

Векторный поиск в PostgreSQL: pgvector изнутри

Дарья Барсукова @brskv_dm



Дарья Барсукова

- Новосибирский государственный университет
- Занимаюсь производительностью и нагрузочным тестированием PostgreSQL в компании Postgres Pro



Postgres Professional – разработчик самой популярной российской СУБД

11 лет

на рынке с 2015

ТОП-5

в мире по вкладу
в PostgreSQL
(100+ патчей ежегодно)

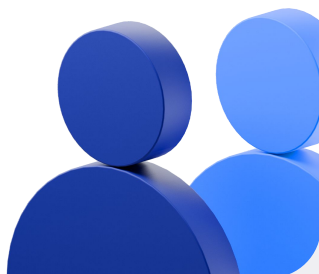
№1

в России среди
разработчиков СУБД,
по данным ЦСР (2024)



500 +

специалистов в команде,
включая Major Contributors
PostgreSQL



3000+ заказчиков

крупнейшие госкомпании,
банки, телеком, критическая
инфраструктура



О чем расскажу

Путь доклада: от иллюзии простоты до инженерных компромиссов на реальных данных

О чем расскажу

Путь доклада: от иллюзии простоты до инженерных компромиссов на реальных данных

01 Постановка задачи: иллюзия простоты

О чем расскажу

Путь доклада: от иллюзии простоты до инженерных компромиссов на реальных данных

01 Постановка задачи: иллюзия простоты

02 Анатомия pgvector:
Access Method API

О чем расскажу

Путь доклада: от иллюзии простоты до инженерных компромиссов на реальных данных

01 Постановка задачи: иллюзия простоты

02 Анатомия pgvector:
Access Method API

03 HNSW: граф и
погоня за указателями

О чем расскажу

Путь доклада: от иллюзии простоты до инженерных компромиссов на реальных данных

01 Постановка задачи: иллюзия простоты

04 IVFFlat: стриминг против прыжков и SIMD

02 Анатомия pgvector:
Access Method API

03 HNSW: граф и
погоня за указателями

О чем расскажу

Путь доклада: от иллюзии простоты до инженерных компромиссов на реальных данных

01 Постановка задачи: иллюзия простоты

04 IVFFlat: стриминг против прыжков и SIMD

02 Анатомия pgvector: Access Method API

05 Практика: perf и загадка о потерянном recall

03 HNSW: граф и погоня за указателями

О чем расскажу

Путь доклада: от иллюзии простоты до инженерных компромиссов на реальных данных

01 Постановка задачи: иллюзия простоты

04 IVFFlat: стриминг против прыжков и SIMD

02 Анатомия pgvector: Access Method API

05 Практика: perf и загадка о потерянном recall

03 HNSW: граф и погоня за указателями

06 Рекомендации

Постановка задачи: иллюзия простоты

От букв к смыслам

От букв к смыслам

Классический подход:

- Полнотекстовый поиск

От букв к смыслам

Классический подход:

- Полнотекстовый поиск
- Ищет совпадение **БУКВ**

Классический подход:

- Полнотекстовый поиск
- Ищет совпадение **БУКВ**
- Ключевые слова,
инвертированный индекс

От букв к смыслам

Классический подход:

- Полнотекстовый поиск
- Ищет совпадение **БУКВ**
- Ключевые слова, инвертированный индекс

Современная реальность:

- Векторный поиск + полнотекстовый поиск

Классический подход:

- Полнотекстовый поиск
- Ищет совпадение **БУКВ**
- Ключевые слова, инвертированный индекс

Современная реальность:

- Векторный поиск + полнотекстовый поиск
- Ищет совпадение **СМЫСЛОВ**

Классический подход:

- Полнотекстовый поиск
- Ищет совпадение **БУКВ**
- Ключевые слова, инвертированный индекс

Современная реальность:

- Векторный поиск + полнотекстовый поиск
- Ищет совпадение **СМЫСЛОВ**
- Плотные эмбединги

Классический подход:

- Полнотекстовый поиск
- Ищет совпадение **БУКВ**
- Ключевые слова, инвертированный индекс

Современная реальность:

- Векторный поиск + полнотекстовый поиск
- Ищет совпадение **СМЫСЛОВ**
- Плотные эмбединги

Цель: минимальная задержка (latency) на миллионах объектов

Пример векторного поиска

Эмбединг — массив чисел, который получается после преобразования текста языковой моделью.

