



Векторные поиски: опыт внедрения и эксплуатации в Sphinx



Михаил Самолкаев

Backend-разработчик в Search Infra

- Разрабатываю **Sphinx** – СУБД, ориентированную на полнотекстовый поиск
- Команда отвечает за инфраструктуру поиска в Авито
- Мой tg: **@msmlkm**
- Подготовить доклад мне помогли:
 - Андрей Аксенов **@shodanium**
 - Никита Петтик



слайды



План выступления

Обзорная часть (алгоритмы, техники, готовые решения)	01
Опыт внедрения и эксплуатации в Sphinx	02
Дальнейшие направления работы	03
Заключение, Q/A	04

Обзорная часть

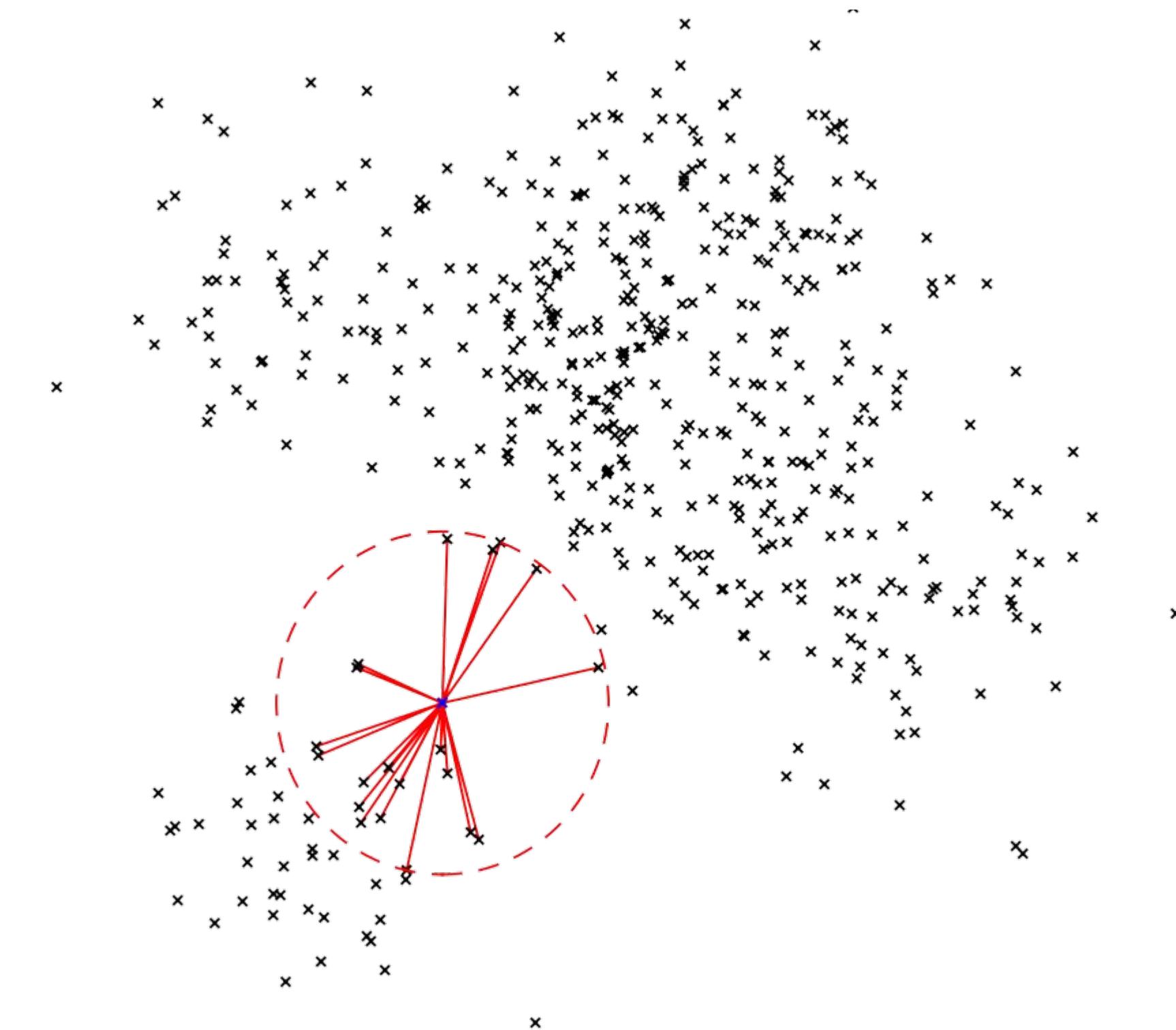
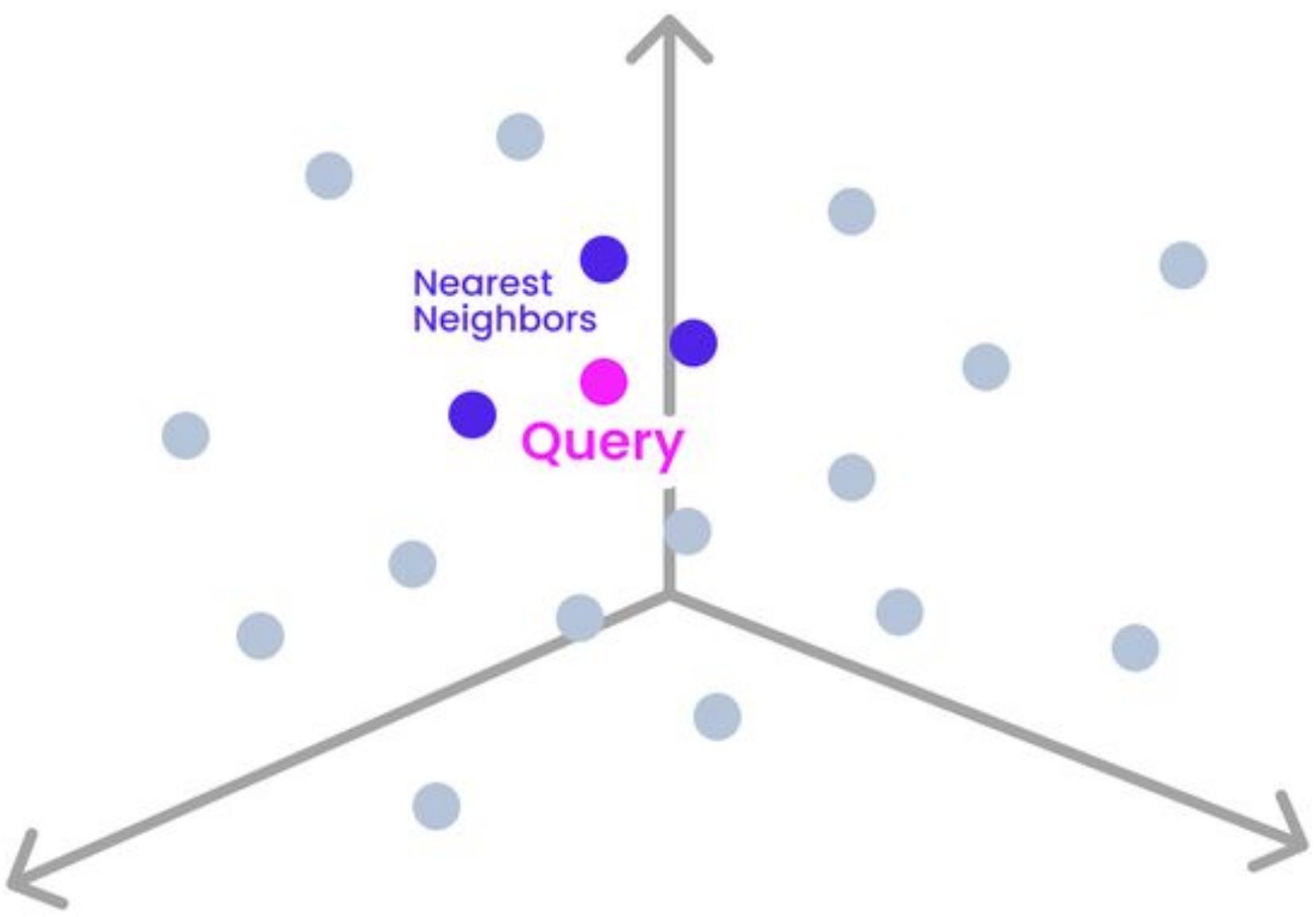
4

2

3

1

Что такое векторные поиски?



Что такое векторные поиски?

Три строки на стародатском

```
Vector<Match> VectorSearch(Vector<float> q, int top) {
    MinHeap<Match> results(top); // keeps top-K biggest dots
    for (int i = 0; i < vectors_.size(); i++) {
        float dist = 0.0f;
        for (int j = 0; j < dimensions_; j++)
            dist += vectors_[i][j] * q[j]; // for example, DOT()
        results.PushHeap({ i, dist });
    }
    return results;
} // really? really.
```

Что такое векторные поиски?

Постановка задачи

- имеем:
 - множество **X** из **M** векторов размерности **d**;
 - вектор-запрос **q** размерности **d**;
- хотим: получить **k** ближайших к вектору **q** из

множества **X**, в смысле метрики расстояния **dist**.

$$X = \{x_1, x_2, \dots, x_{M-1}, x_M\}, x_i \in \mathbb{R}^d$$

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

$$n = (n_1, n_2, \dots, n_{k-1}, n_k)$$

$$n_1 = \underset{x_i \in X}{\operatorname{argmin}} dist(q, x_i)$$

$$n_2 = \underset{x_i \in X \setminus x_{n_1}}{\operatorname{argmin}} dist(q, x_i)$$

...

Что такое векторные поиски?

Постановка задачи

- имеем:
 - множество **X** из **M** векторов размерности **d**;
 - вектор-запрос **q** размерности **d**.
- хотим: получить **k** ближайших к вектору **q** из множества **X**, в смысле метрики расстояния **dist**;
- **dist** можно взять разный, в зависимости от исходной задачи.

$$\text{dist} = \left\{ \begin{array}{l} L1(x, y) = \sum_{i=1}^d |x_i - y_i| \\ L2(x, y) = \sum_{i=1}^d (x_i - y_i)^2 \\ \text{dot}(x, y) = \sum_{i=1}^d x_i y_i \\ \dots \end{array} \right.$$

Что такое векторные поиски?

Постановка задачи

- задача известна как **kNN** (a.k.a k Nearest Neighbors);
- идея: иногда нам достаточно искать **приближенно**;
- задача приближенного поиска известна как **aNN**
(a.k.a. approximate Nearest Neighbors)
- в aNN важна **полнота** (recall): сколько из
действительно ближайших векторов смогли найти.

$$X = \{x_1, x_2, \dots, x_{M-1}, x_M\}, x_i \in \mathbb{R}^d$$

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

$$n = (n_1, n_2, \dots, n_{k-1}, n_k)$$

$$n_1 = \underset{x_i \in X}{\operatorname{argmin}} dist(q, x_i)$$

$$n_2 = \underset{x_i \in X \setminus x_{n_1}}{\operatorname{argmin}} dist(q, x_i)$$

...

$$\text{recall}(n, n') = \frac{|n \cap n'|}{|n|}$$

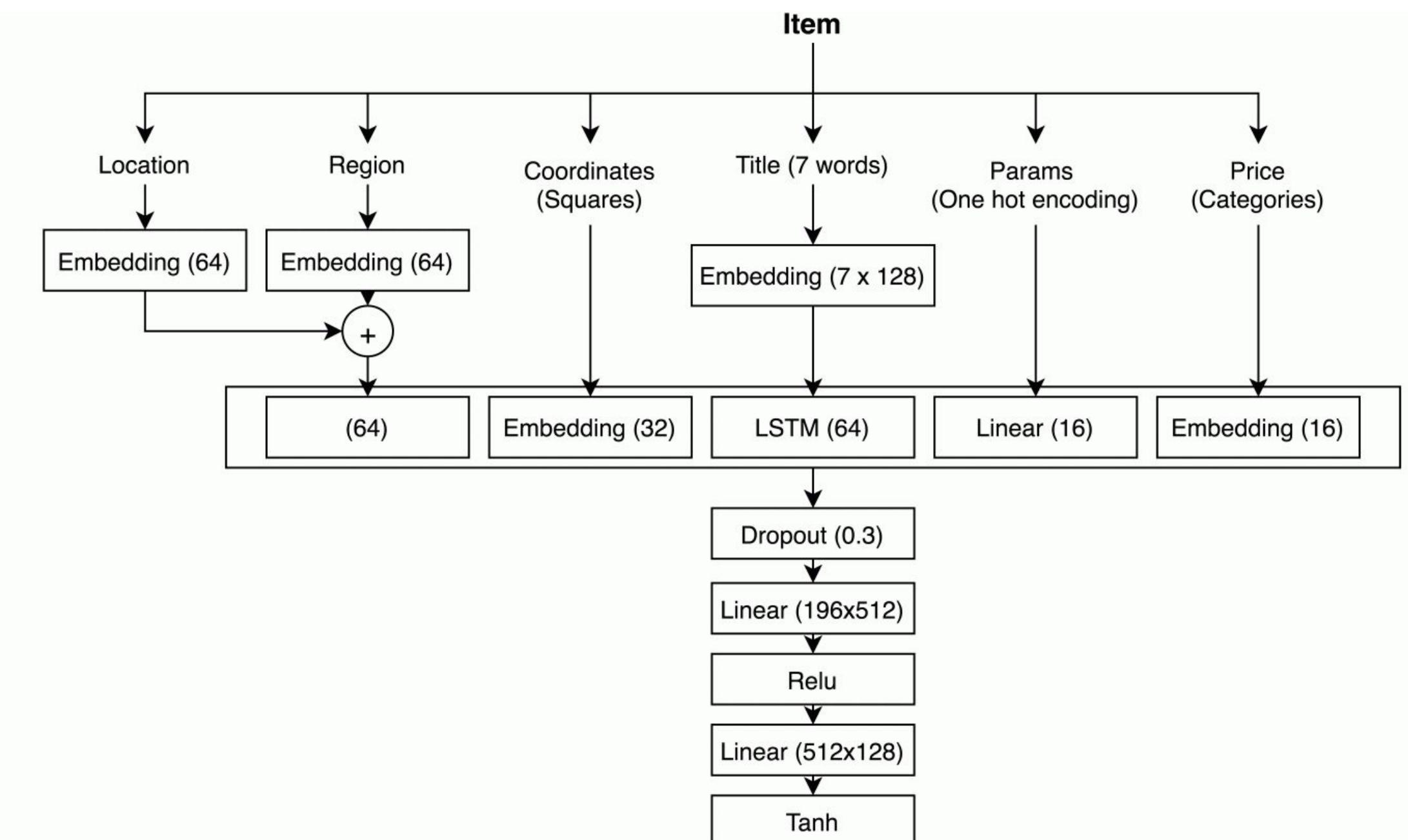
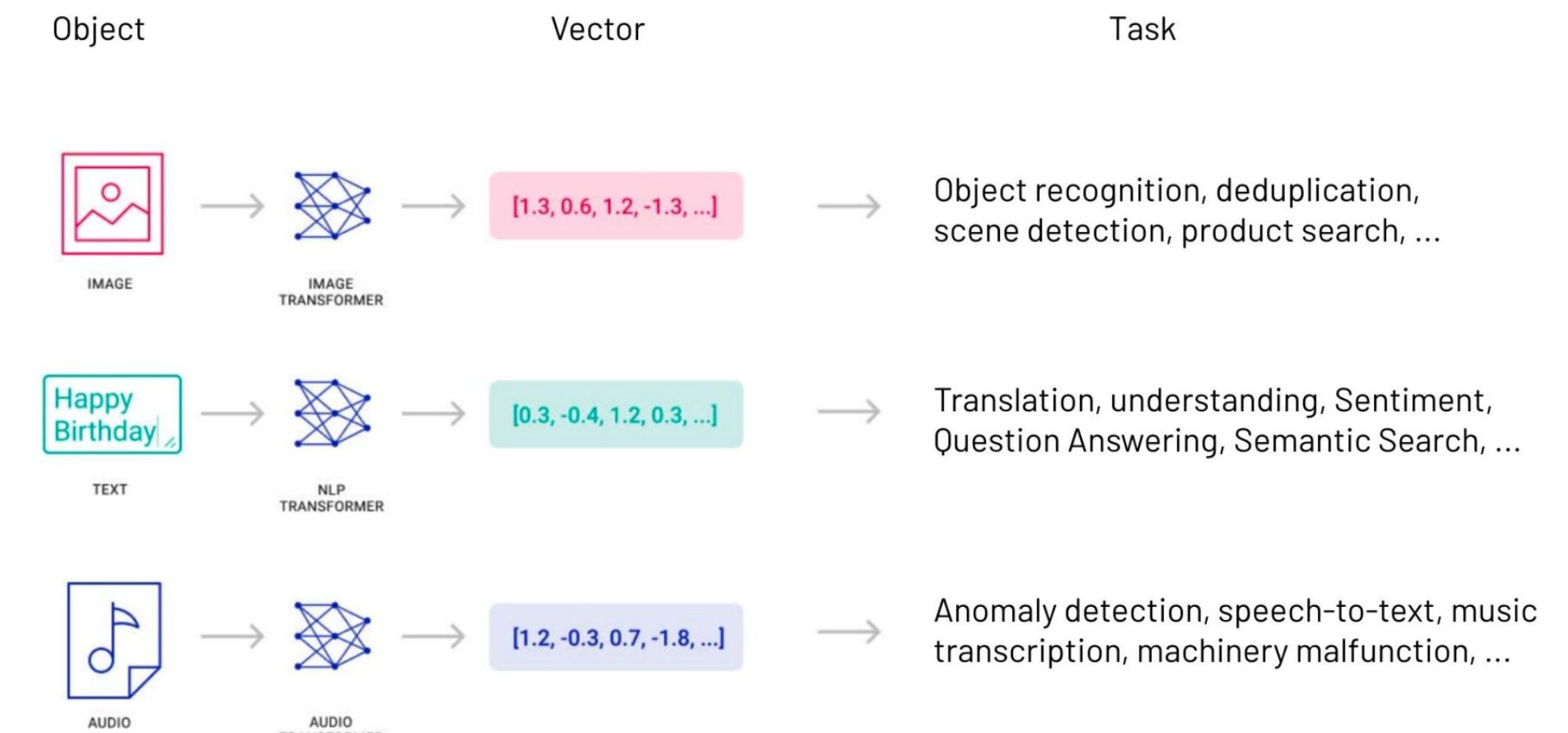
Что такое векторные поиски?

