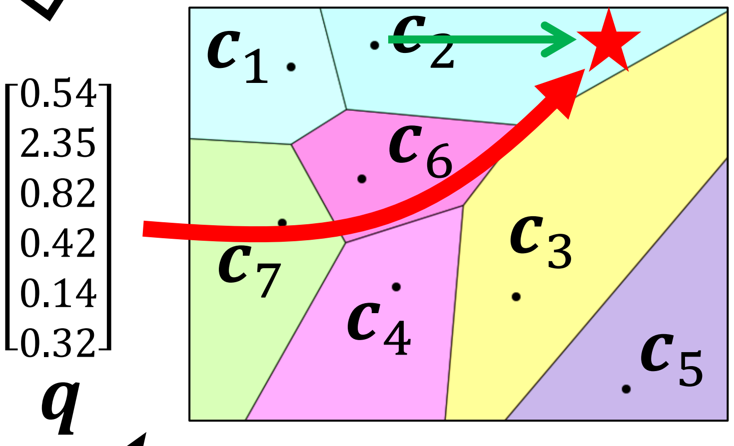
coarse quantizer

- $\boldsymbol{c}_2$ is the closest to $\boldsymbol{q}$
- Compute the residual: $\boldsymbol{r}_q = \boldsymbol{q} - \boldsymbol{c}_2$

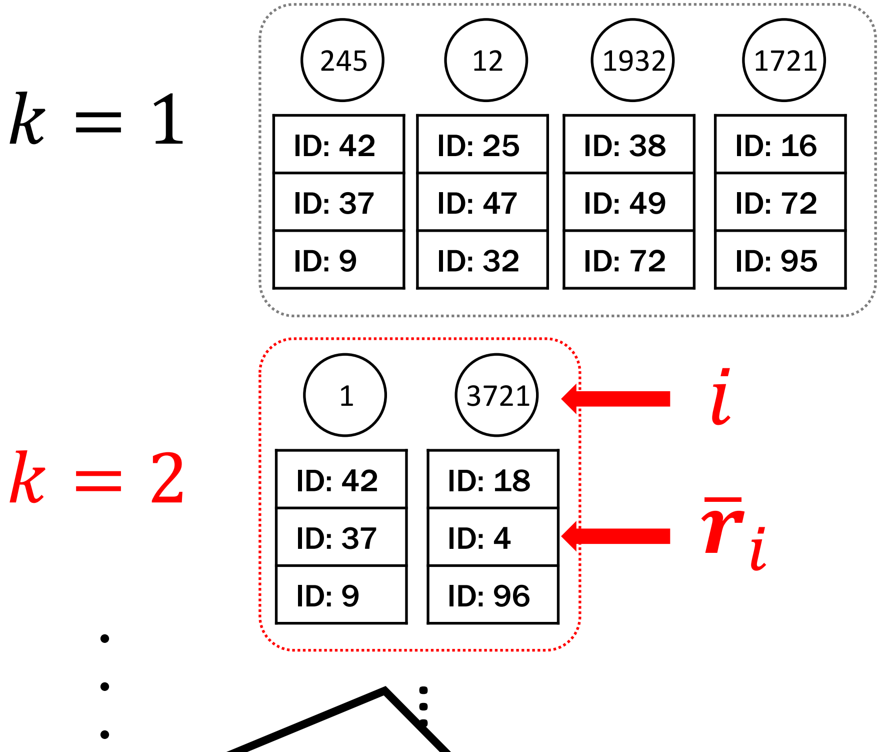


Inverted index + PQ: Search

Find the nearest vector to $\boldsymbol{q}$



coarse quantizer



- $\boldsymbol{c}_2$ is the closest to $\boldsymbol{q}$
- Compute the residual: $\boldsymbol{r}_q = \boldsymbol{q} - \boldsymbol{c}_2$

- For all $(i, \bar{\boldsymbol{r}}_i)$ in $k = 2$, compare $\bar{\boldsymbol{r}}_i$ with $\boldsymbol{r}_q$:

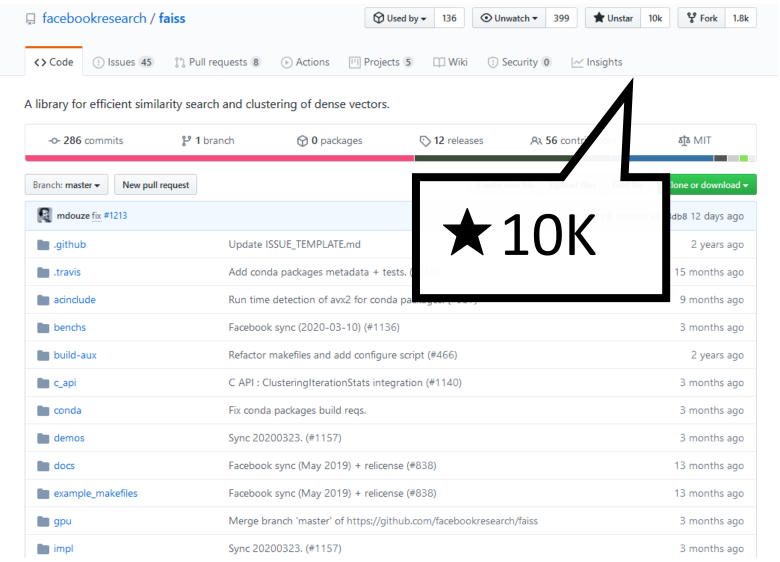
$$d(\boldsymbol{q}, \boldsymbol{x}_i)^2 = d(\boldsymbol{q} - \boldsymbol{c}_2, \boldsymbol{x}_i - \boldsymbol{c}_2)^2$$
$$= d\left(\boldsymbol{r}_q, \boldsymbol{r}_i\right)^2 \sim d_A\left(\boldsymbol{r}_q, \bar{\boldsymbol{r}}_i\right)^2$$