Интерстеллар

Прибытие

Комедия



word2vec

Пример векторного поиска

Эмбединг — массив чисел, который получается после преобразования текста языковой моделью.

Sci-Fi	Драма	Экшн	Эпич.	Филос.	Юмор	Роман.	Напряж.
--------	-------	------	-------	--------	------	--------	---------

Интерстеллар

Прибытие

Комедия

Для наглядности показаны 8 условных семантических признаков; реальные эмбединги имеют сотни и тысячи измерений.



word2vec

Пример векторного поиска

Эмбединг — массив чисел, который получается после преобразования текста языковой моделью.



Для наглядности показаны 8 условных семантических признаков; реальные эмбединги имеют сотни и тысячи измерений.



word2vec

Пример векторного поиска

Эмбединг — массив чисел, который получается после преобразования текста языковой моделью.

Метрики для сравнения векторов:

- Cosine similarity

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$



word2vec

Пример векторного поиска

Эмбединг — массив чисел, который получается после преобразования текста языковой моделью.

Метрики для сравнения векторов:

- Cosine similarity

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

- Евклидова
- Манхэттенская
- Hamming



word2vec

Пример векторного поиска

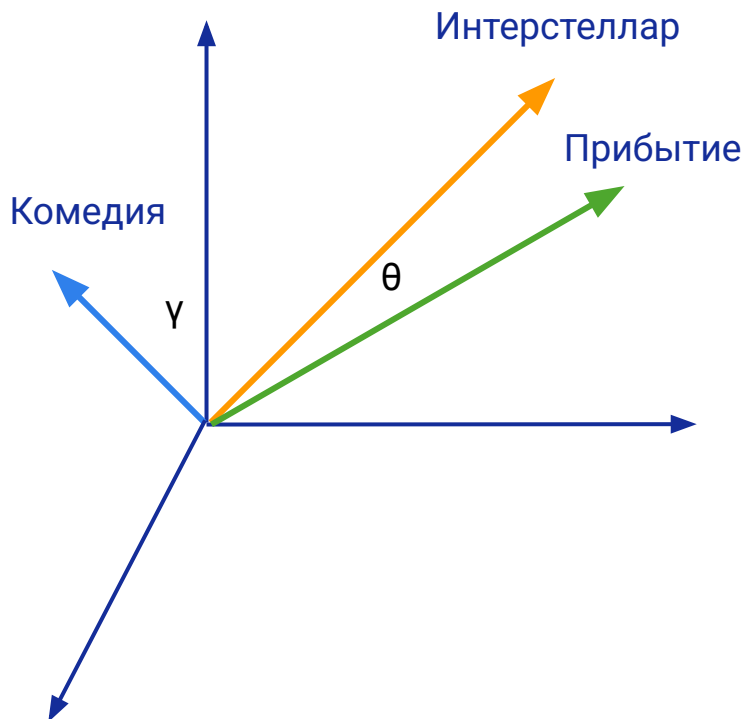
Эмбединг — массив чисел, который получается после преобразования текста языковой моделью.

Метрики для сравнения векторов:

- Cosine similarity

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

- Евклидова
- Манхэттенская
- Hamming



$\cos(\text{Интерстеллар}, \text{Прибытие}) \approx 0.993$
 $\cos(\text{Интерстеллар}, \text{Комедия}) \approx 0.332$



word2vec

Что с инфраструктурой?