Актуальность задачи векторного поиска

- Современный ML позволяет представлять разнообразные объекты (изображения, аудио, тексты, etc) в виде векторов с **осмысленной метрикой расстояния**;
- Задача о **поиске «похожих» объектов** сводится к векторному поиску.



item2vec, Avito



Как оцениваем/сравниваем разные решения?

01

Latency

Сколько занимает операция поиска?

02

Recall

Насколько полны результаты поиска, если он приближенный?

03

Память

В основном RAM, частично disk

04

Время построения индекса

Многие решения подразумевают долгие предварительные вычисления для построения индекса

Алгоритмы векторного поиска



02
IVF

03
HNSW

Fullscan

Полный перебор!

Плюсы:

- kNN (точный поиск)!
- Простая идея и реализация
- Хорошо подходит для небольших наборов данных
- Не требует предварительных построений
- Можно ускорить вычисления, используя SIMD

Минусы:

- Перебираем все векторы на каждый поиск
- На больших наборах данных получаем **адские значения latency**

```
5 vector<int> FullScan(const vector<vector<float>> & X, const vector<float> & q, int k)
6 {
7     vector<int> n;
8     for (int i = 0; i < X.size(); ++i)
9     {
10         // ...
11     }
12     // ...
13     return n;
14 }
```

Алгоритмы векторного поиска

01

Fullscan
(naive)



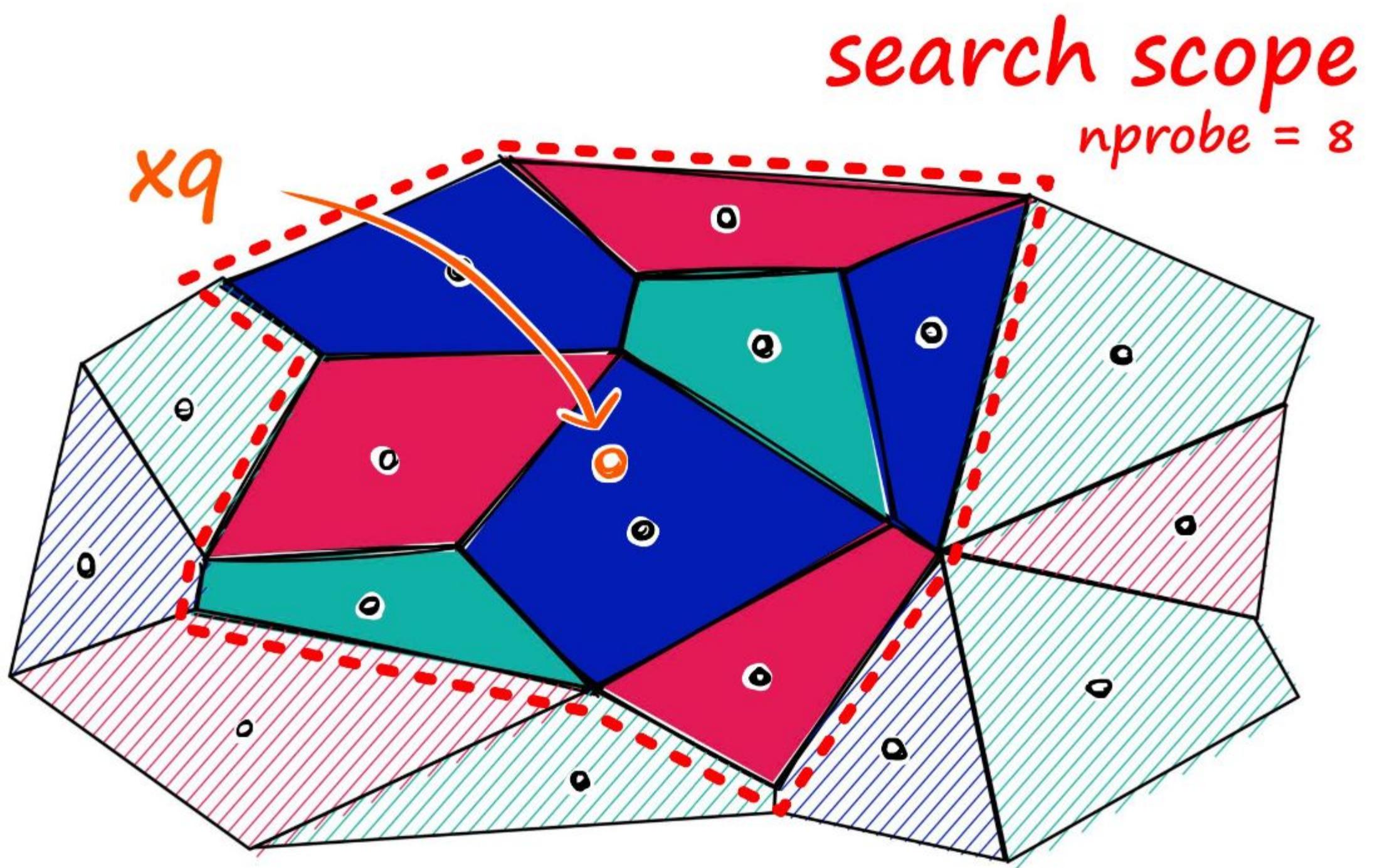
03

HNSW

IVF

Побьем вектора на кластеры!

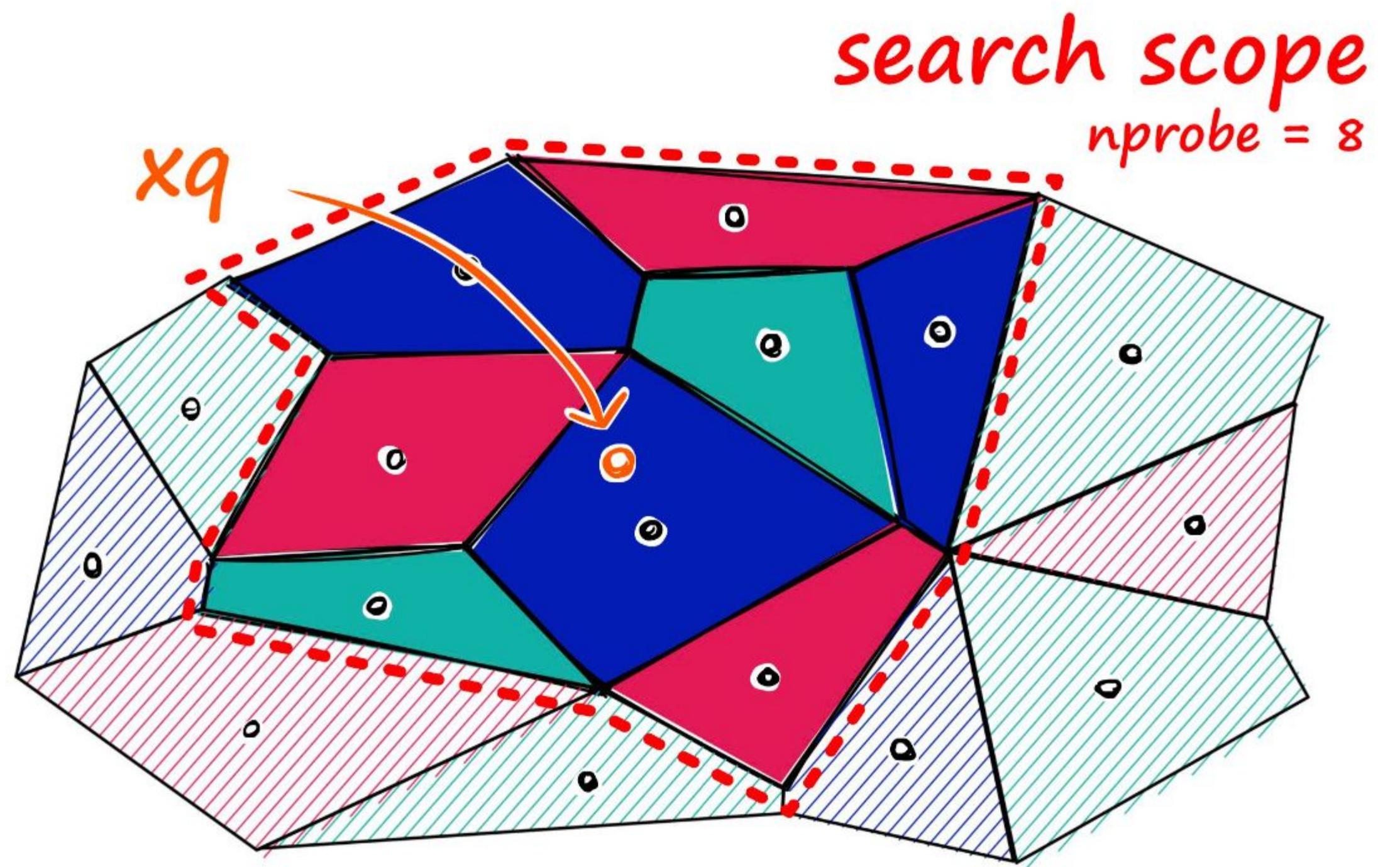
- разделим векторное пространство на **C** кластеров, используя некоторый набор векторов для обучения (Train);
- каждый кластер характеризуется «центроидом» (вектор из набора);
- векторы **X** сложены в инвертированный файл по кластерам;
- вместо полного перебора будем вести поиск лишь в **nprobe** «лучших» кластеров (чтобы их определить, смотрим на центроид).



IVF

Побьем вектора на кластеры!

- Плюсы:
 - при хорошем разделении на кластеры мы кратно ускоряем поиски, не сильно теряя в полноте;
 - оверхед памяти минимален.
- Минусы:
 - построение кластеров занимает много времени и практически невозможно в realtime;
 - aNN.



Алгоритмы векторного поиска

01

Fullscan
(naive)