- Find the smallest one (several strategies)

Faiss

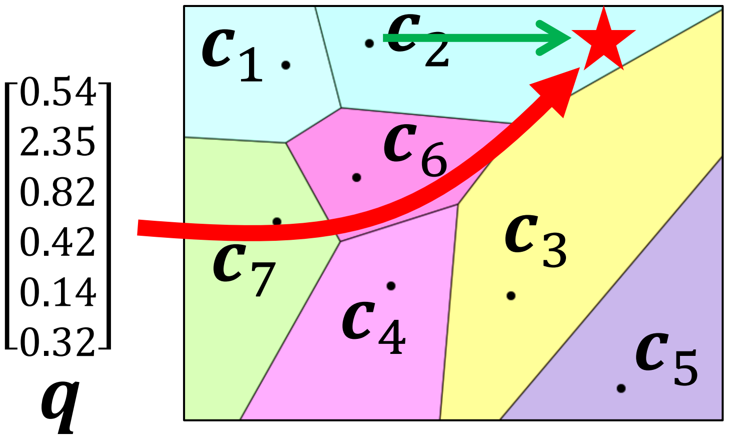
https://github.com/facebookresearch/faiss

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

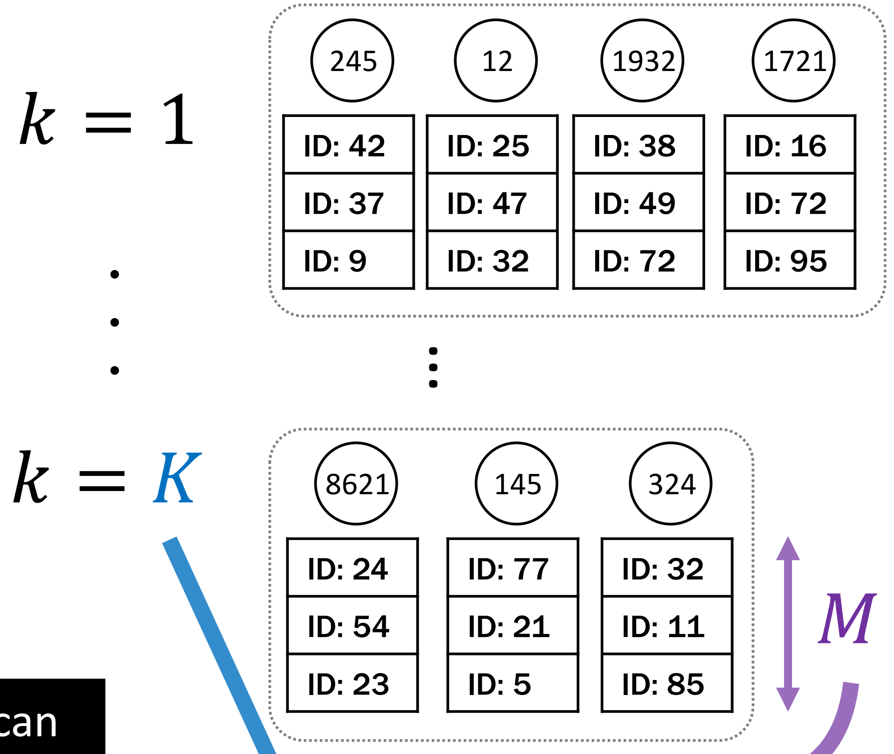


- ➢From the original authors of the PQ and a GPU expert, FAIR
- ➢CPU-version: all PQ-based methods
- ➢GPU-version: some PQ-based methods
- ➢Bonus:
    - ➢NN (not ANN) is also implemented, and quite fast
    - ➢k-means (CPU/GPU). Fast.

Benchmark of k-means:
https://github.com/DwangoMediaVillage/pqkmeans/blob/master/tutorial/4_comparison_to_faiss.ipynb