Дилемма архитектора на старте

Путь 1. Специализированные VectorDB

- Новые лицензии и стек

Дилемма архитектора на старте

Путь 1. Специализированные VectorDB

- Новые лицензии и стек
- Боль синхронизации данных (Dual-write)

Дилемма архитектора на старте

Путь 1. Специализированные VectorDB

- Новые лицензии и стек
- Боль синхронизации данных (Dual-write)
- Отдельный мониторинг и бэкапы

Дилемма архитектора на старте

Путь 1. Специализированные VectorDB

- Новые лицензии и стек
- Боль синхронизации данных (Dual-write)
- Отдельный мониторинг и бэкапы
- Инфраструктурный оверхед

Дилемма архитектора на старте

Путь 1. Специализированные VectorDB

- Новые лицензии и стек
- Боль синхронизации данных (Dual-write)
- Отдельный мониторинг и бэкапы
- Инфраструктурный оверхед

Путь 2. PostgreSQL + pgvector

- Привычная экосистема

Дилемма архитектора на старте

Путь 1. Специализированные VectorDB

- Новые лицензии и стек
- Боль синхронизации данных (Dual-write)
- Отдельный мониторинг и бэкапы
- Инфраструктурный оверхед

Путь 2. PostgreSQL + pgvector

- Привычная экосистема
- Стандартные SQL-запросы

Дилемма архитектора на старте

Путь 1. Специализированные VectorDB

- Новые лицензии и стек
- Боль синхронизации данных (Dual-write)
- Отдельный мониторинг и бэкапы
- Инфраструктурный оверхед

Путь 2. PostgreSQL + pgvector

- Привычная экосистема
- Стандартные SQL-запросы
- Жесткие гарантии ACID из коробки

Дилемма архитектора на старте

Путь 1. Специализированные VectorDB

- Новые лицензии и стек
- Боль синхронизации данных (Dual-write)
- Отдельный мониторинг и бэкапы
- Инфраструктурный оверхед

Путь 2. PostgreSQL + pgvector

- Привычная экосистема
- Стандартные SQL-запросы
- Жесткие гарантии ACID из коробки
- Вектор – это просто колонка в таблице

Задача решена?

Не совсем...

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

10x

во столько раз выросла
база данных в тесте

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

10x

во столько раз выросла
база данных в тесте

46x

во столько раз выросло
время построения индекса

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

10x

во столько раз выросла
база данных в тесте

46x

во столько раз выросло
время построения индекса

Падение QPS с ростом базы

100K



2850 qps

На примере OpenAI text-embedding-3-small (1536d)

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

10x

во столько раз выросла
база данных в тесте

46x

во столько раз выросло
время построения индекса

Падение QPS с ростом базы



На примере OpenAI text-embedding-3-small (1536d)

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

10x

во столько раз выросла
база данных в тесте

46x

во столько раз выросло
время построения индекса

Падение QPS с ростом базы



На примере OpenAI text-embedding-3-small (1536d)

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

10x

во столько раз выросла
база данных в тесте

46x

во столько раз выросло
время построения индекса

Падение QPS с ростом базы



На примере OpenAI text-embedding-3-small (1536d)