02

IVF

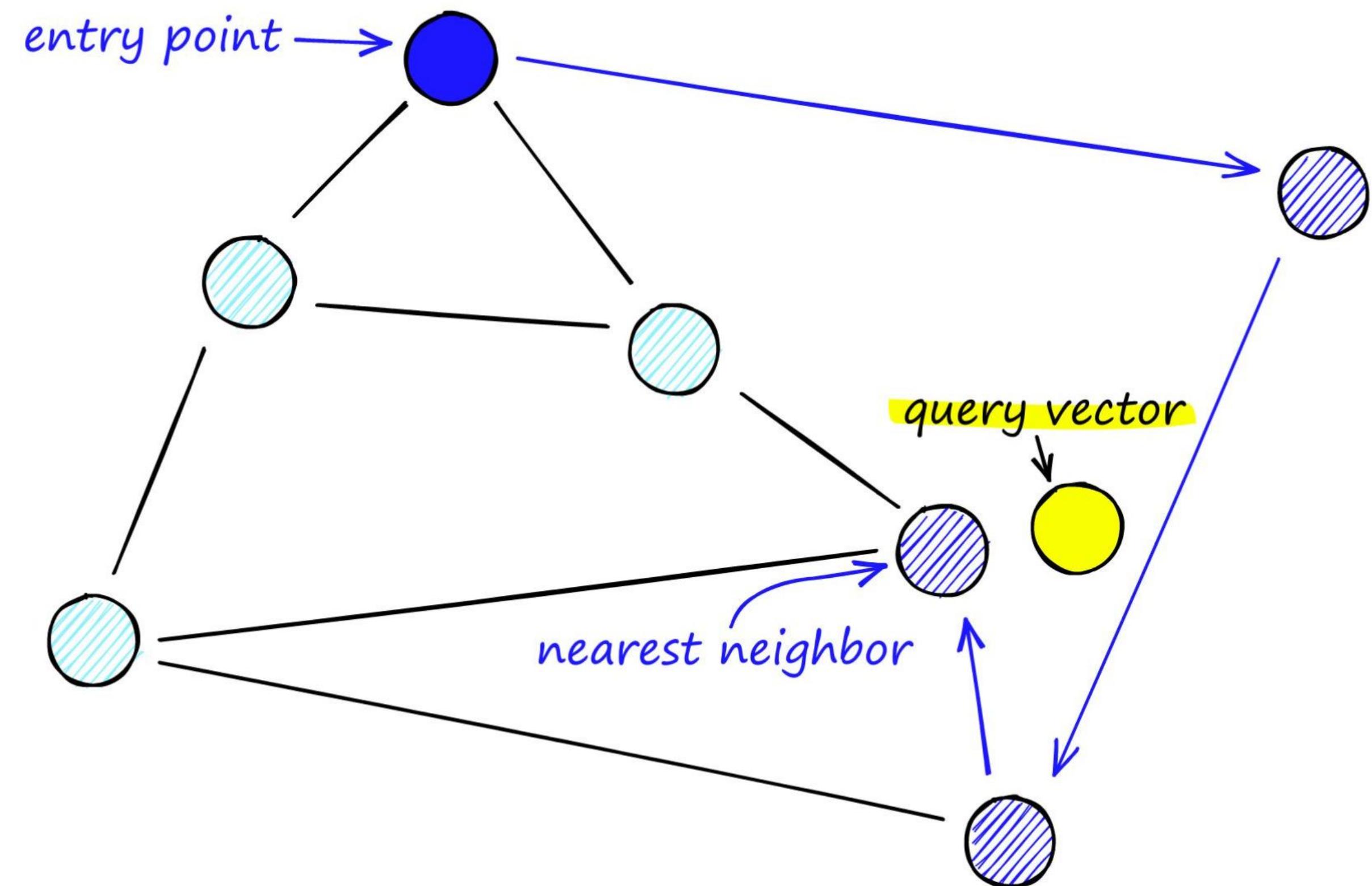
03

HNSW

HNSW

Графовый подход

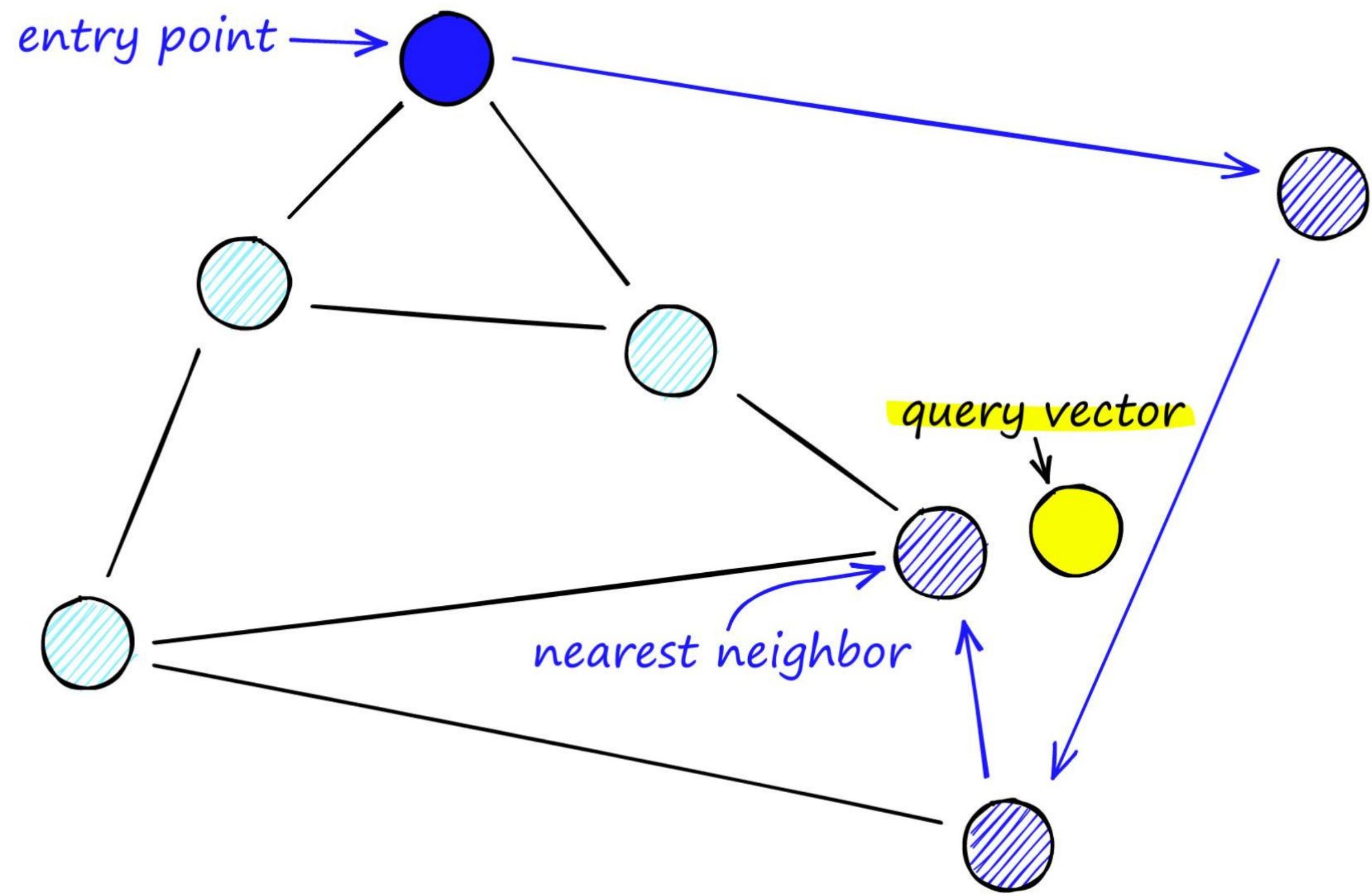
- сначала рассмотрим «одномерный» случай – **NSW** (a.k.a Navigable Small World);
- идея: будем поддерживать граф, в котором вершины – вектора из X , а ребра – отношение «соседства»;
- каждая вершина соединена ребрами с не более чем N , ближайших к ней (в терминах соответствующих векторов и $dist$);
- поиск: начнем со **случайной** вершины, на каждом шаге будем переходить по ребру в соседнюю вершину, которая ближе всего к вектору-запросу q .



HNSW

Графовый подход

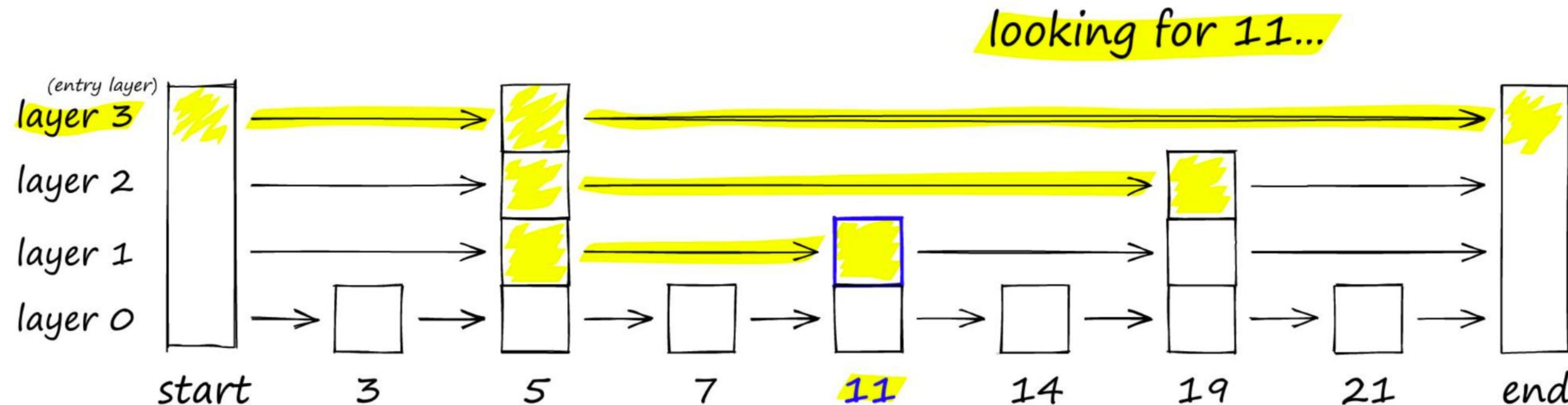
- в итоге окажемся в вершине v , из которой невозможно сделать шаг (все ее соседи более удалены от q);
- добавим в ответ вершину v и запустим «волну» (а-ля обход в ширину) по соседям, дополнив ответ вершинами из окрестности v ;
- процедуру с выбором случайной стартовой точки можно провести несколько раз, после чего объединить результаты.



HNSW

Слоеный пирог

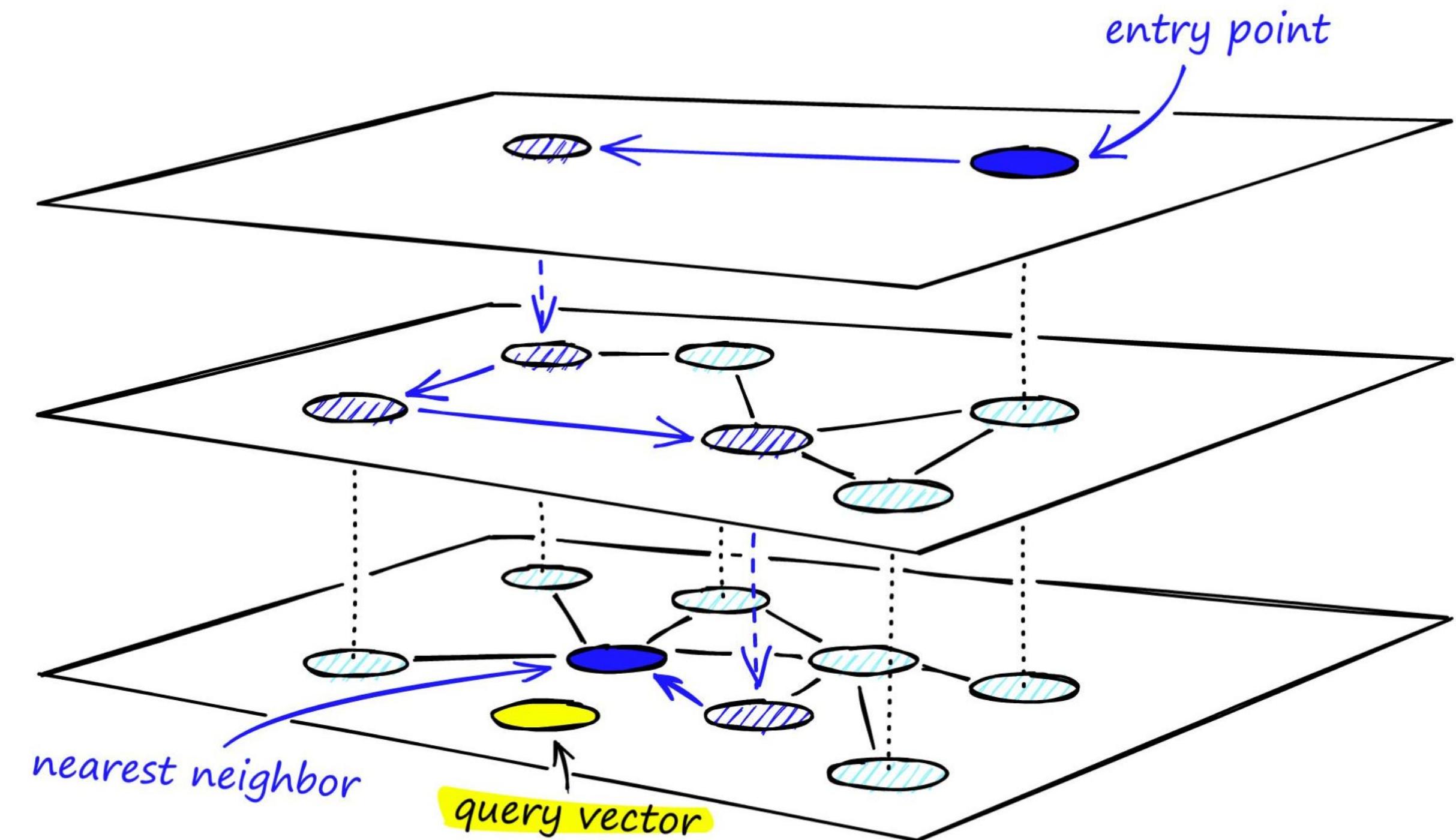
- проблема: слишком много шагов в процессе поиска, связи в графе очень локальны;
- идея: вдохновимся **skip-list**, надстроим над основной структурой **вспомогательные разреженные слои**;
- получаем **HNSW** (a.k.a. Hierarchical NSW).



HNSW

Слоеный пирог

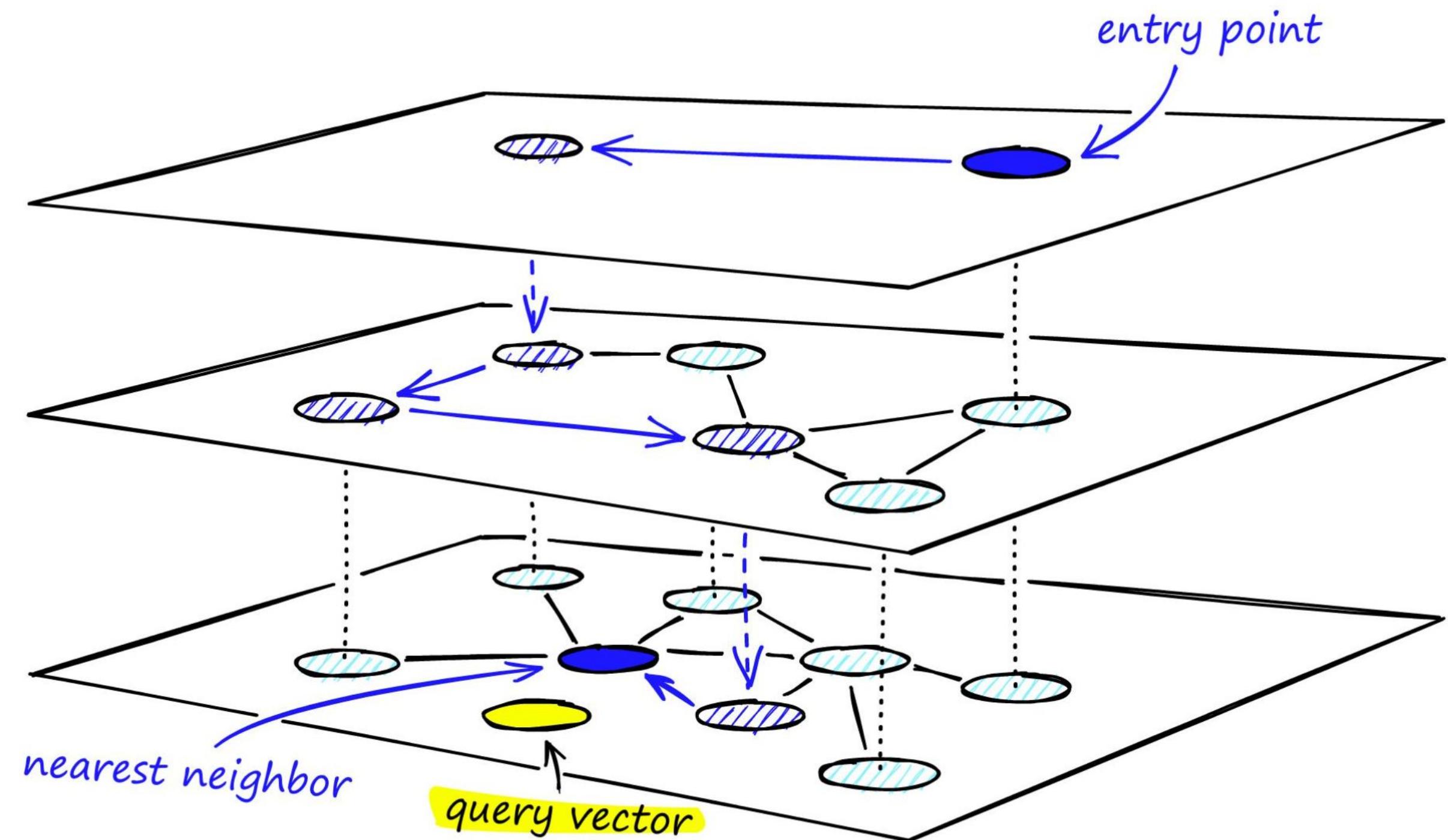
- правила те же, но одновременно будем поддерживать L слоев;
- слой 0 содержит все вектора (аналогичен NSW);
- в слой i добавляем вектор из слоя $i-1$ с вероятностью $p \in (0, 1)$;
- начинаем поиск из случайной вершины верхнего слоя;
- спускаемся на уровень ниже, когда поиск на текущем уровне закончен;
- запускаем “волну” на слое 0