coarse quantizer



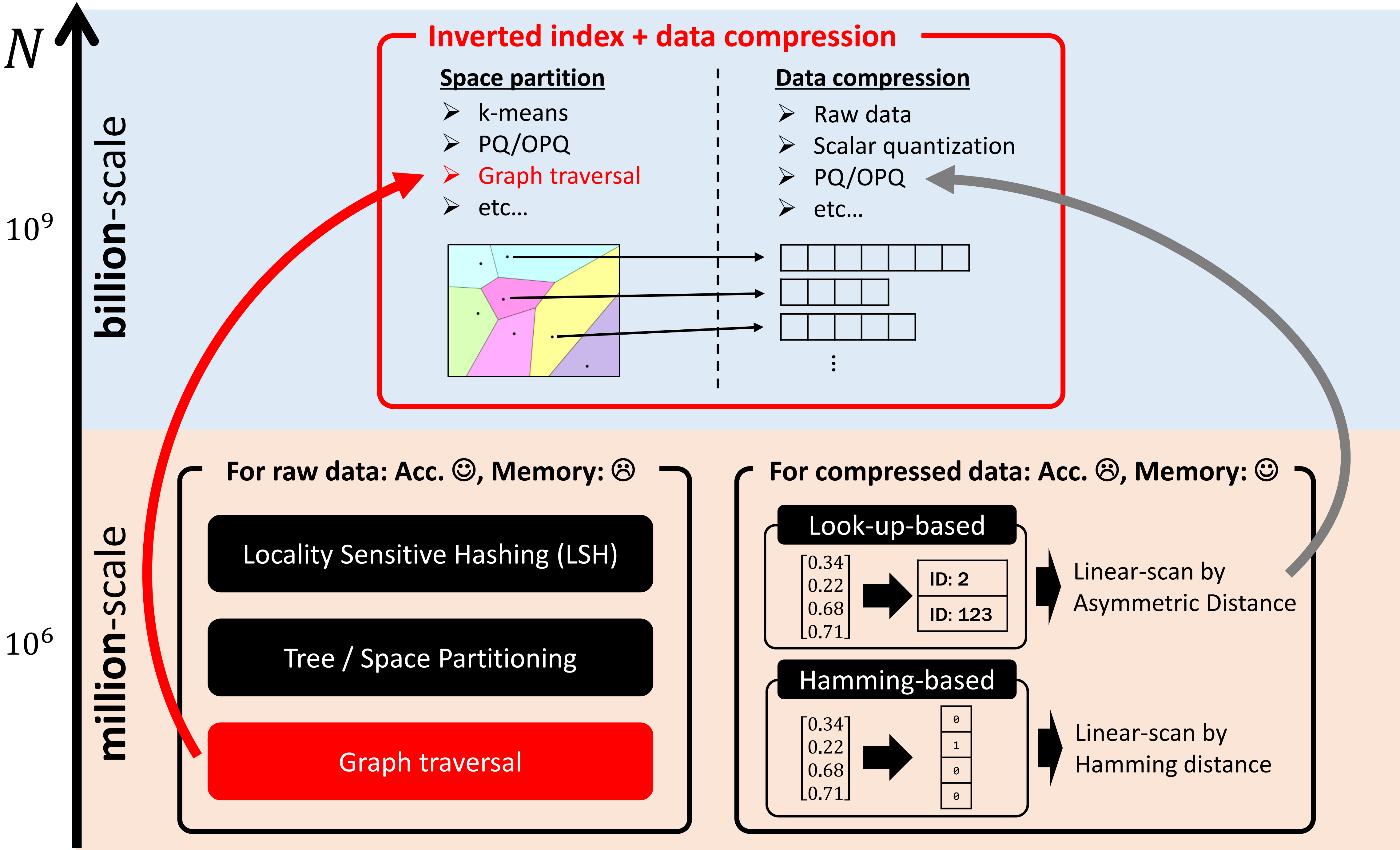
Simple linear scan

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

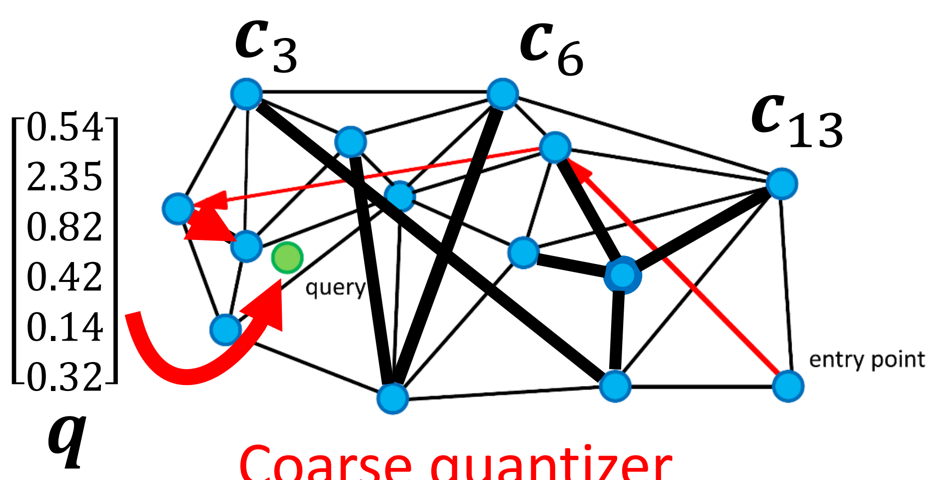
Select a coarse quantizer

Usually, 8 bit

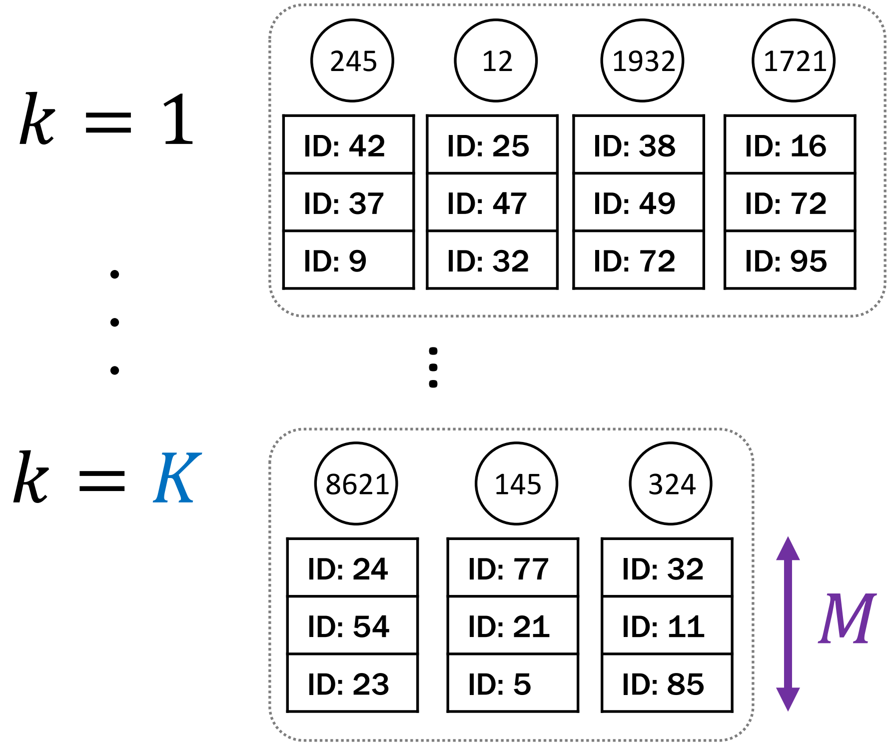
```
index.train(Xt)  # Train
index.add(X)  # Record data
index.nprobe = nprobe  # Search parameter
dist, ids = index.search(Q, topk)  # Search
```

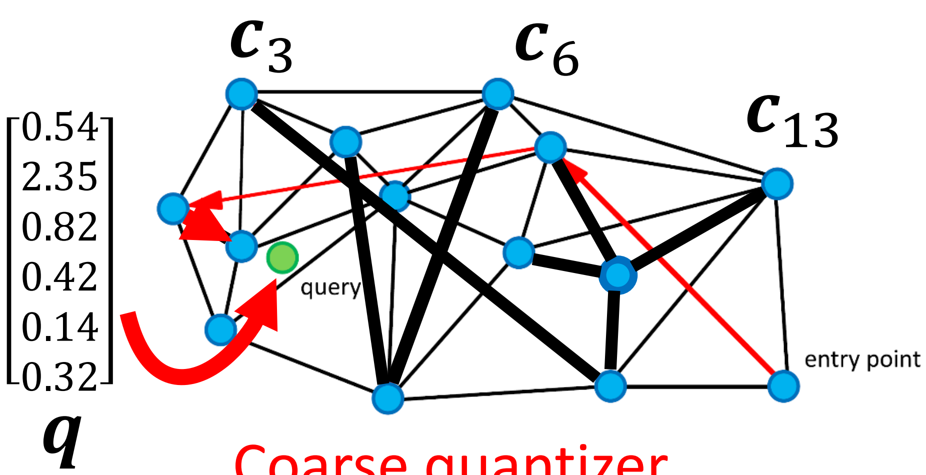


N
10^9
10^6
billion-scale
million-scale
Inverted index + data compression
Space partition
k-means
PQ/OPQ
Graph traversal
etc…
Data compression
Raw data
Scalar quantization
PQ/OPQ
etc…
For raw data: Acc. ☺, Memory: ☹
Locality Sensitive Hashing (LSH)
Tree / Space Partitioning
Graph traversal
For compressed data: Acc. ☹, Memory: ☺
Look-up-based
[0.34 0.22 0.68 0.71]
ID: 2
ID: 123
Linear-scan by Asymmetric Distance
Hamming-based
[0.34 0.22 0.68 0.71]
0 1 0 0
Linear-scan by Hamming distance