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

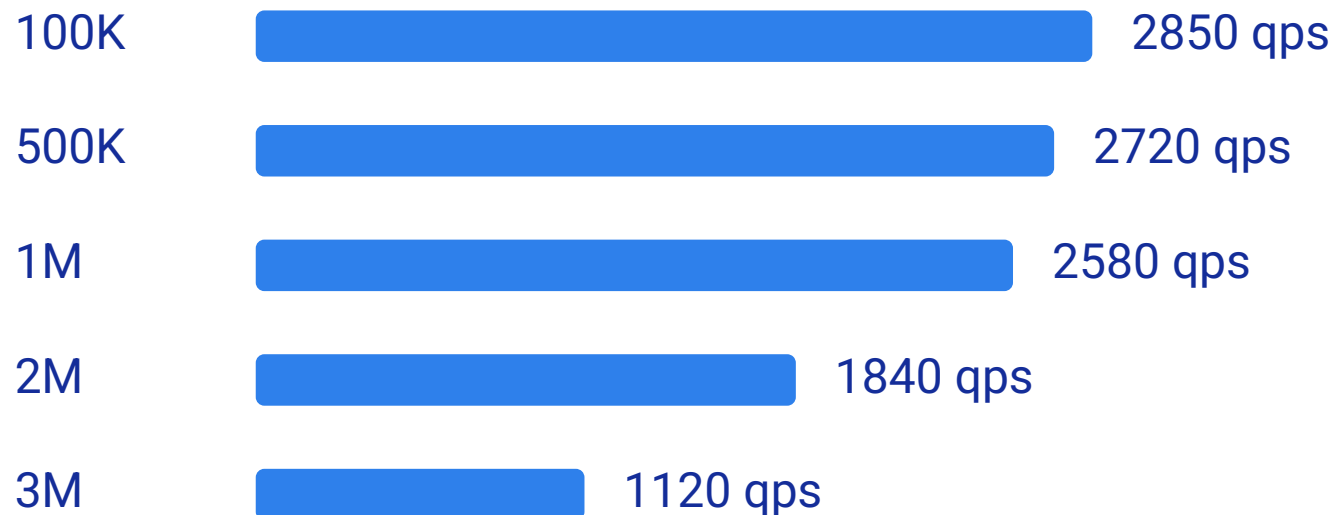
10x

во столько раз выросла
база данных в тесте

46x

во столько раз выросло
время построения индекса

Падение QPS с ростом базы



На примере OpenAI text-embedding-3-small (1536d)

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

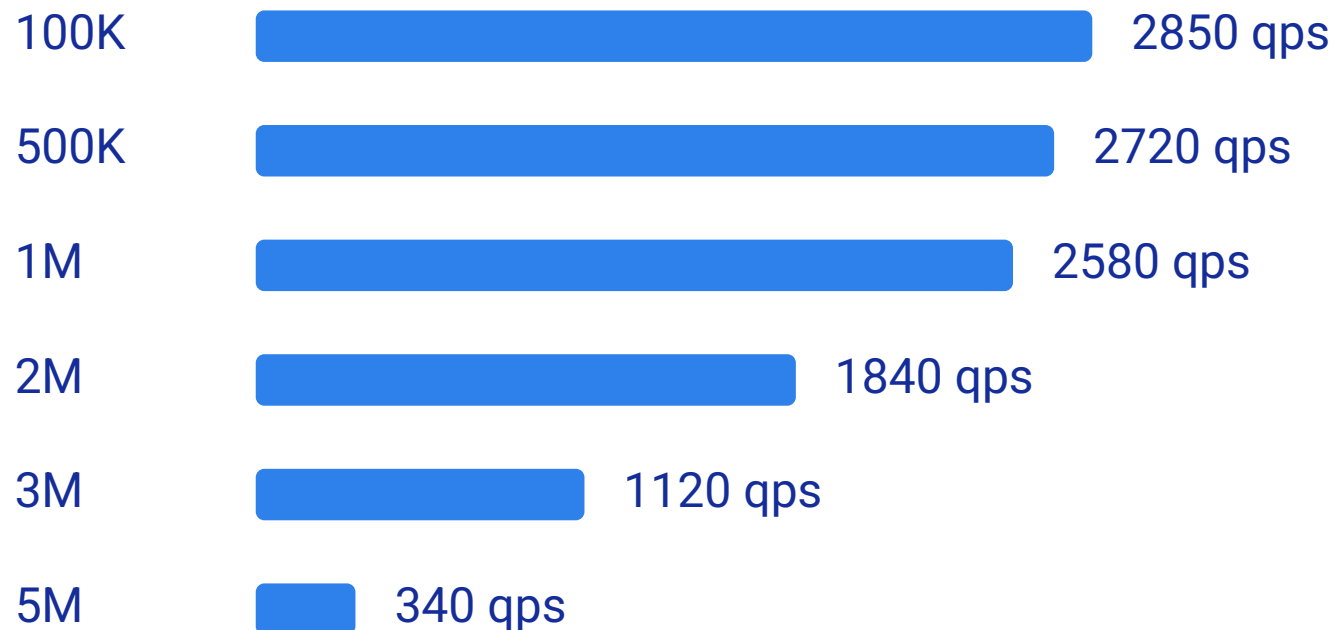
10x

во столько раз выросла
база данных в тесте

46x

во столько раз выросло
время построения индекса

Падение QPS с ростом базы



На примере OpenAI text-embedding-3-small (1536d)