HNSW

Слоеный пирог

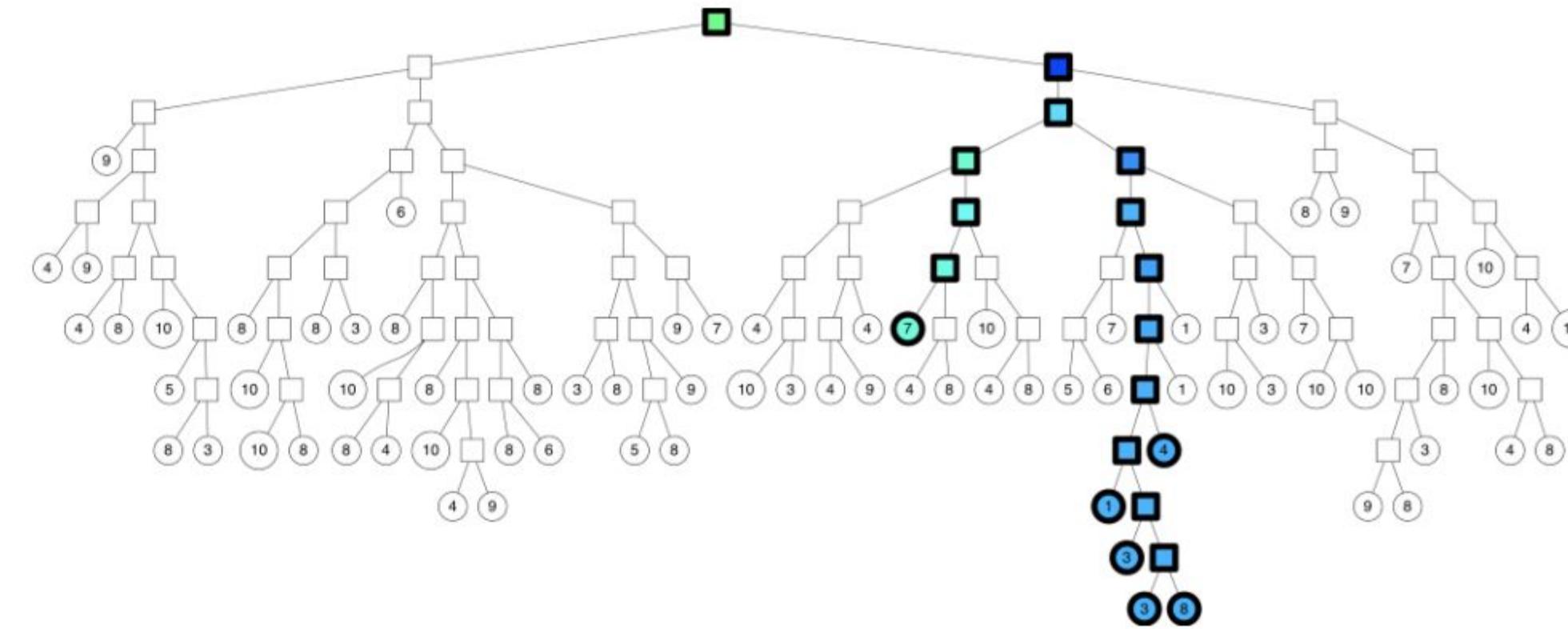
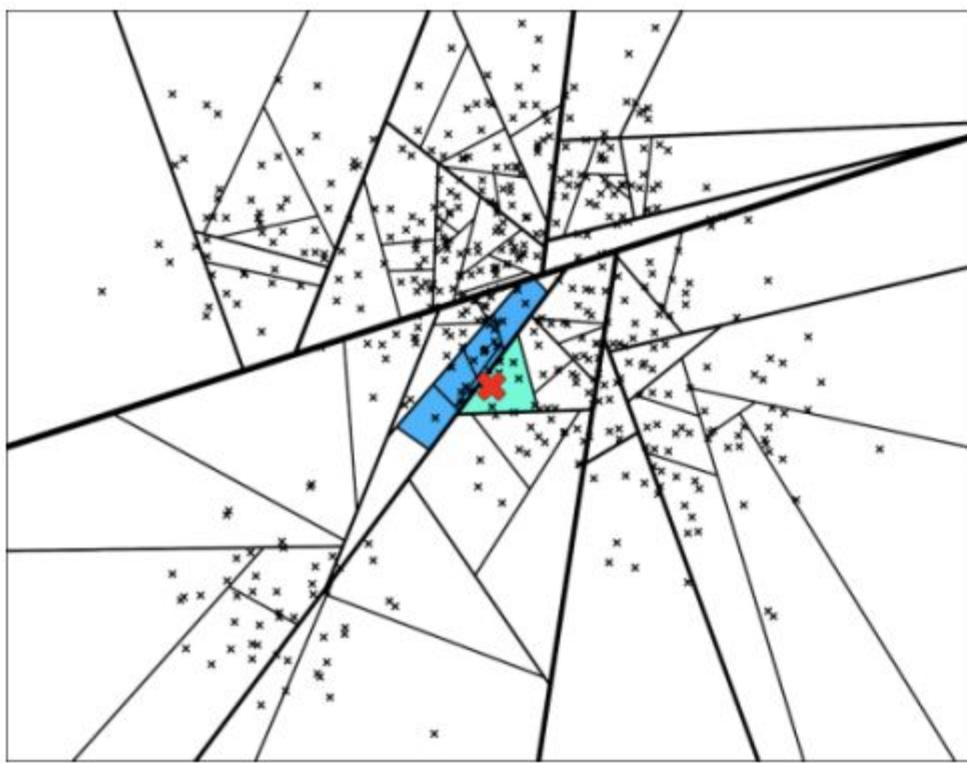
- Плюсы:
 - возможность добавлять/удалять векторы в realtime;
 - гибкая структура;
 - хорошее latency;
- Минусы:
 - концепция сложнее для понимания и реализации;
 - оверхед на хранение графа (можно оптимизировать);
 - aNN.



Алгоритмы векторного поиска

А еще... много всего

- tree-based approaches: VP-tree, Annoy approach;
- NGT: ANNG (Approximate k-Nearest Neighbor Graph);
- NGT: ONNG (Optimized Nearest Neighbors Graph);
- Vamana / DiskANN: disk-based (!) indexes.



tree based approach (annoy, scaNN, etc)

Квантизация

01

**Scalar
Quantization**

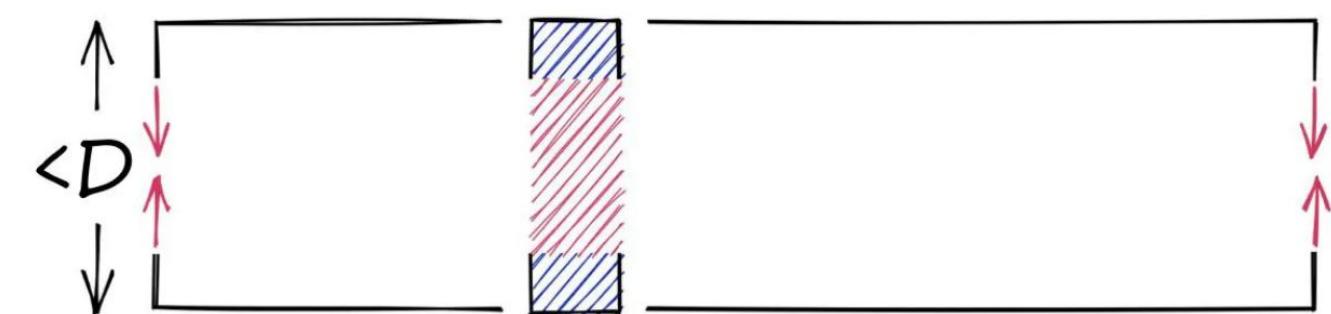
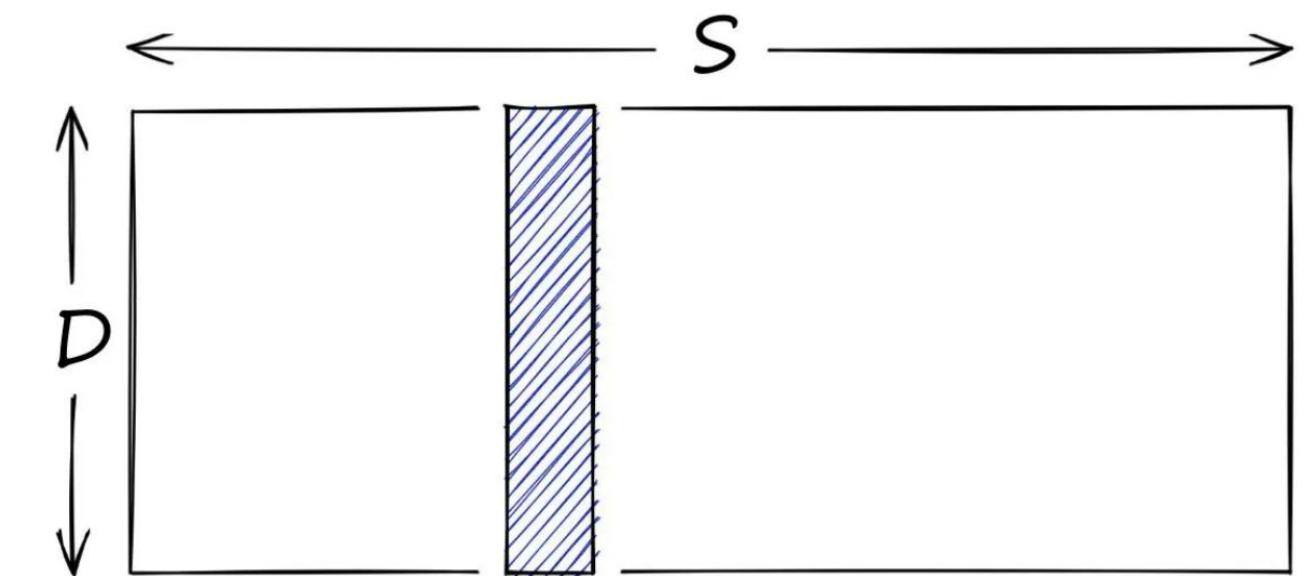
02

**Product
Quantization**

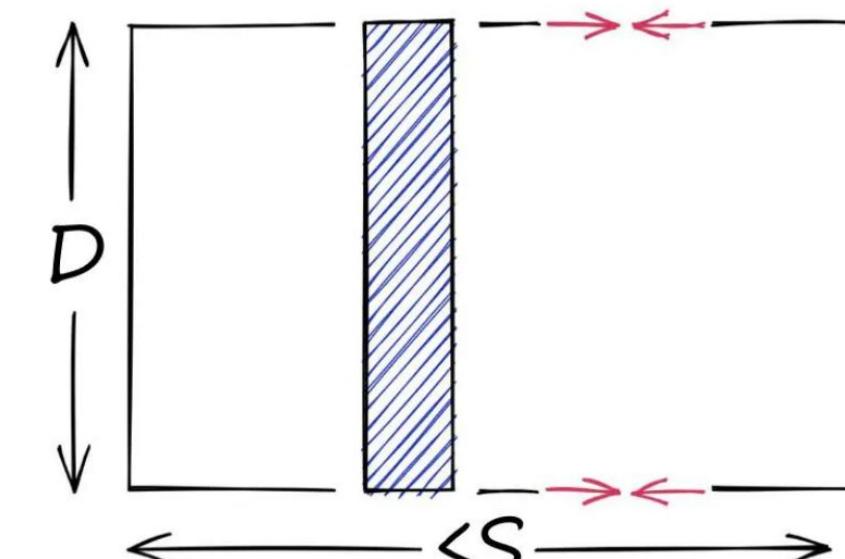
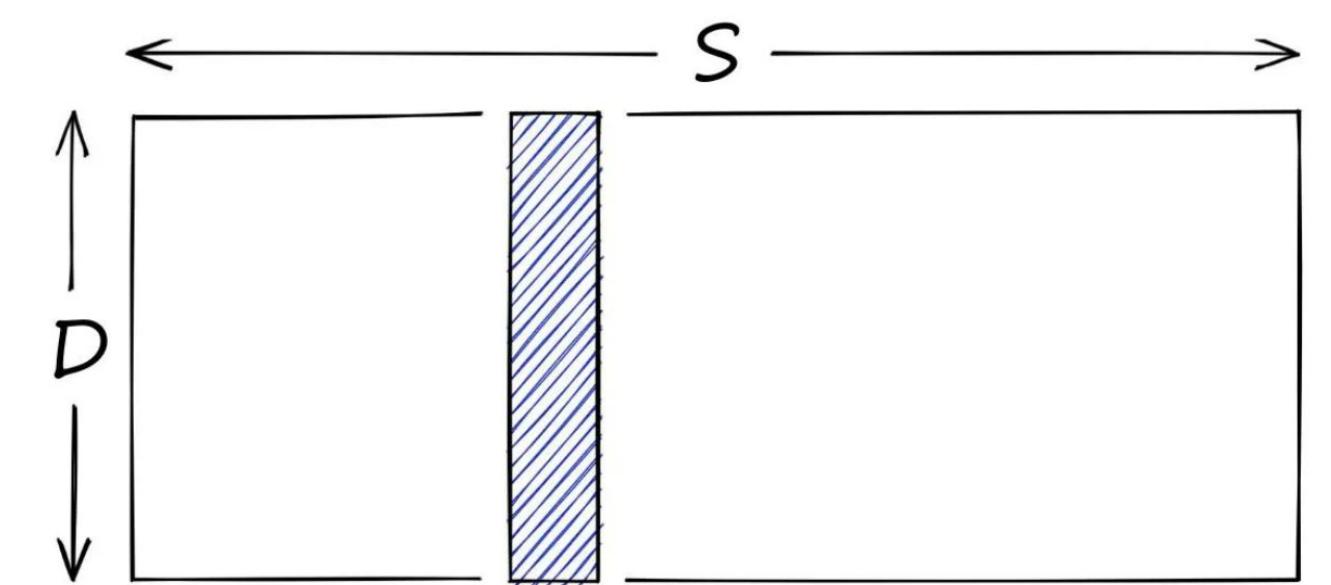
Квантизация

Хотим сделать векторы более «легкими»

- хотим: работать не с исходными векторами, а с их образами в неком менее широком пространстве;
- уменьшаем не размерность векторов (**D**), а размерность пространства (**S**);
- фактически: хотим **поменьше бит на координату**;
- что получаем:
 - **кратно** меньше памяти RAM/disk;
 - **кратно** быстрее происходят вычисления (если есть возможность работать с векторами без разжатия (а она есть)).