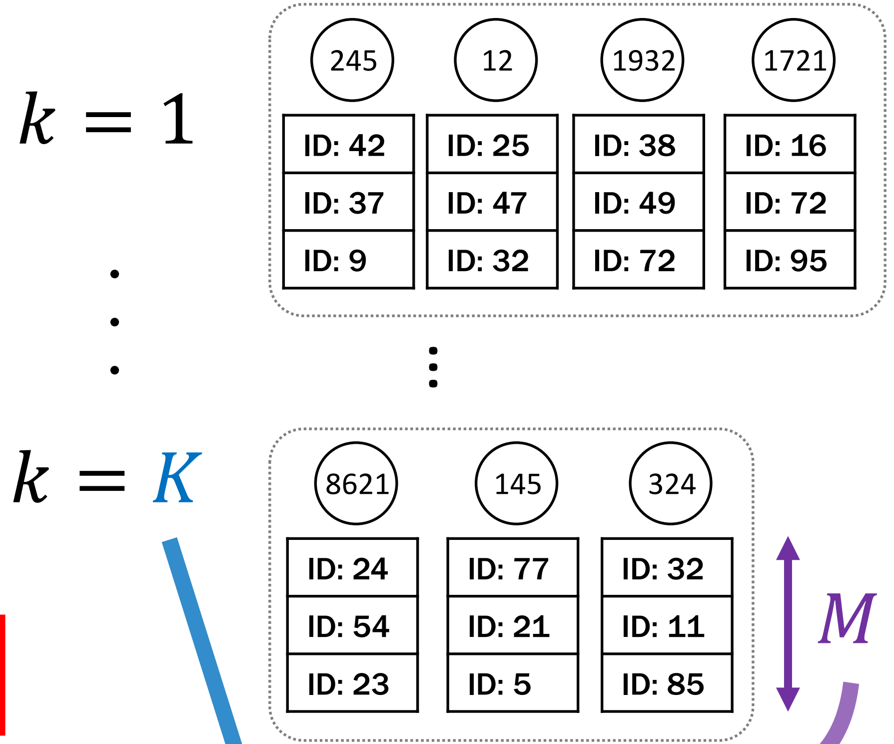


Coarse quantizer





Coarse quantizer



HNSW

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

Select a coarse quantizer

Usually, 8 bit

- Switch a coarse quantizer from linear-scan to HNSW
- The best approach for billion-scale data as of 2020
- The backbone of [Douze+, CVPR 2018] [Baranchuk+, ECCV 2018]

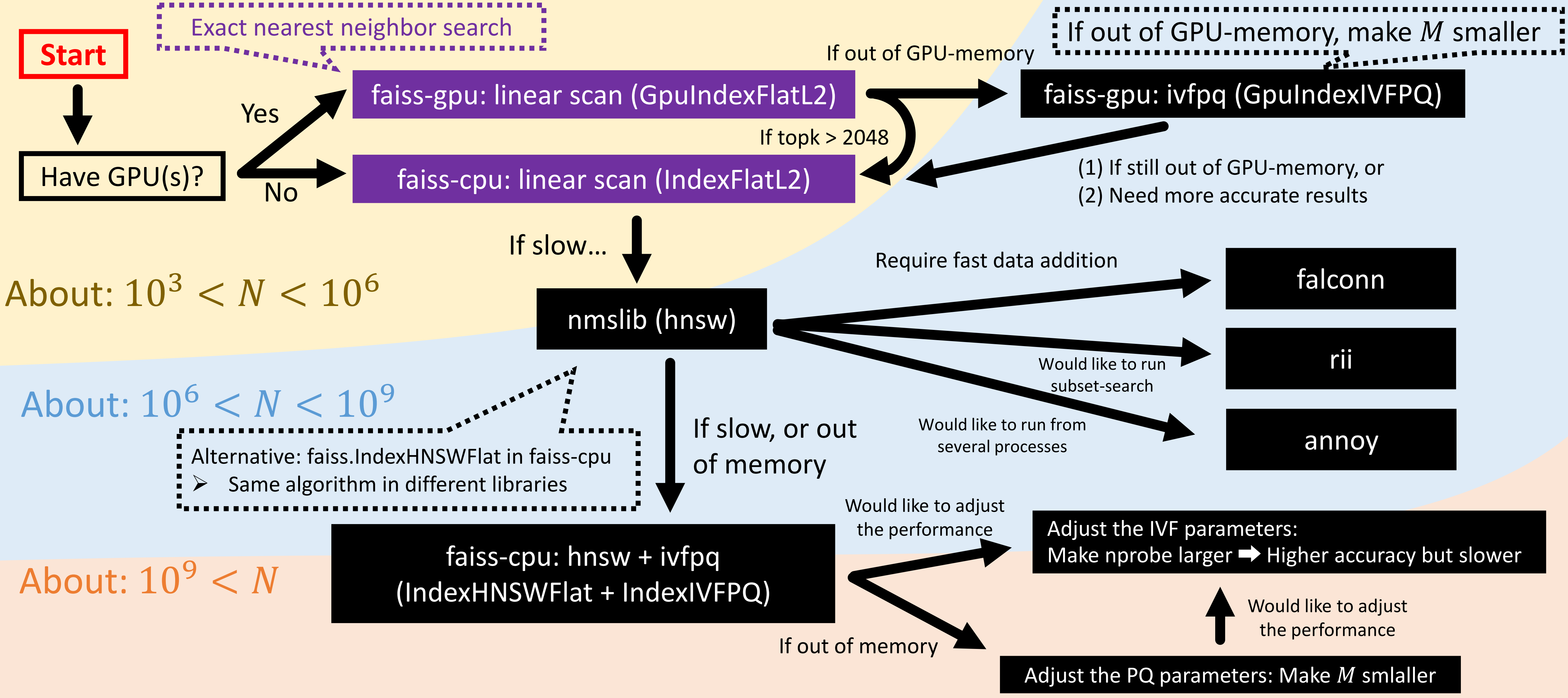
☺ From the original authors of PQ. Extremely efficient (theory & impl)
☺ Used in a real-world product (Mercari, etc)
☺ For billion-scale data, Faiss is the best option
☺ Especially, large-batch-search is fast; #query is large

☹ Lack of documentation (especially, python binding)
☹ Hard for a novice user to select a suitable algorithm
☹ As of 2020, anaconda is required. Pip is not supported officially