Столкновение с физикой железа

15 мин

от установки расширения
до работающего RAG-прототипа

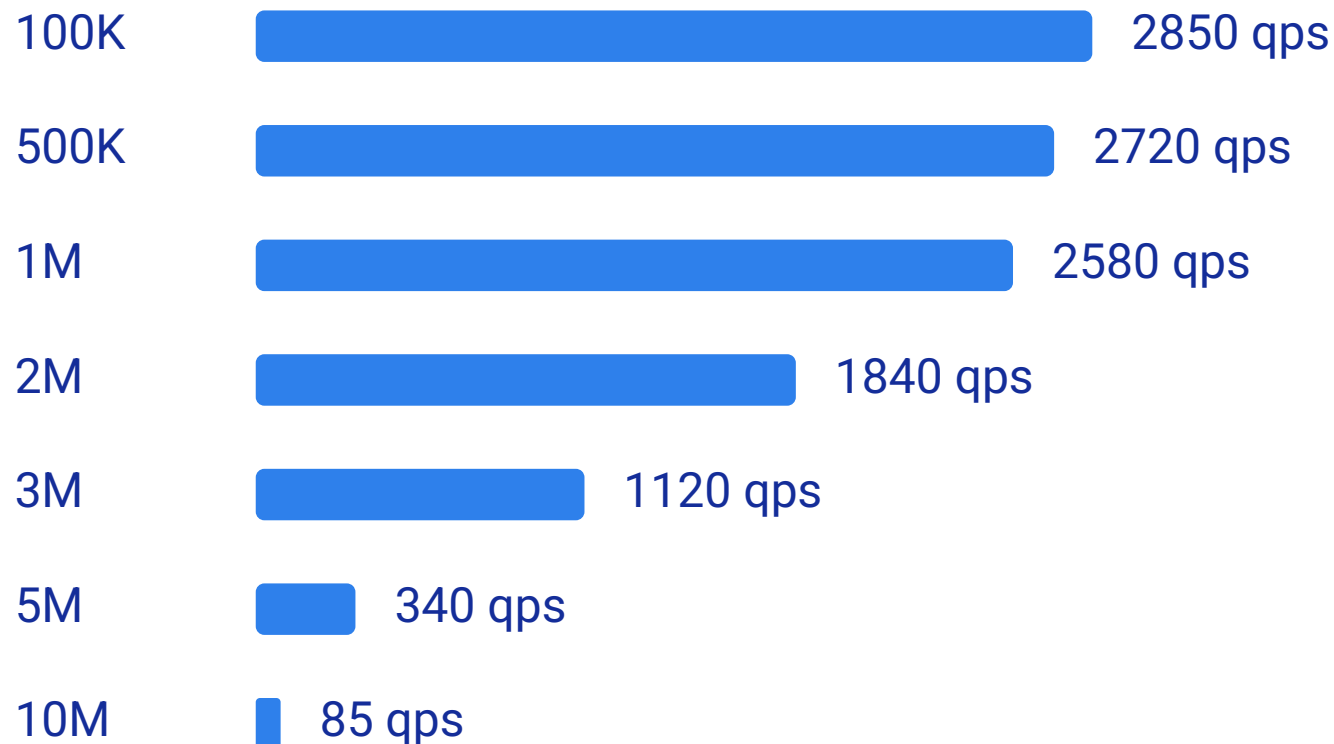
10x

во столько раз выросла
база данных в тесте

46x

во столько раз выросло
время построения индекса

Падение QPS с ростом базы



На примере OpenAI text-embedding-3-small (1536d)

Что именно должно быть
быстрым?

Что ломается первым?