VS



Квантизация

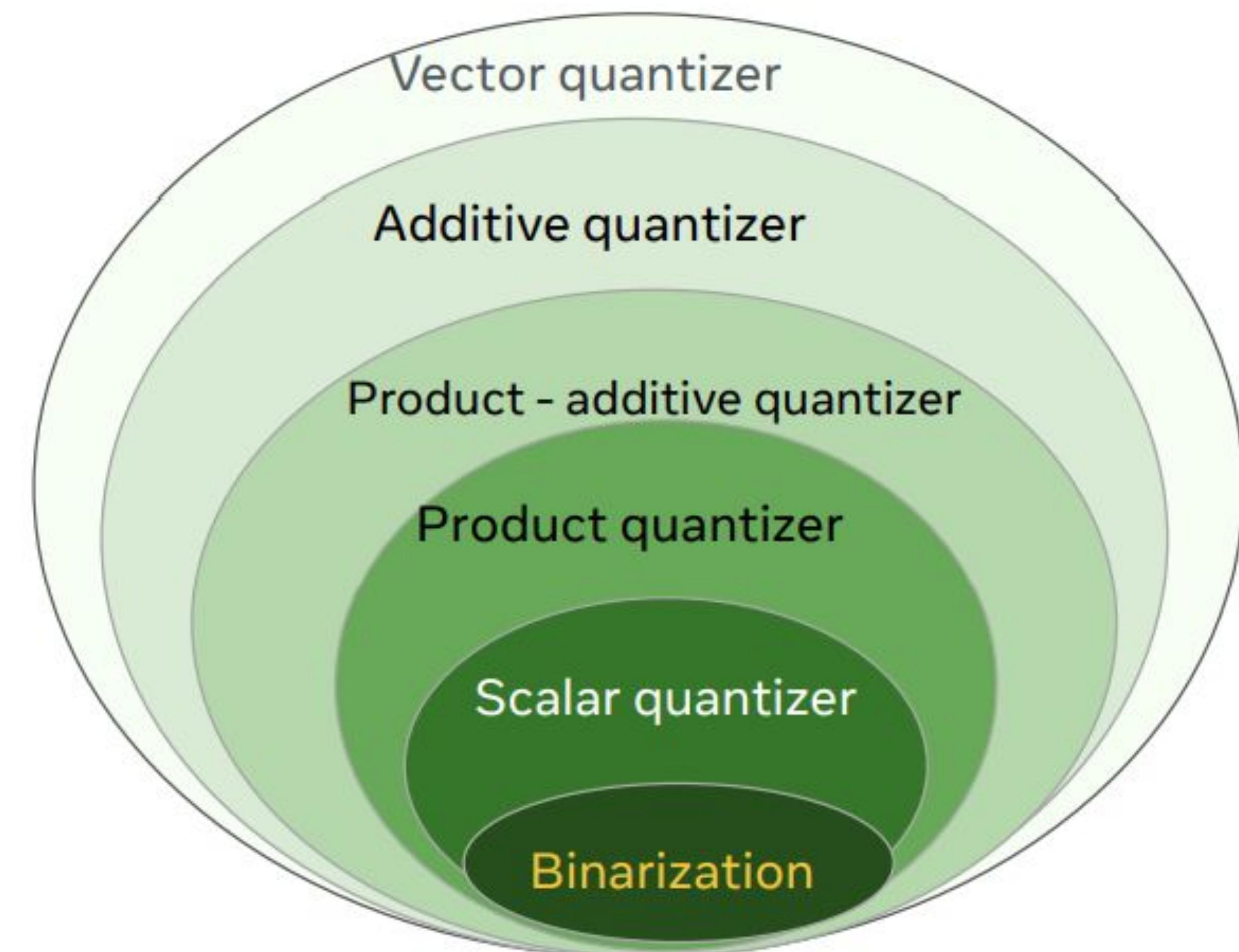


Figure 2: The hierarchy of quantizers. Each quantizer can represent the set of reproduction values of the enclosed quantizers.

Квантизация



Scalar Quantization (SQ)

e.g.: float64[] -> int8[]

- Сжимаем вектор по координатам, сохраняя размерность;
- часть информации теряется;
- можно восстановить исходные значения используя дополнительную информацию;
- сохраняем отношение порядка в смысле расстояний.

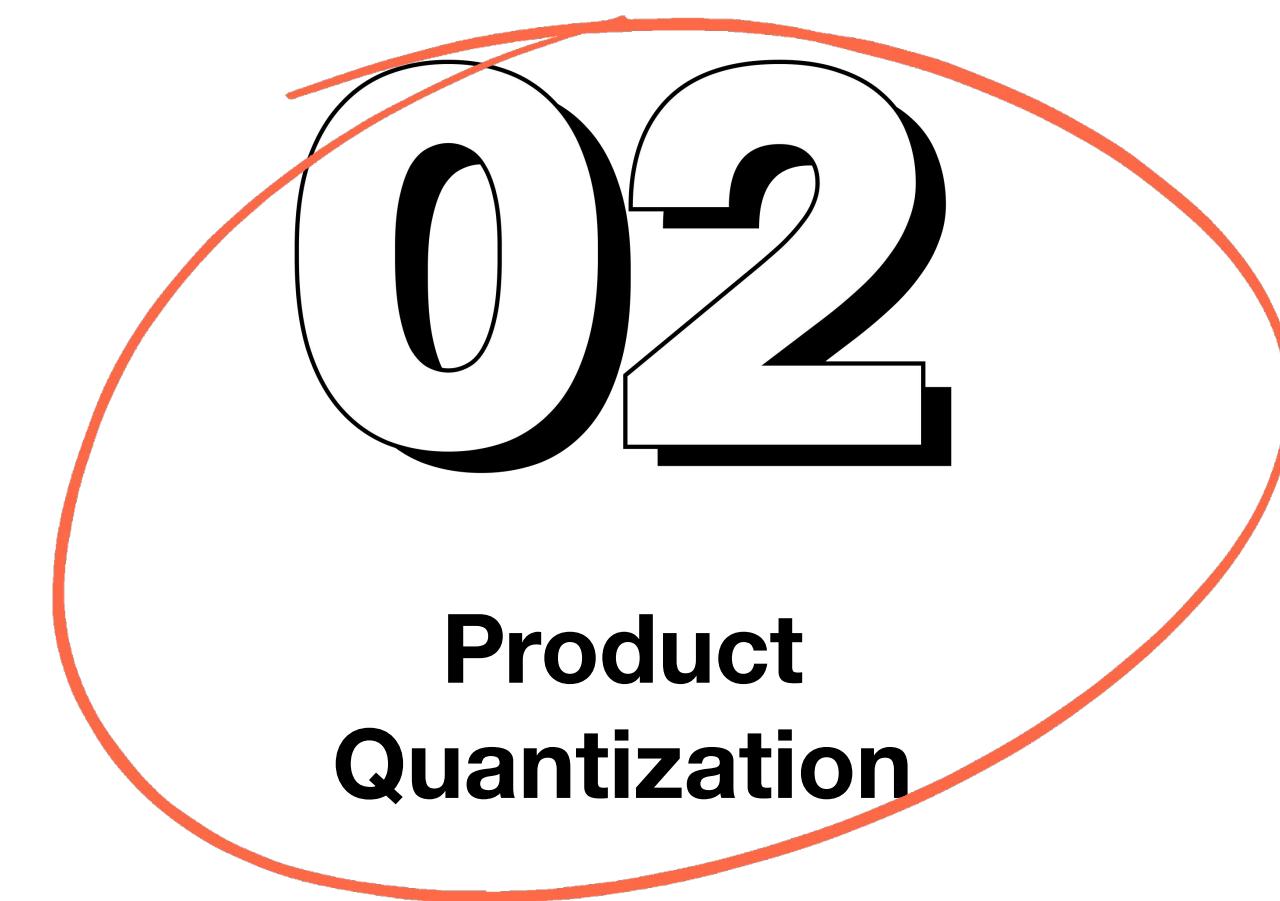
```
vector<uint8_t> ScalarQuant(const vector<double> & vec)
{
    double max = *std::max_element(vec.begin(), vec.end());
    double min = *std::min_element(vec.begin(), vec.end());
    vector<uint8_t> res(vec.size());
    for (size_t i = 0; i < vec.size(); ++i)
        res[i] = 255.0 * (vec[i] - min) / (max - min);
    return res;
}
```

(0.1, 0.2, 0.3) -> (0, 127, 254)

Квантизация

01

**Scalar
Quantization**



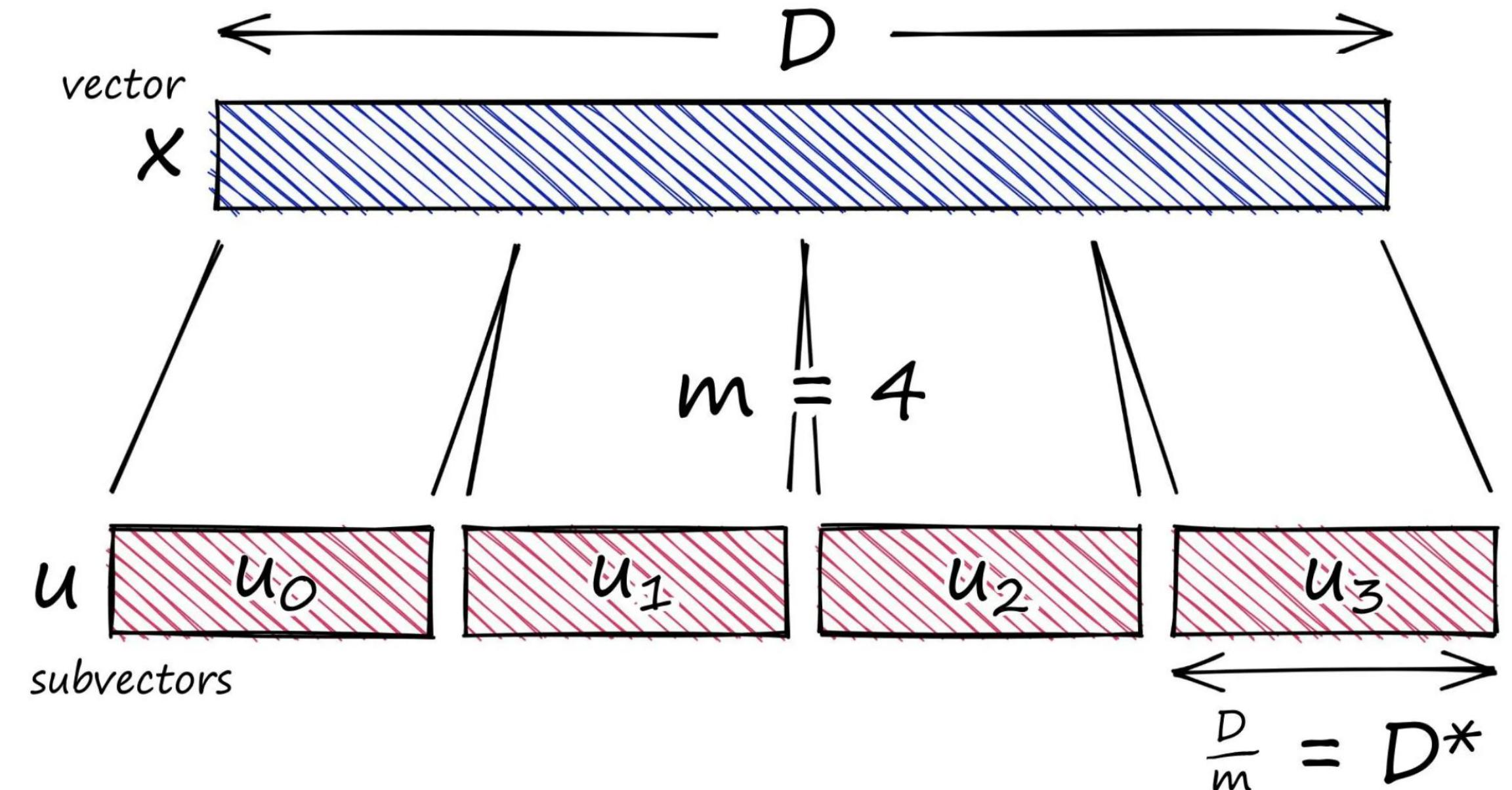
Product Quantization (PQ)

Разрежем, закодируем

- Будем рассматривать вектор \mathbf{x} как конкатенацию m независимых векторов размерности d/m
- Разобьем каждое из m новых векторных пространств на кластера (никак не соотносится с IVF!). Например, методом kmeans
- Сопоставим оригинальному вектору \mathbf{x} вектор индексов (в терминах этих кластеров)

$$\mathbf{x} \mapsto \mathbf{q}(\mathbf{x}) = [\mathbf{q}^1(\mathbf{x}^1), \dots, \mathbf{q}^M(\mathbf{x}^M)]$$

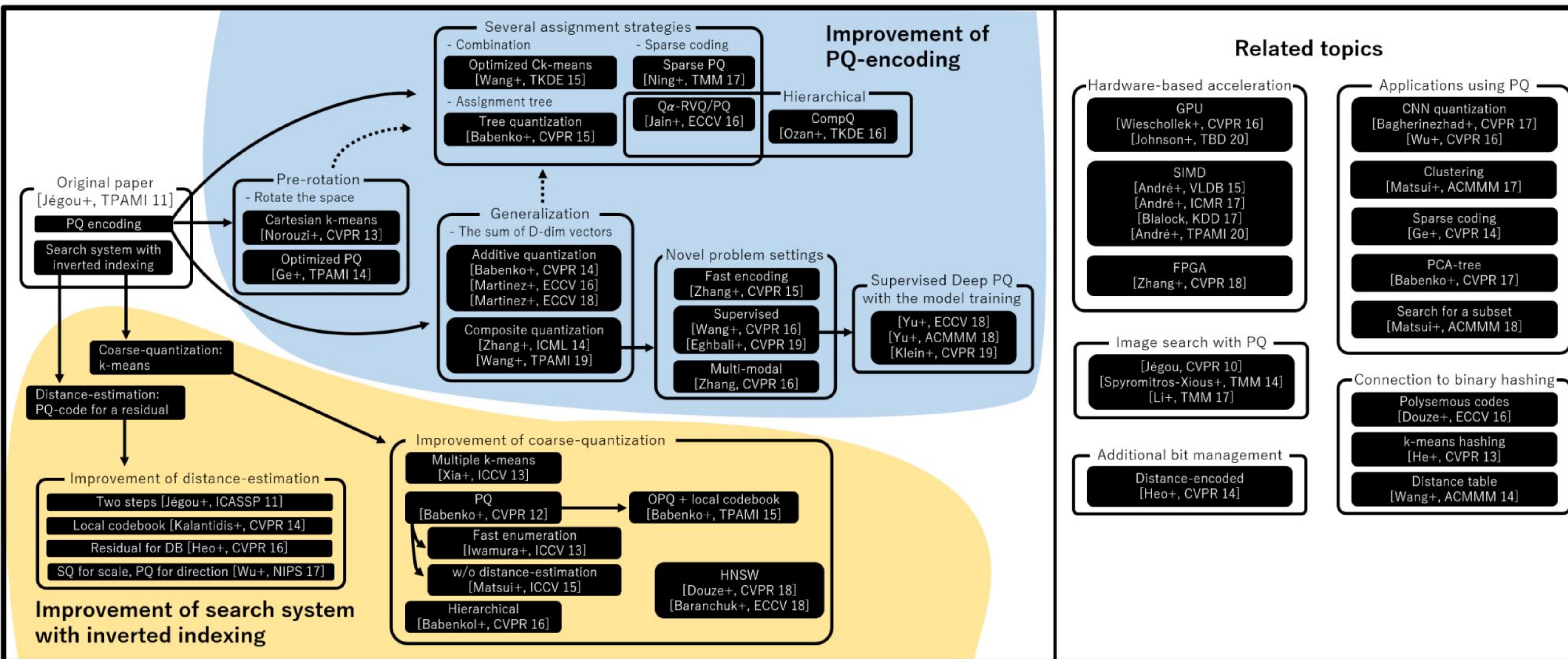
$$\mathbf{x}^m \mapsto \mathbf{q}^m(\mathbf{x}^m) = \arg \min_{\mathbf{c}^m \in \mathcal{C}^m} \|\mathbf{x}^m - \mathbf{c}^m\|.$$



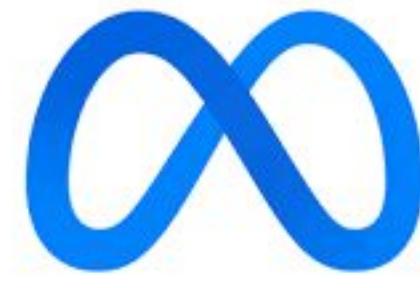
А еще...