Reference

- Faiss wiki: [https://github.com/facebookresearch/faiss/wiki]
- Faiss tips: [https://github.com/matsui528/faiss_tips]
- Julia implementation of lookup-based methods [https://github.com/una-dinosauria/Rayuela.jl]
- PQ paper: 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

cheat-sheet for ANN in Python （as of 2020. Can be installed by conda or pip）

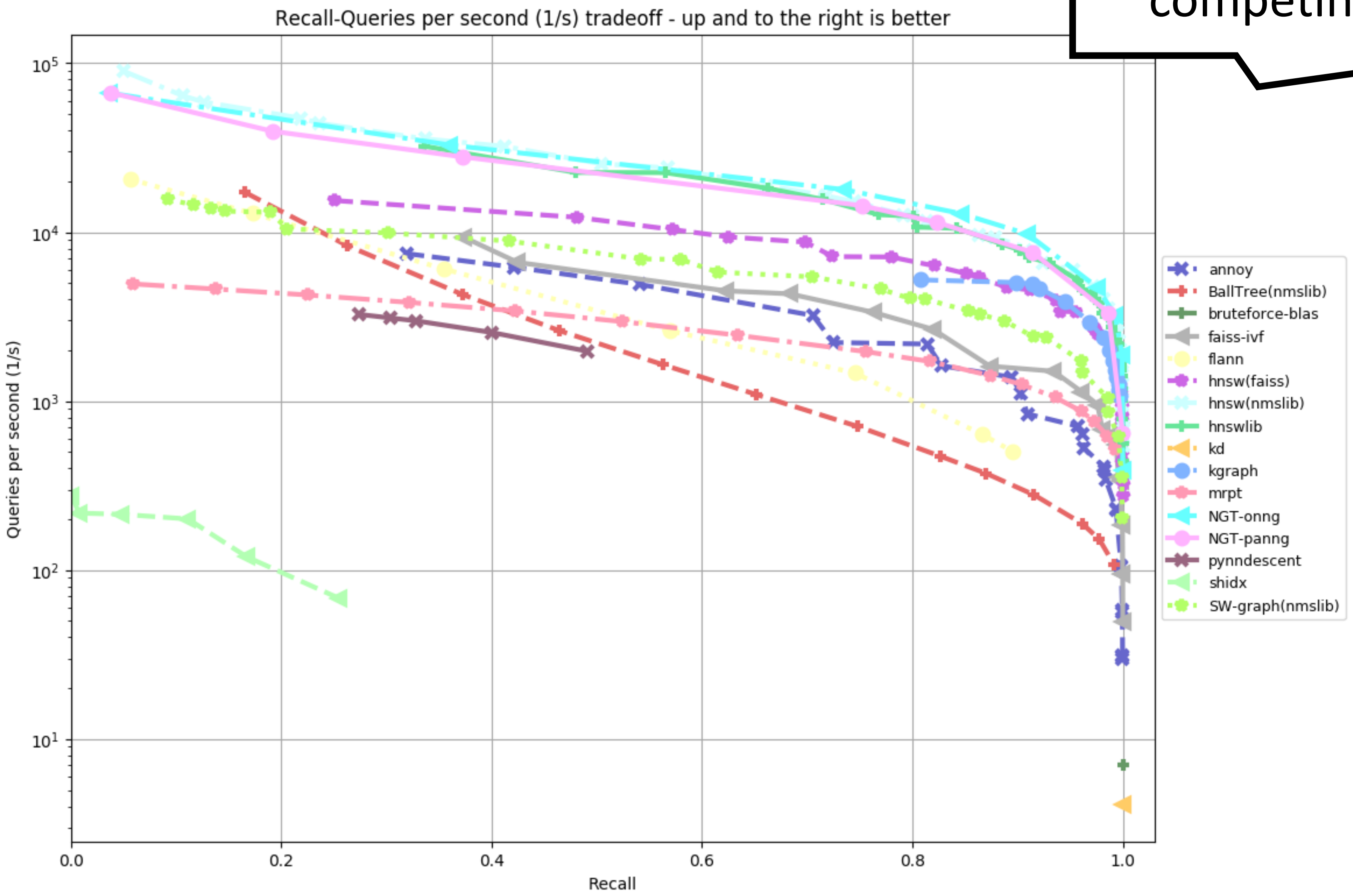


Note: Assuming $D \cong 100$. The size of the problem is determined by DN. If $100 \ll D$, run PCA to reduce D to 100

Benchmark 1: ann-benchmarks

- https://github.com/erikbern/ann-benchmarks
- Comprehensive and thorough benchmarks for various libraries. Docker-based