Запрос

```
SELECT
  id,
  title,
  platform
FROM movies
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```

- 5 млн векторов (размерность D)
- найти top-100 за миллисекунды/десятки мс

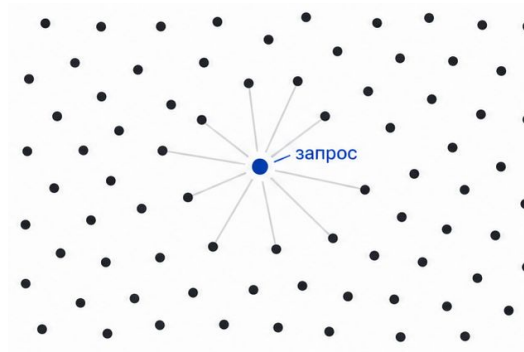
Что ломается первым?

Запрос

```
SELECT
  id,
  title,
  platform
FROM movies
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```

- 5 млн векторов (размерность D)
- найти top-100 за миллисекунды/десятки мс

Полный перебор



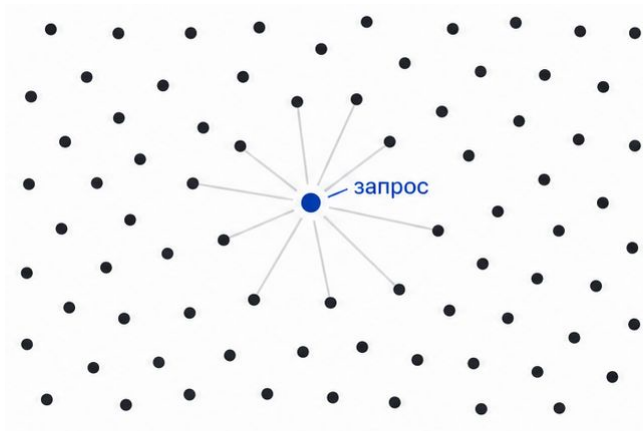
- Полное сравнение всех векторов

Точный поиск требует сравнить запрос с каждым вектором: $O(N \cdot D)$. На миллионах объектов это слишком дорого.

Так может искать иначе?

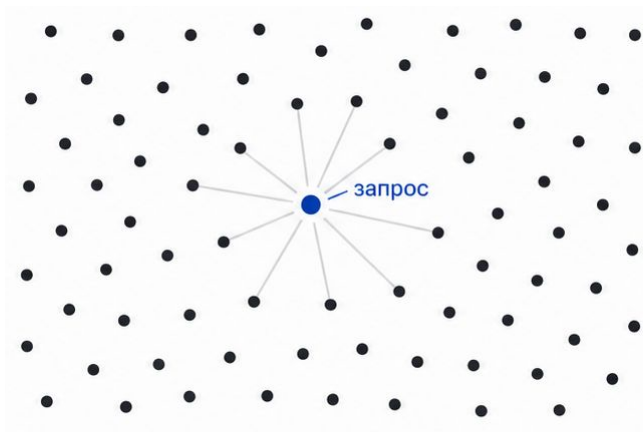
KNN (k ближайших соседей):

- точный поиск
- полный перебор всех векторов
- recall 100%
- медленно



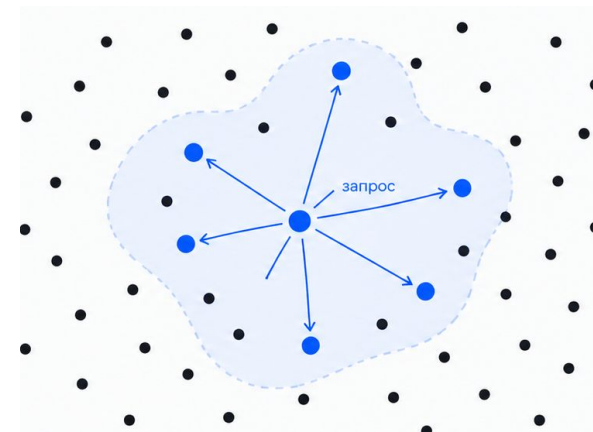
KNN (k ближайших соседей):

- точный поиск
- полный перебор всех векторов
- recall 100%
- медленно



ANN (примерно ближайшие соседи):