Разнообразные полезные техники

- **PCA** перед квантизацией, если вектора имеют большую размерность и есть подозрение, что координаты скоррелированы
- Построение (части) индексов на **GPU**
- **Binary Quantization**, $f32 \rightarrow \{0, 1\}$
- **Ternary Quantization**, $f32 \rightarrow \{-1, 0, 1\}$



Библиотеки



Faiss

PQ, SQ
Flat, IVF, HNSW
GPU build
PCA



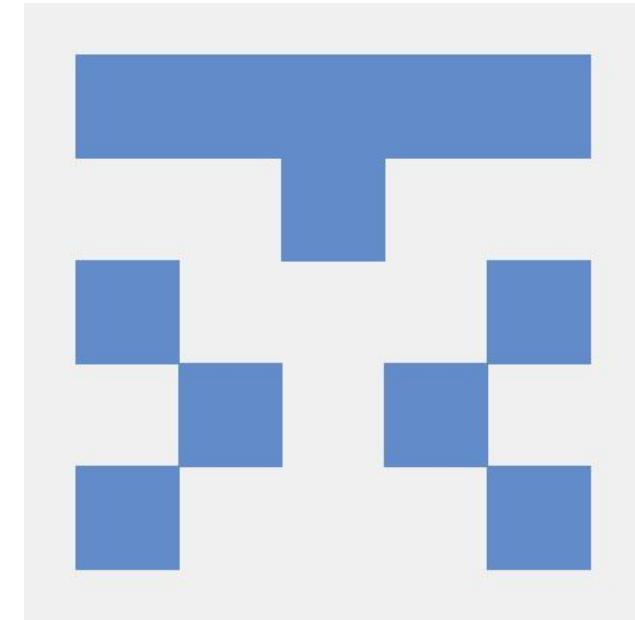
ScaNN

tree-based partitioning
PQ, AQ



Usearch

SQ
HNSW



nmslib

IVF, HNSW, VP-tree,
SW-graph, NAPP
no quant

Векторные СУБД



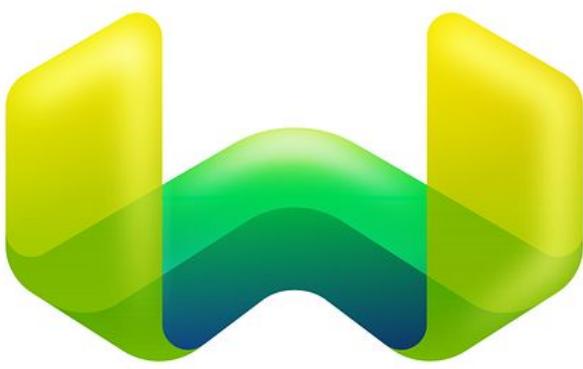
Qdrant

Своя реализация
HNSW (Rust)



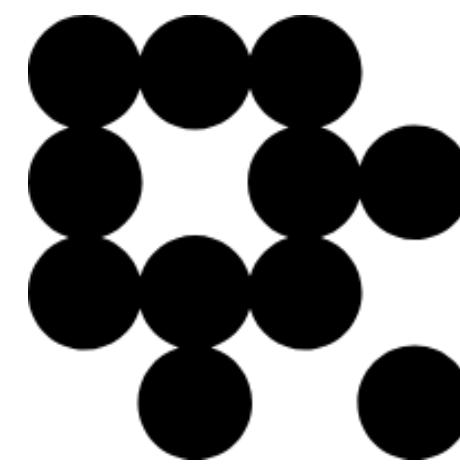
Milvus

Faiss, ScaNN
IVF, HNSW
PQ, SQ



Weaviate

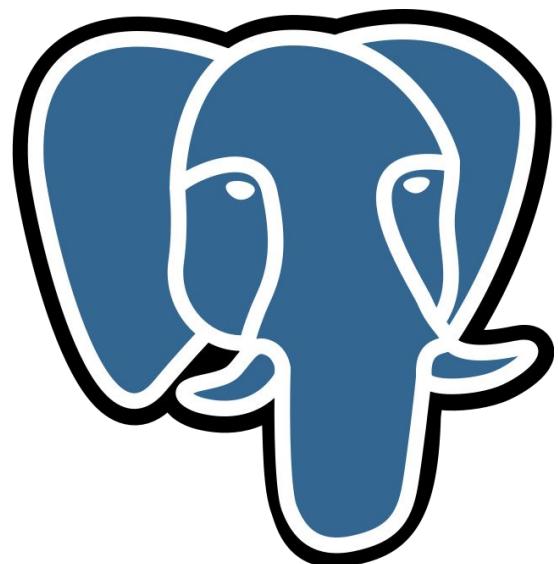
Своя реализация
HNSW (golang)



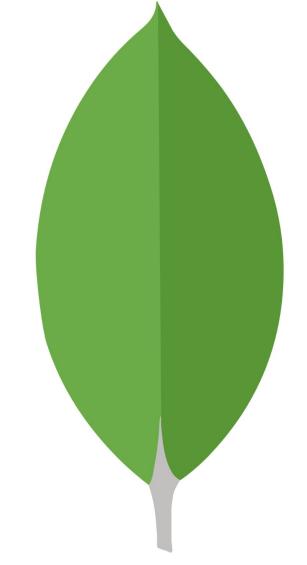
LanceDB

Своя реализация
IVF+PQ (Rust)

Поддержка в «обычных» СУБД



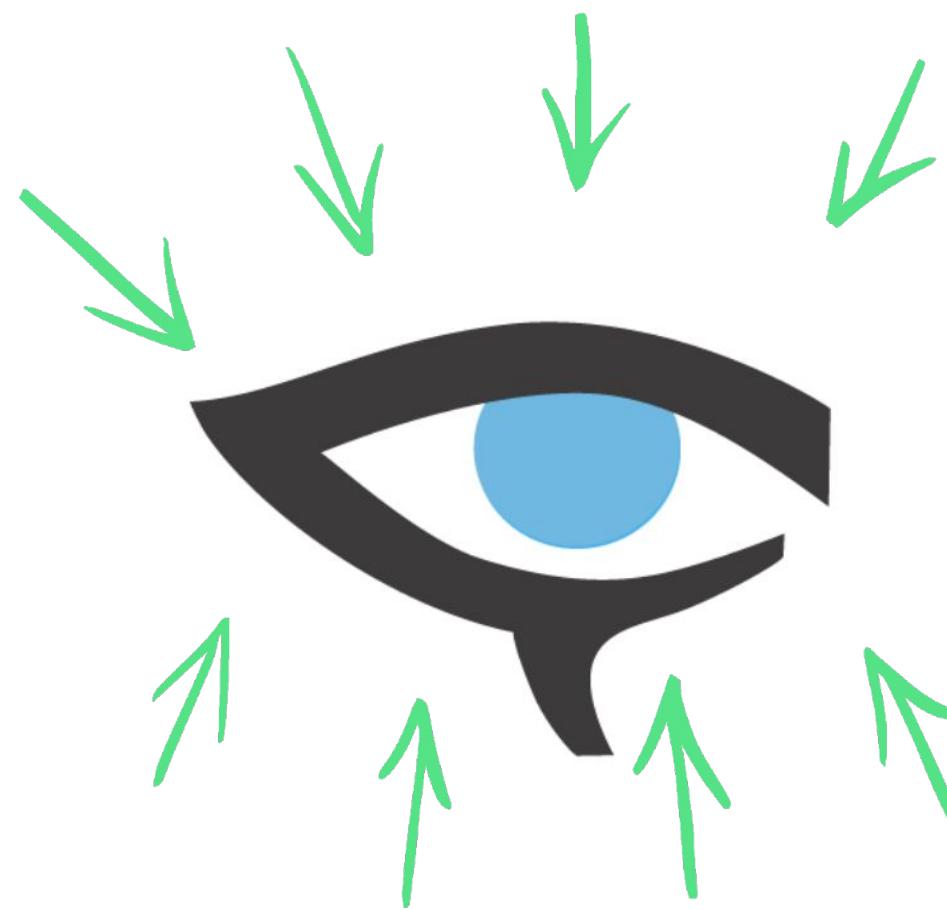
Postgres



MongoDB



ClickHouse



Sphinx

pgvector extension
HNSW, IVF-flat

RDBMS = check!

HNSW
built on top of Apache Lucene

Document DB = check!

HNSW
Usearch

OLAP = check!

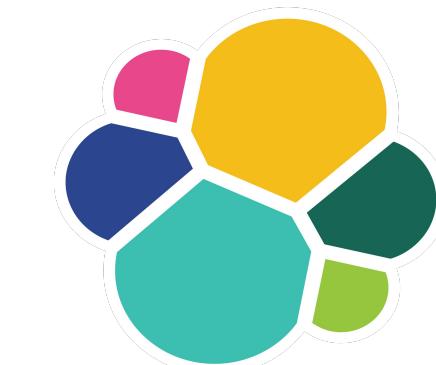
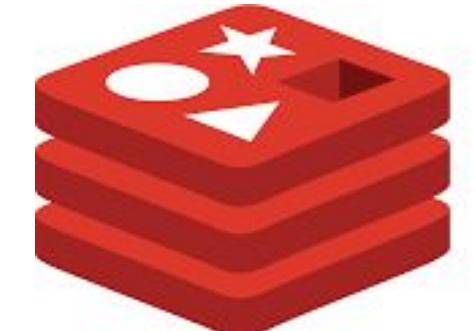
Faiss IVFPQ, HNSW
Свои SQ, HNSW

Full-text search = check!

А еще...

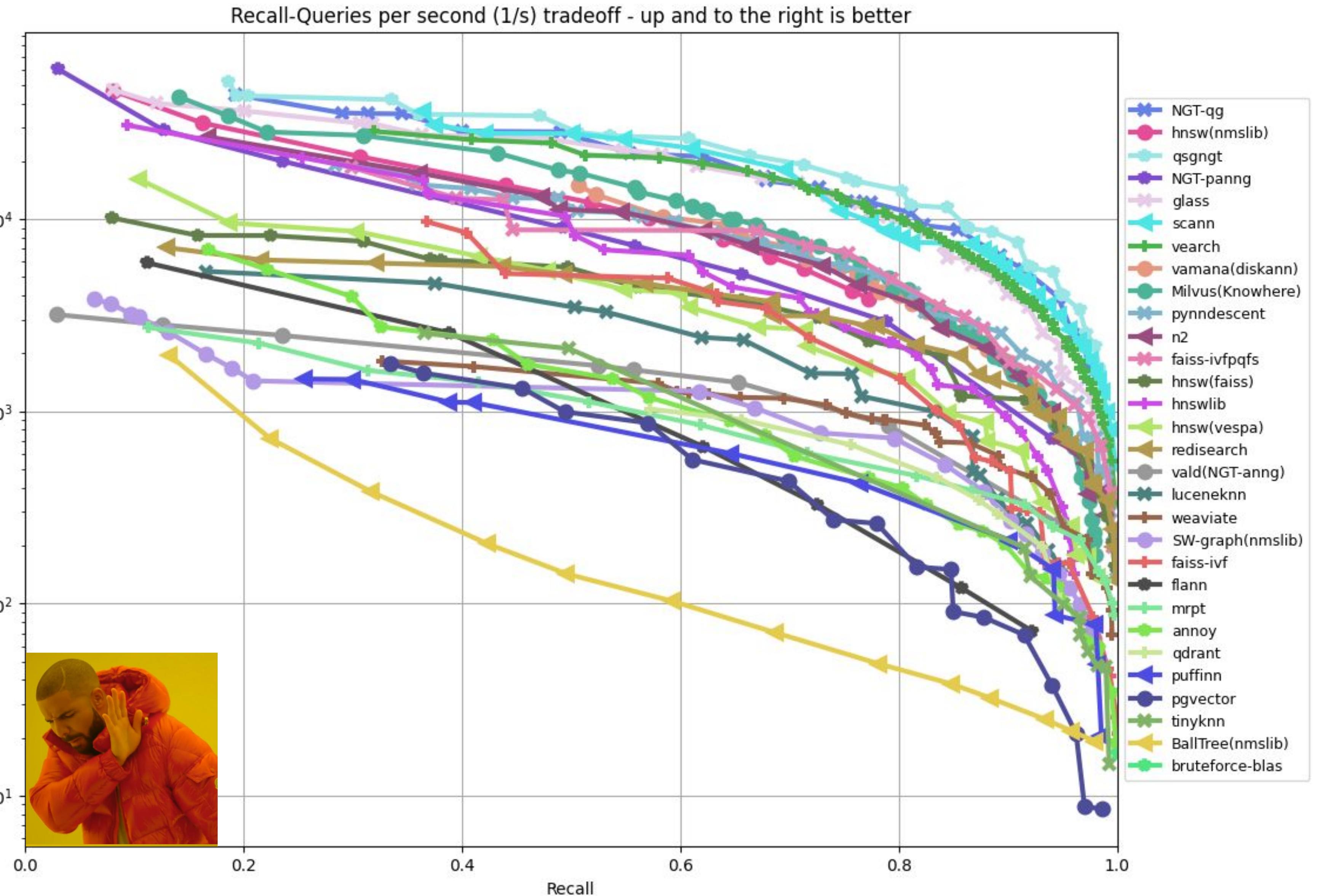
Библиотек и СУБД – очень много!

- Библиотеки: hnswlib, Annoy, NGT, ...
- Векторные СУБД: Pinecone, Chroma, Vespa, Vald, Vearch ...
- Поддержка в «обычных» СУБД: Redis, ElasticSearch, Lucene, Oracle, MySQL, MariaDB ...



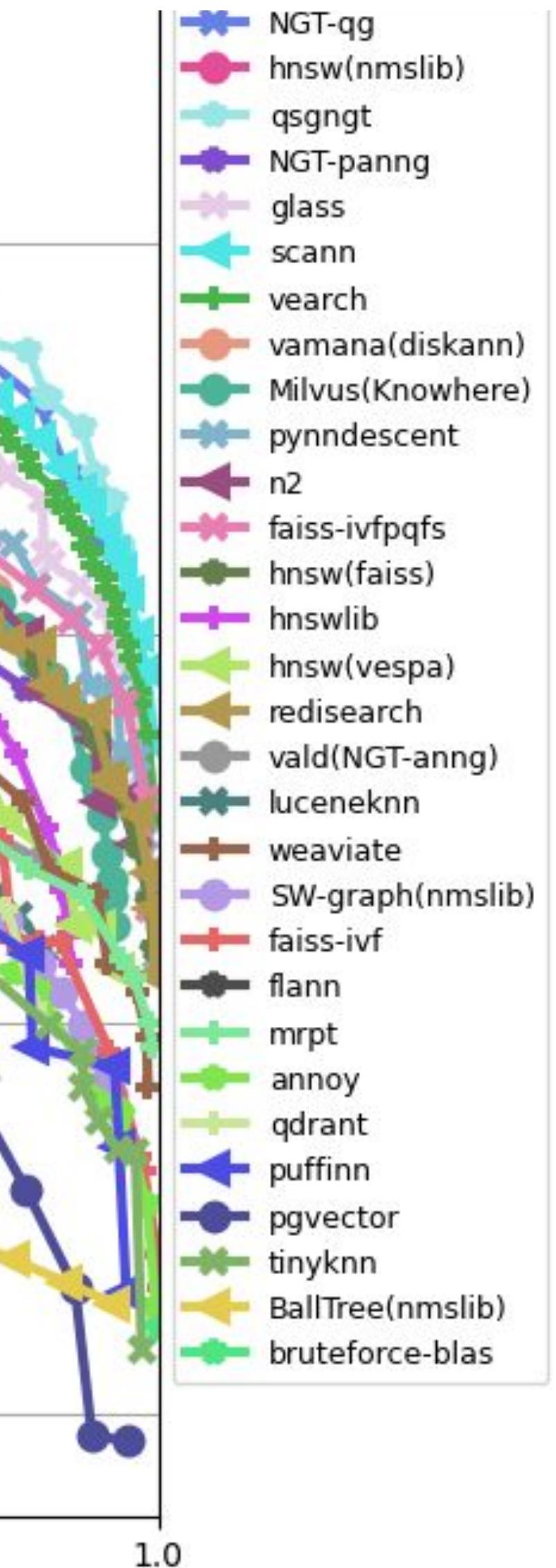
ann-benchmarks

- dataset: GloVe-100
- metric: angular (dot)
- k=10



ann-benchmarks

- pro-tip!
- НИКТО:
- **абсолютно никто:**
- ann-benchmarks: whee нам очень очень нужно recall 0.1 давайте оси от нуля!!!