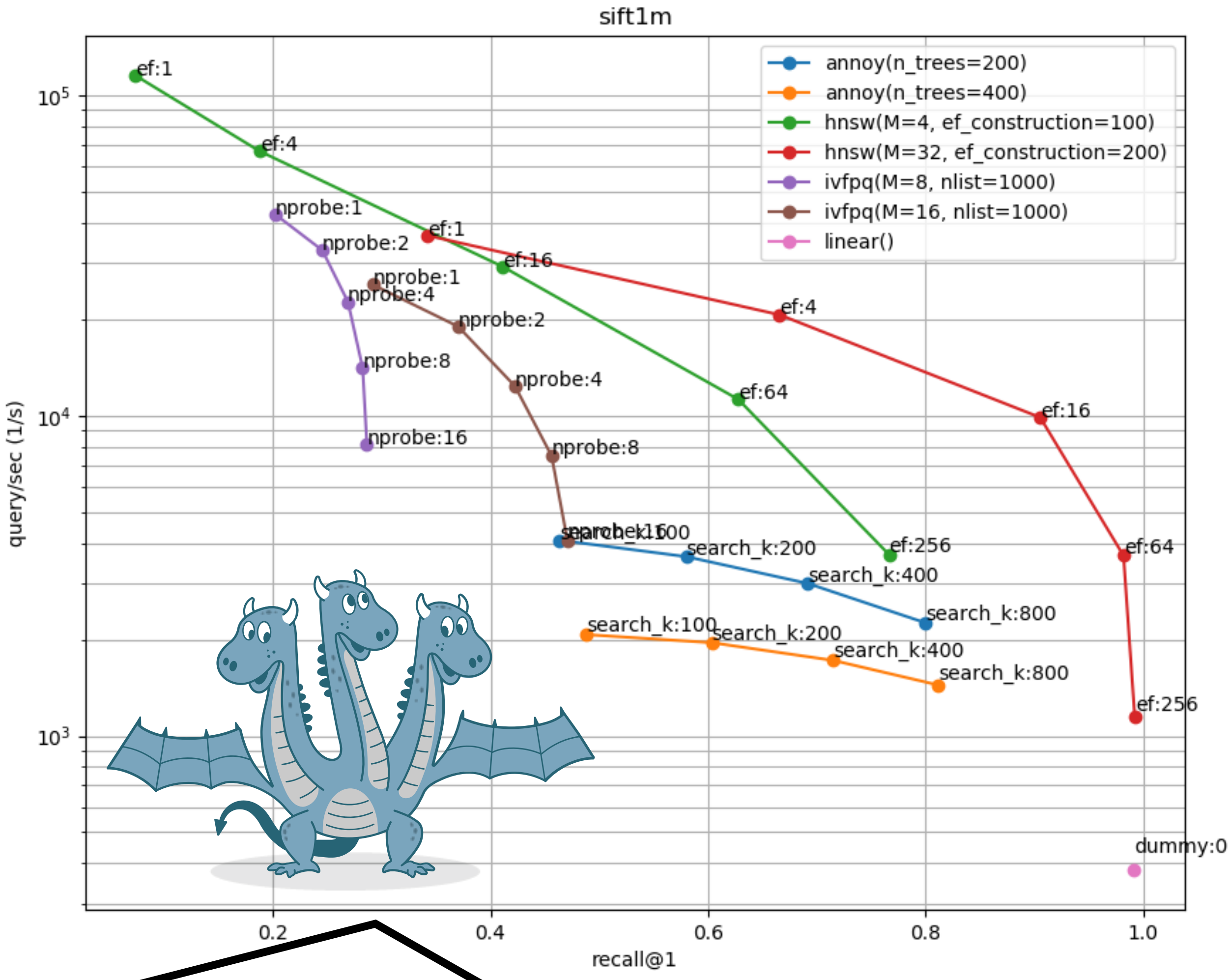
- Top right, the better
- As of June, 2020, NMSLIB and NGT are competing each other for the first place



Benchmark 2: annbench

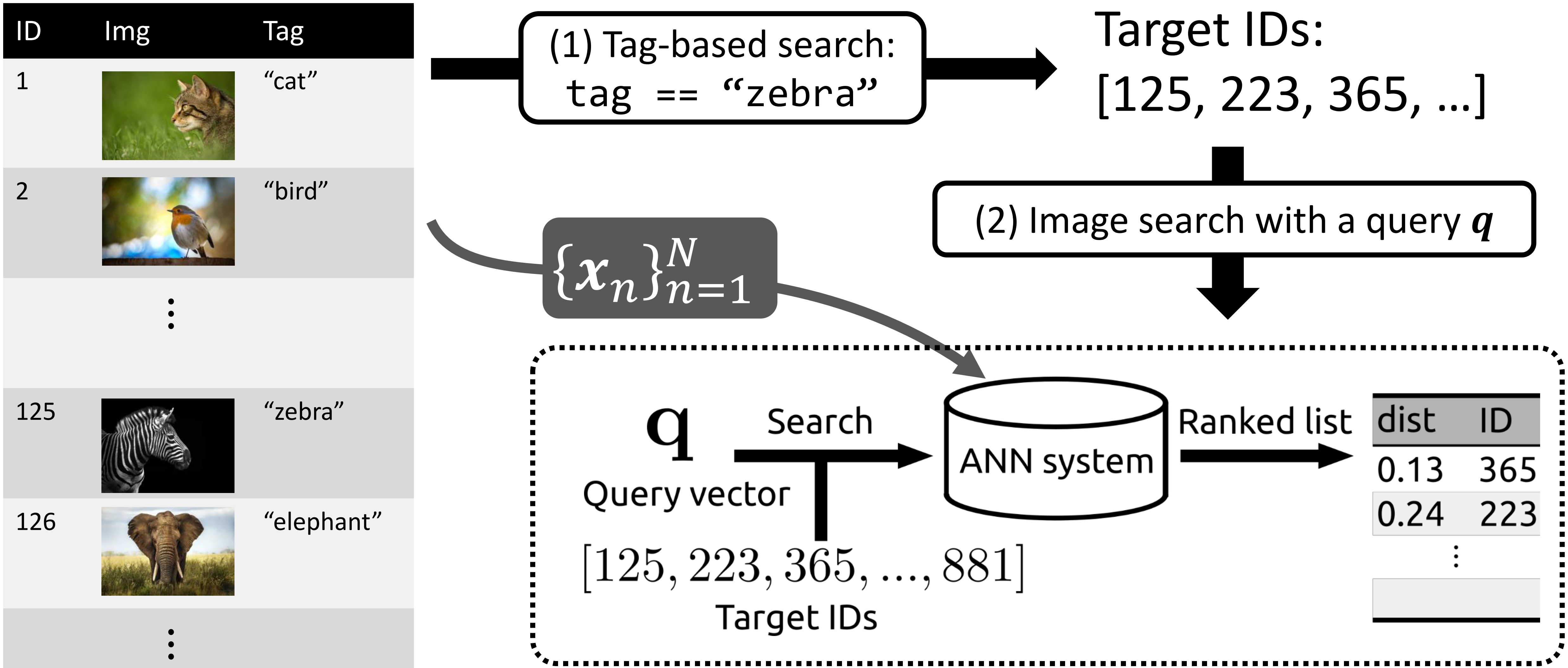
- https://github.com/matsui528/annbench
- Lightweight, easy-to-use

```
# Install libraries
pip install -r requirements.txt

# Download dataset on ./dataset
python download.py dataset=siftsmall

# Evaluate algos. Results are on ./output
python run.py dataset=siftsmall algo=annoy

# Visualize
python plot.py
```




```
# Multi-run by Hydra
python run.py --multirun dataset=siftsmall,sift1m algo=linear,annoy,ivfpq,hnsw
```