Опыт внедрения и эксплуатации в Sphinx

4

2

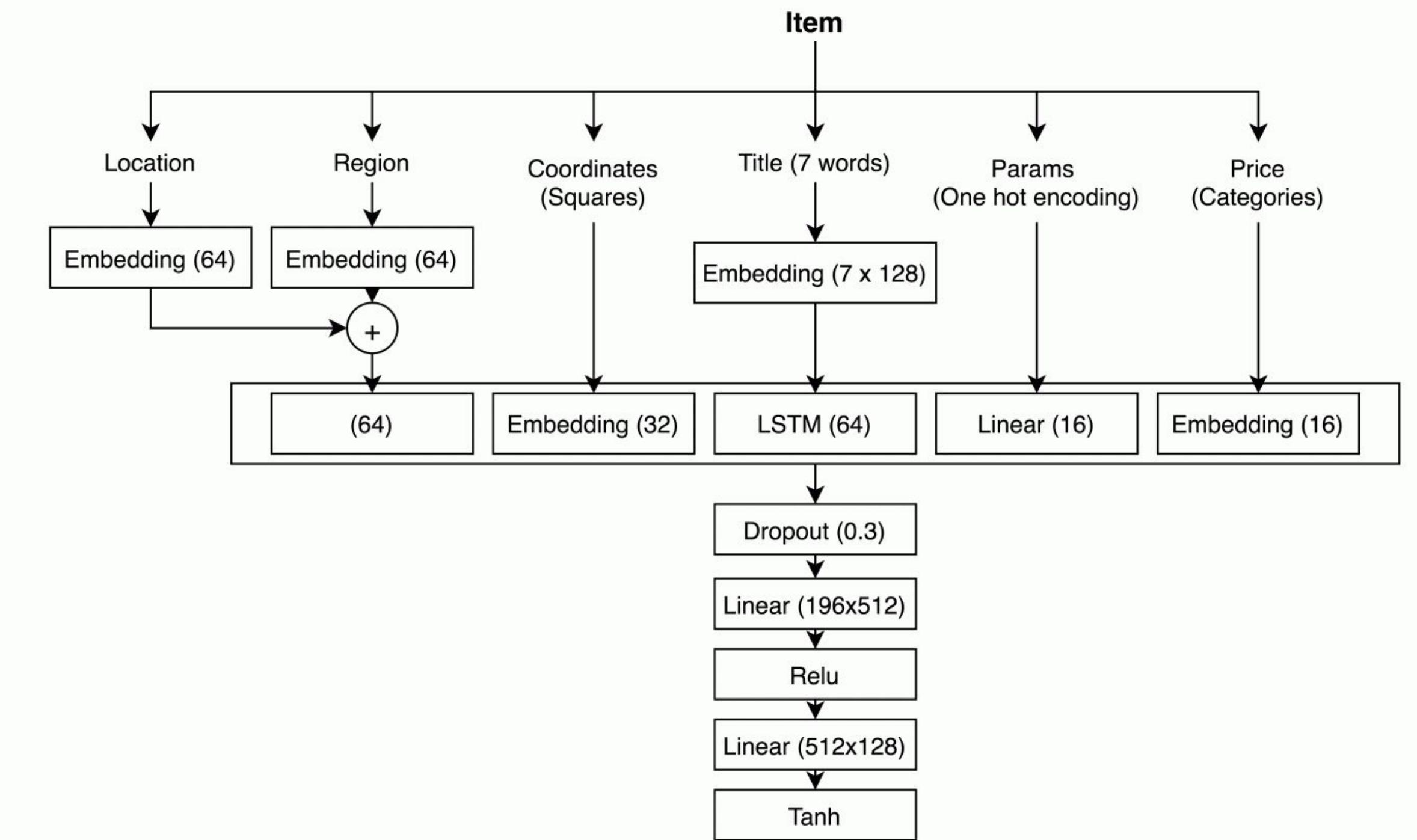
3

1

Основной поиск Авито

Векторные поиски в production

- модель **item2vec**: ищем похожие объявления;
- на выходе из модели (после Tanh): [-1, 1] float вектор размерности 128;
- после этого float вектор конвертируется в int8 вектор (умножение на 127 + округление) для экономии памяти;
- cosine similarity (эквивалентна dot product на нормированных векторах).



статья на Хабре про item2vec



Основной поиск Авито

Векторные поиски в production

- **SphinxSearch;**
- OLTP + FTS ofc (+ geosearch + vector search + ...);
- **~220M+ rows** (синонимично: документов, объявлений, айтемов);
- Шардирование на 4 по id (int64, ~random+unique) → **55M+** / box;
- Всё влезает в RAM – и векторные индексы влезают в RAM.

```
index items_category_XXb
{
    type = rt
    ...
    attr_int8_array = i2v[128]
    attr_int8_array = i2v_alt[128]
    attr_int8_array = i2v_3[128]
    ...
    pretrained_index = \
        items_category_XX.pretrain
}
```

MySQL [(none)]> DESC items_category_XXb;

	int8_array[128]	i2v_idx
i2v	int8_array[128]	i2v_idx
i2v_alt	int8_array[128]	i2v_alt_idx
i2v_3	int8_array[128]	i2v_3_idx

vector cols with faiss indexes

Основной поиск Авито

Векторные поиски в production

```
SELECT * FROM (
    SELECT id,
        ml_udf(factors()) AS ml_rank,
        dot(i2v, fvec(-19, 51, /* ... */, -86, 127)) as dot_rank
    FROM items_category_XX
    WHERE MATCH('iphone')
    ORDER BY dot_rank DESC
    LIMIT 0, 300
) ORDER BY ml_rank desc LIMIT 8;
```

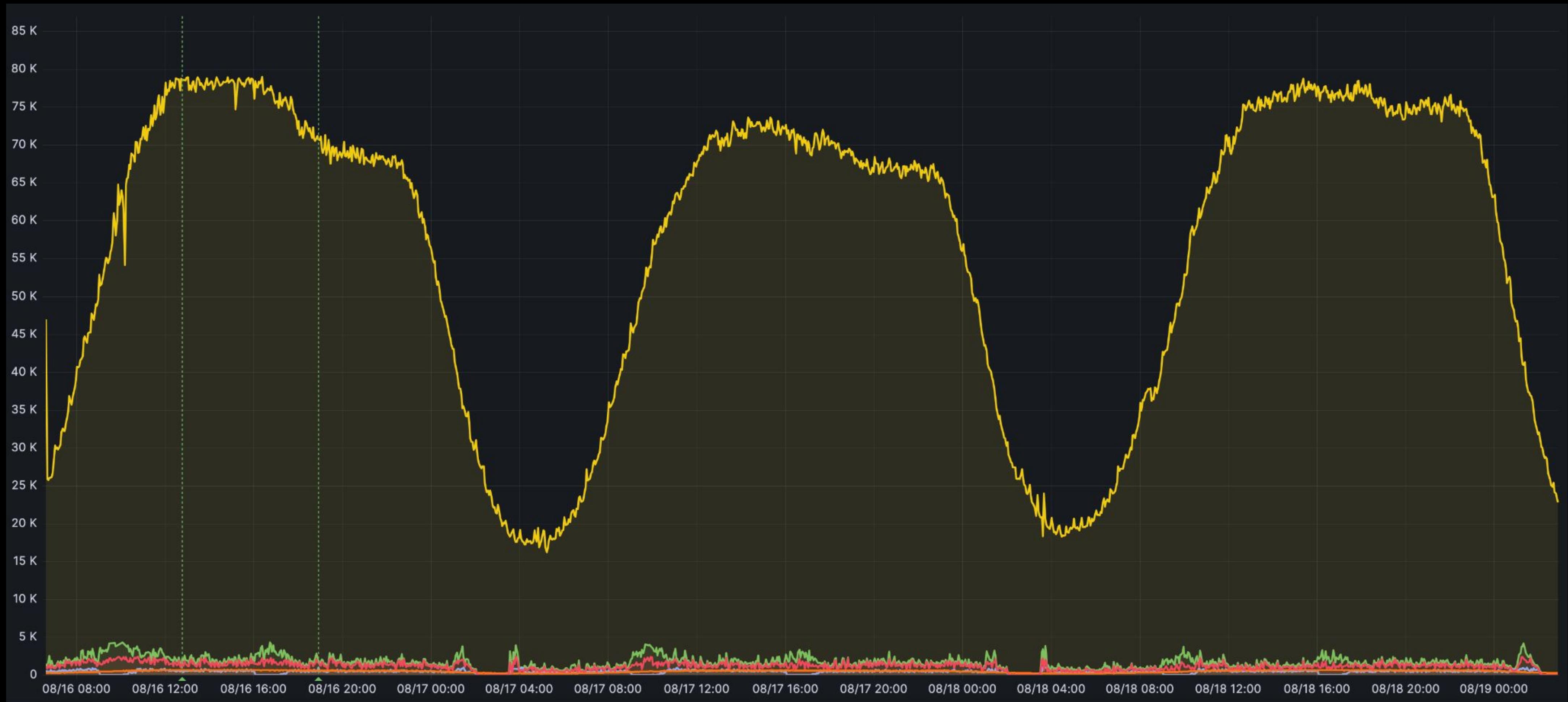
Редуцированный prod запрос в Sphinx

Основной поиск Авито

Векторные поиски в production

```
SELECT * FROM (
    SELECT id,
        ml_udf(factors()) AS ml_rank,
        dot(i2v, fvec(-19, 51, /* ... */, -86, 127)) as dot_rank
    FROM items_category_XX
    WHERE MATCH('iphone')
    ORDER BY dot_rank DESC
    LIMIT 0, 300
) ORDER BY ml_rank desc LIMIT 8;
```

Редуцированный prod запрос в Sphinx



В пике, суммарно:
select: ~70-80k RPS
replace+update+delete: ~5-7k RPS



В пике per-shard:
select: ~300-1k RPS
replace+update+delete: ~10 RPS

Fullscan+SIMD

2018 год

- **Fullscan** a.k.a пробегаем по всем векторам;
- Ускоряем вычисление dot(), l1dist() при помощи SIMD (AVX, SSE);
- До сих пор используем для RAM-сегментов (подробнее – далее).

```
// load 32 chars from each vector
// signed-unpack them into shorts, to avoid multiplication
// multiply those shorts
__m256i a1 = _mm256_loadu_si256((const __m256i *)a);
__m256i b1 = _mm256_loadu_si256((const __m256i *)b);
_mm_prefetch((const char *)a + 32, _MM_HINT_T0); // ...
_mm_prefetch((const char *)b + 32, _MM_HINT_T0);
auto a2 = _mm256_unpacklo_epi8(a1, a1); // [a0..a7, a16..a23]
auto a3 = _mm256_unpackhi_epi8(a1, a1); // [a8..a15, a24..a31]
auto b2 = _mm256_unpacklo_epi8(b1, b1); // [b0..b7, b16..b23]
auto b3 = _mm256_unpackhi_epi8(b1, b1); // [b8..b15, b24..b31]
a2 = _mm256_srai_epi16(a2, 8); // [a0..a7, a16..a23]
a3 = _mm256_srai_epi16(a3, 8); // [a8..a15, a24..a31]
b2 = _mm256_srai_epi16(b2, 8); // [b0..b7, b16..b23]
```

кусочек avx2dotcc

Faiss IVFPQ

2020-21 год

- IVF+PQ для dot(). **FastScan** (block invlists + PQ codebook влезает в регистры)
- `sFaissSpec.SetSprintf("IVF3000, PQ%dx4fs", iVecSize / 2);`
- PQ: M=128/2=64, nbits = 4
- Выбрали Faiss (в основном vs ScaNN) так как:
 - Простота интеграции
 - Много фичей из коробки, большая вариативность

	No encoding	PQ encoding	scalar quantizer
Flat	IndexFlat	IndexPQ	IndexScalarQuantizer
IVF	IndexIVFFlat	IndexIVFPQ	IndexIVFScalarQuantizer
HNSW	IndexHSNFWFlat	IndexHNSWPQ	IndexHNSWScalarQuantizer

basic index types (faiss paper)

String	Class name	Output dimension
PCA64, PCAR64, PCA64, PCAWR64	PCAMatrix	64
OPQ16, OPQ16_64	OPQMatrix	d, 64
RR64	RandomRotation	64
L2norm	NormalizationTransform	d
ITQ256, ITQ	ITQMatrix	256, d
Pad128	RemapDimensionsTransform	128

vector transforms (faiss wiki)

Особенности внедрения

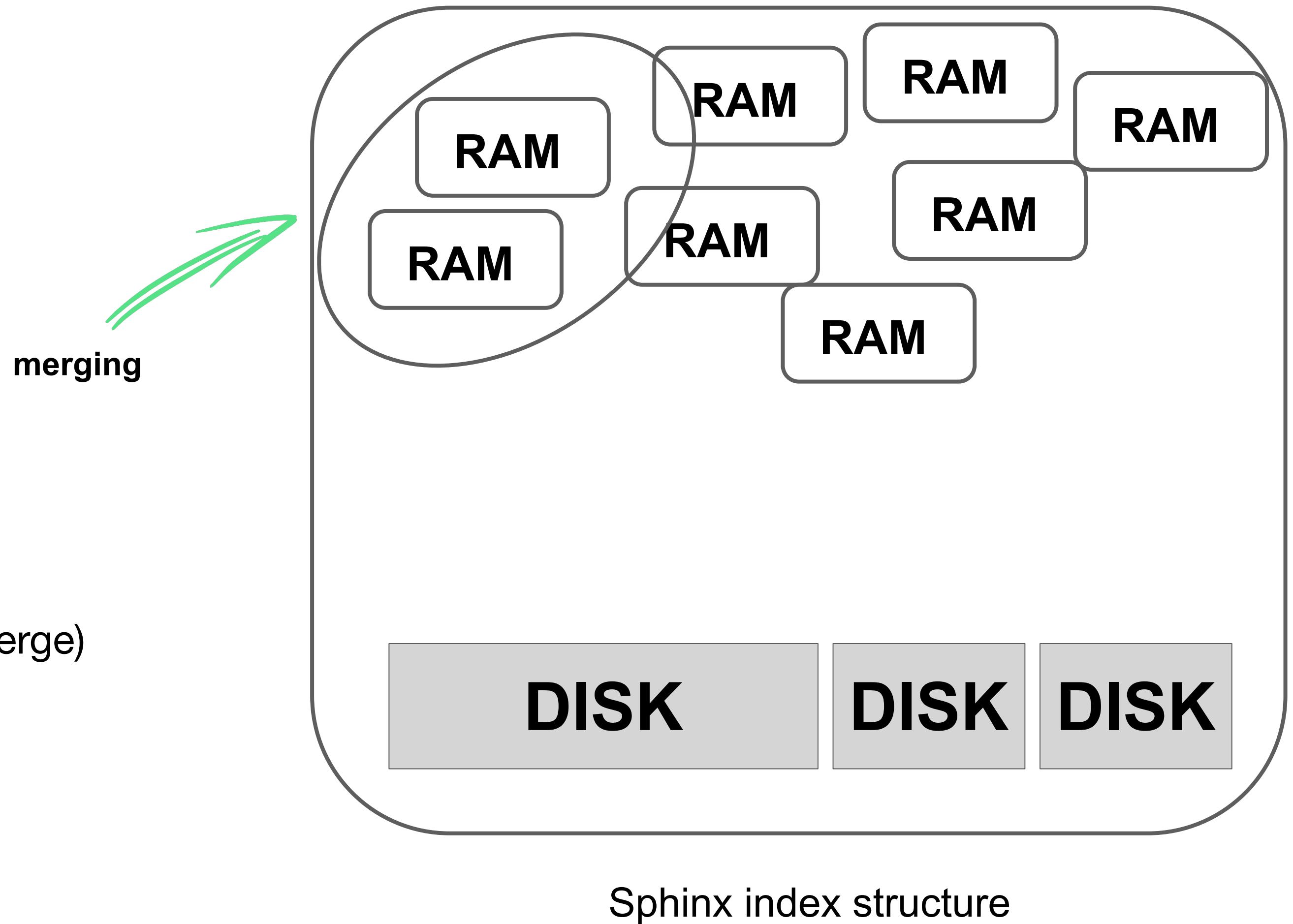
Требования:

- 1k select RPS => хотим хорошее **latency** для каждого запроса;
- **recall** ~0.99 с учетом фильтров (!);
- **realtime** «пишущая» нагрузка: **replace+update**;
- IVF подразумевает довольно дорогую процедуру построения центроидов (a.k.a **Train**), хотим делать ее в **offline**;
- Необходимо подружить векторные поиски с другим функционалом Sphinx: полнотекстовые поиски, «обычные» b-tree-индексы.

Особенности внедрения

Несколько слов об архитектуре Sphinx:

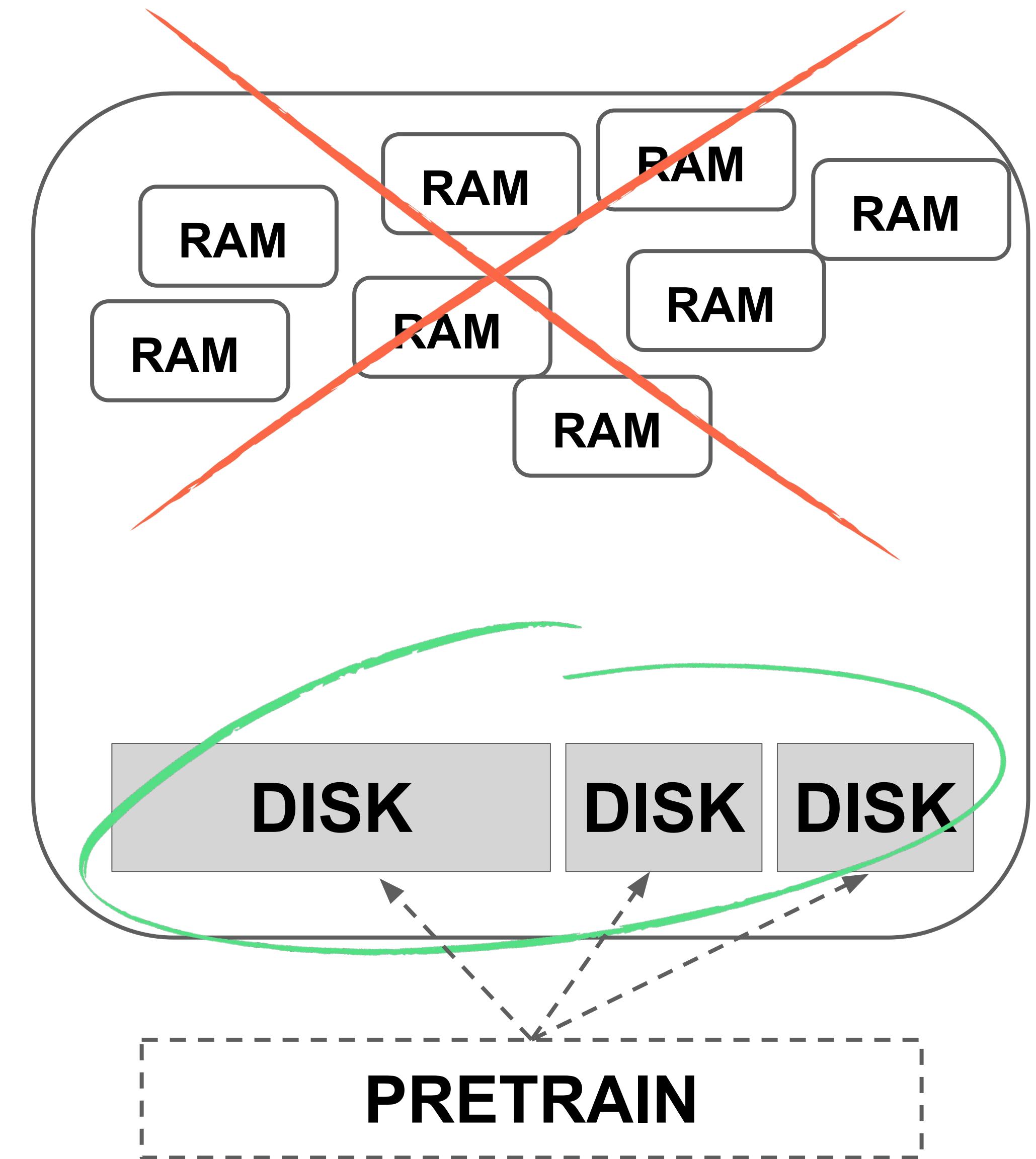
- **RT-index** (таблица) в Sphinx представляет собой несколько сегментов;
- сегменты бывают двух типов: RAM segment, disk segment;
- RAM сегменты: относительно небольшие; часто создаются новые (при replace) и объединяются (merge) старые;
- Disk сегменты: появляются редко, дорогой merge, большой размер.



Особенности внедрения

disk only + pretrain

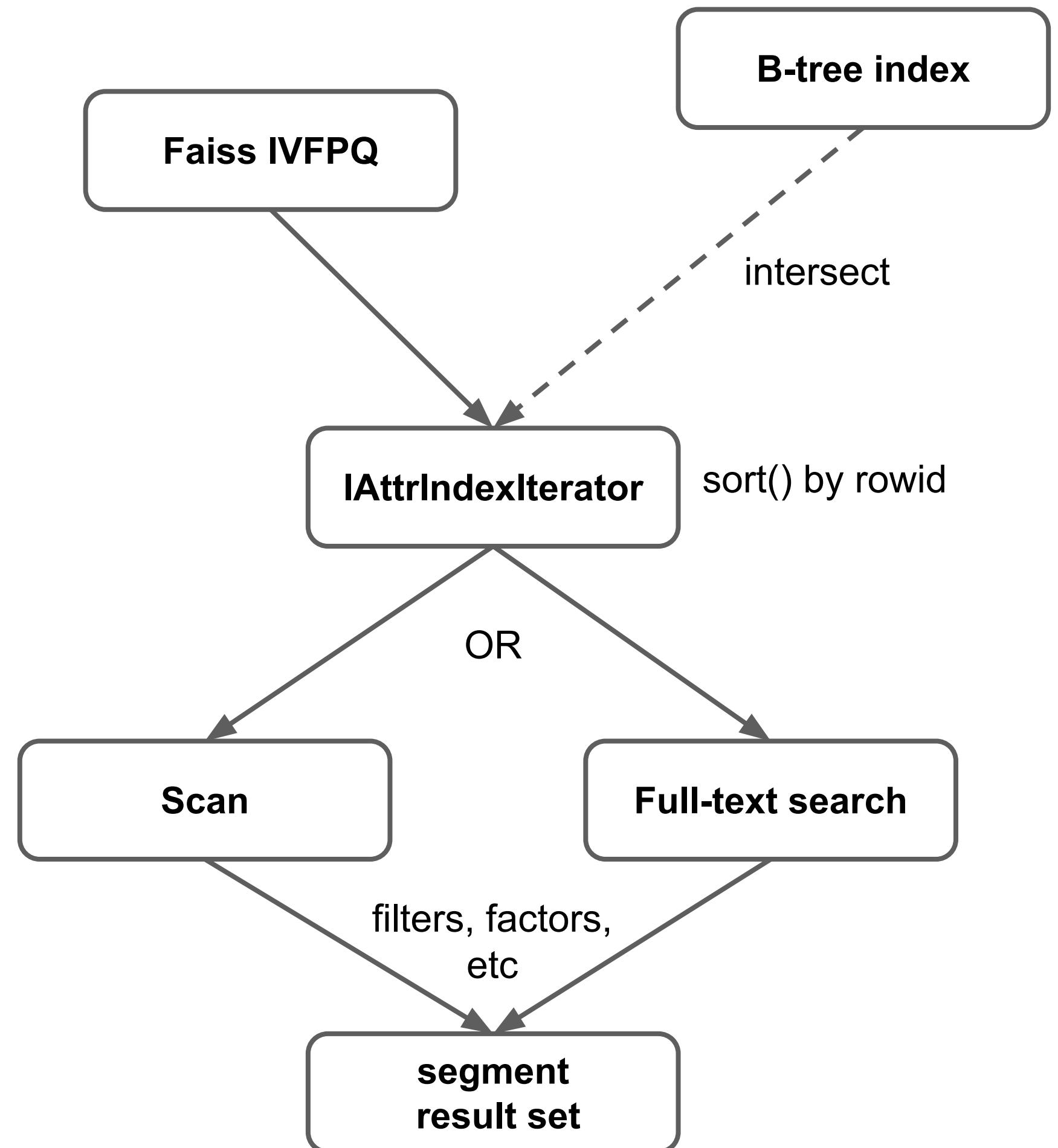
- идея 1: используем Faiss IVFPQ только для дисковых сегментов;
- идея 2: вынесем процедуру Train из realtime, будем делать ее в оффлайн а.к.а **pretrain**;
- pretrain-файл содержит: центроиды IVF кластеров, PQ codebook, прочая мета-информация;
- иными словами – faiss индекс, но **БЕЗ самих векторов**;
- `./indexer pretrain --out [OUT_FILE] [INDEX]`



Особенности внедрения

attribute indexes

- все происходит внутри одного сегмента;
- обворачиваем векторный индекс в generic **attrindex** интерфейс;
- если упрощать, то attrindex по запросу отдает stream (итератор) упорядоченных **rowid** (номер документа внутри сегмента);
- умеем делать **intersect** и **union**;
- итератор можно прокинуть как в **full-text** так и в **scan** path



Особенности внедрения

СВО

- с помощью **Cost-Based Optimizer** определяем, какой набор индексов лучше всего подходит для запроса;
- иногда fullscan лучше похода в faiss!
- а иногда другие атрибутивные индексы более селективны;
- сейчас «гвоздями прибиты» costs для prod-случая (128 координат, dot)

faiss read cost: **12ms** base cost + **1.2ms** per 10k requested docs

vector filter cost: **85ms** per 1M resulting docs

Особенности внедрения

Байка: FAISS vs OpenMP

- Faiss позволяет распараллеливать вычисления (поиск, train, etc);
- под капотом – **OpenMP**;
- Sphinx (searchd) – сам по себе многопоточное приложение;
- в процессе обкатки векторных поисков на проде начали выедать трэды (faiss начинал плодить много потоков на свои нужды);
- решение – вручную лимитировать количество трэдов, доступных OpenMP;
- изначально так: hardcode 1 трэда на поиск, 20 на построение индекса;
- затем добавили SELECT ... OPTION threads=5 .

omp_set_num_threads(iThreads)

thread-local!

Faiss L1 HNSW-flat

2024 год

- добавили поддержку **L1DIST** через Faiss **HNSW-Flat**-индекс
- архитектура из 2020 имени Faiss IVFPQ сделала задачу по добавлению нового типа совсем тривиальной 
- Взлетело неубедительно...
 - иногда (иногда) видели проблемы с recall (абсолютный 0 в некоторых кейсах), потом перестали;
 - у разных людей на тестах слишком сильно разные скорости построения индекса (от 2 до 15 мин), разбираемся;
 - всегда жрёт много памяти (и копия векторов и сам индекс);
 - возможно, виноват “просто” старый Faiss, мы как следует не обновляли с 2021 (оказывается); 
- ...но и к лучшему!!!
 - зато снова занялись векторными индексами поплотнее;
 - иначе ни своего SQ, ни своего HNSW, ни планов обновлений, ни этого доклада не было бы!

Sphinx SQ

Scalar Quantization on Sphinx side

- ОПТИМИЗАЦИЯ для **fullscan**-пути;
- some **SIMD** stuff, of course;
- хорошо подходит для **RAM сегментов**, так как отсутствует тяжелый этап построения индекса;
- имеет смысл применять для **float32**-векторов.

```
__m128i a2 = _mm_and_si128(a1, m); // extract low nibbles
__m128i b2 = _mm_and_si128(b1, m);
a1 = _mm_and_si128(_mm_srli_epi32(a1, 4), m); // extract high nibbles
b1 = _mm_and_si128(_mm_srli_epi32(b1, 4), m);
r1 = _mm_add_epi64(r1, _mm_sad_epu8(a2, b2));
r1 = _mm_add_epi64(r1, _mm_sad_epu8(a1, b1));
```

кусочек SseL1Dist44

Дальнейшие направления работы

1

2

3

4

HNSW: Usearch vs Faiss vs self-made

Проблемы с Faiss HNSW привели к R&D

- Старый (!) Faiss HNSW вероятно (!) проигрывает Usearch HNSW (были произведены замеры)
- ...нооо другие замеры других коллег показывают, что и FAISS хорош!
- USearch требует аккуратной сборки (чувствительность к настройкам компиляции и т. д.), но затем работает бодро!
- библиотеки по дефолту сохраняют вектора, но мы база данных и храним вектора самостоятельно!
Хотим уметь подсовывать свой storage;
- Вероятно будем интегрировать **свое решение** (уже написали прототип), потому, что можем!

Доделки, кастомизация

Дать пользователю большую свободу, починить проблемы

- ХОТИМ дать пользователю:
 - а) Возможность указывать `nprobe` для запросов в IVF индекс;
 - б) Возможность задавать количество центроидов для IVF (now hardcoded 3000);
 - с) PQ settings, other related stuff (on request).
- **Cost-Based Optimizer**: сейчас гвоздями прибиты косты для 128d int8 случая (prod case);
- **Cost-Based Optimizer**: воспроизводимость и согласованность костов;
- обновить Faiss, собрать актуальный бенчмарк, позволяющий быстро сравнить перф/качество.

Заключение

1

2

3

4

Lessons learned

Sphinx vs vectors

- **SIMD** абсолютно критичен:
 - AVX2 практически baseline, SSE2 иногда окей (иногда уже нет)
 - AVX512 пока неоднозначен (на железе из 2020, т.е. 6240R)
- **Faiss** оказался годным general purpose выбором, **рекомендуем**
- Известные публичные ANN benchmarks **не очень релевантны**; своя специфика => свои бенчмарки
- Бенчмарки, Jupiter Notebook-и, датасеты **надо хранить!**
- Векторные поиски == адский **комбинаторный взрыв**; оптимальное решение находится путем проб и ошибок
- Писать свои реализации - OK, но первым шагом - лучше воспользоваться готовой либой!

Summary



Векторный поиск, подходы:
Fullscan, IVF, HNSW;
PQ, SQ.



Много готовых решений под разные случаи.
Библиотеки, специальные СУБД,
поддержка в обычных СУБД.



Faiss.
IVF-PQ хорошо показывает себя на протяжении лет, HNSW пока под вопросом.



Писать код - круто!
Своя реализация SQ показала себя хорошо, будем пробовать внедрить свой HNSW.



Полезные ссылки

- [original paper about PQ](#);
- [faiss paper](#);
- [faiss wiki](#);
- [Avito item2vec Habr](#);
- [обзор векторных СУБД](#);
- доклад на NoML от Андрея Волкова (rus): [YouTube](#), [слайды](#);
- доклад от Yusuke Matsui (eng): [YouTube](#), [слайды](#);
- [материал](#) от Pinecone про векторные поиски.
- [ANN-benchmarks](#)

слайды



That's it!
Готов ответить на вопросы

слайды