Search for a "subset"



Subset-search

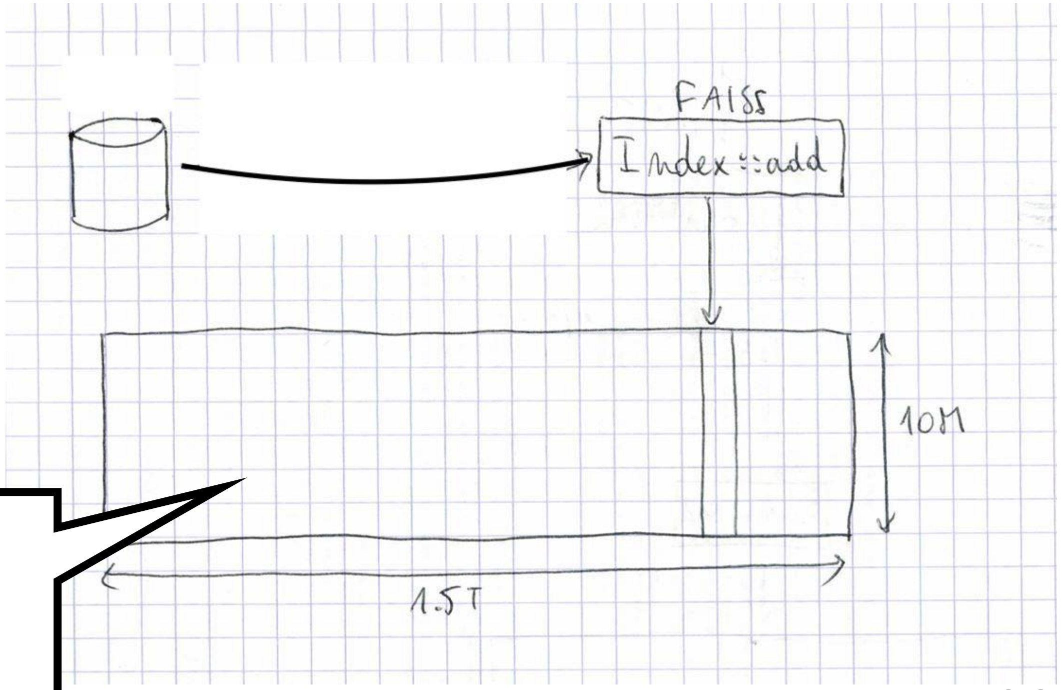
Y. Matsui+, "Reconfigurable Inverted Index", ACMMM 18

Trillion-scale search: $N = 10^{12}$ (1T)

Sense of scale

- K($= 10^3$) Just in a second on a local machine
- M($= 10^6$) All data can be on memory. Try several approaches
- G($= 10^9$) Need to compress data by PQ. Only two datasets are available (SIFT1B, Deep1B)
- T($= 10^{12}$) Cannot even imagine

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



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

- Only in Faiss wiki
- Distributed, mmap, etc.

A sparse matrix of 15 Exa elements?

Nearest neighbor search engine: something like **ANN + SQL**

- The algorithm inside is faiss, nmslib, or NGT

VEARCH

https://github.com/vearch/vearch

Open Distro for Elasticsearch

Elasticsearch KNN

https://github.com/opendistro-for-elasticsearch/k-NN

Milvus

https://github.com/milvus-io/milvus

Vald

https://github.com/vdaas/vald

Problems of ANN

- No mathematical background.
  - ✓ Only actual measurements matter: recall and runtime
  - ✓ The ANN problem was mathematically defined 10+ years ago (LSH), but recently no one cares the definition.

- Thus, when the score is high, it's not clear the reason:
  - ✓ The method is good?
  - ✓ The implementation is good?
  - ✓ Just happens to work well for the target dataset?
  - ✓ E.g.: The difference of math library (OpenBLAS vs Intel MKL) matters.

- If one can explain "why this approach works good for this dataset", it would be a great contribution to the field.

- Not enough dataset. Currently, only two datasets are available for billion-scale data: SIFT1B and Deep1B