- приблизительный поиск
- смотрим только ближайшие к запросу
- recall $\leq 100\%$
- быстро



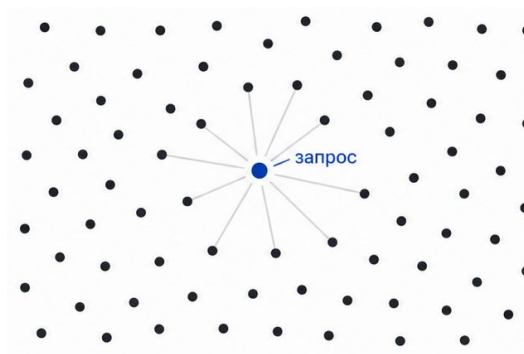
Что ломается первым?

Запрос

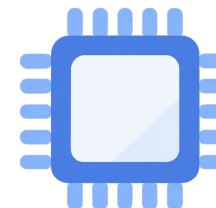
```
SELECT
  id,
  title,
  platform
FROM movies
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```

- 5 млн векторов (размерность D)
- найти top-100 за миллисекунды/десятки мс

Перебор

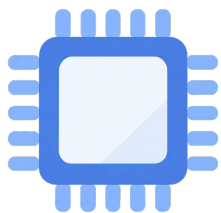


pgvector



Точный поиск требует сравнить запрос с каждым вектором: $O(N \cdot D)$. На миллионах объектов это слишком дорого.

pgvector

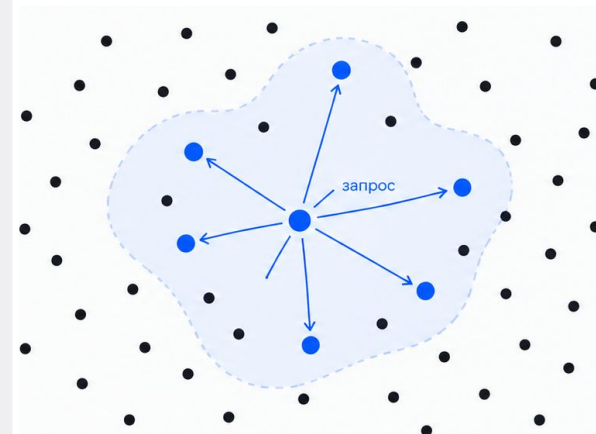


+

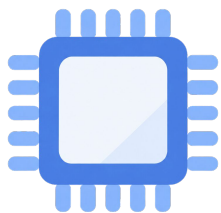
?

=

Неполный
перебор

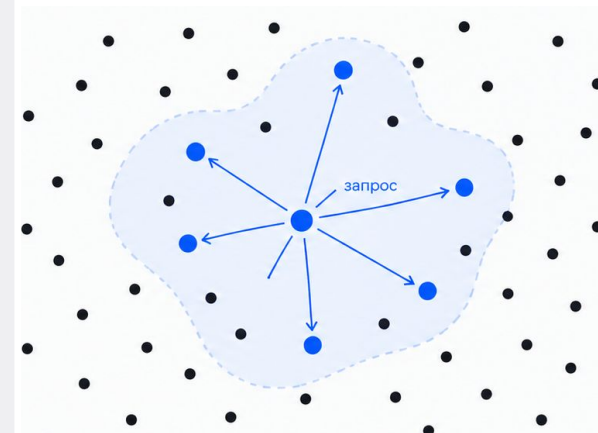


pgvector



+ индекс =

Неполный
перебор



Что за индексы?

IVFFlat



HNSW



IVFFlat

(Инвертированный файл с плоским сжатием)



Идея:

Разбить N-мерное пространство
на ячейки Вороного
(кластеры вокруг центроидов)

Принцип работы:

Ищем не по всей базе, а только внутри
probes ближайших кластеров

HNSW

(Иерархический навигационный малый мир)



Идея:

Построить многослойный граф, в котором каждый вектор — узел, а связи ведут к его ближайшим соседям.

Принцип работы:

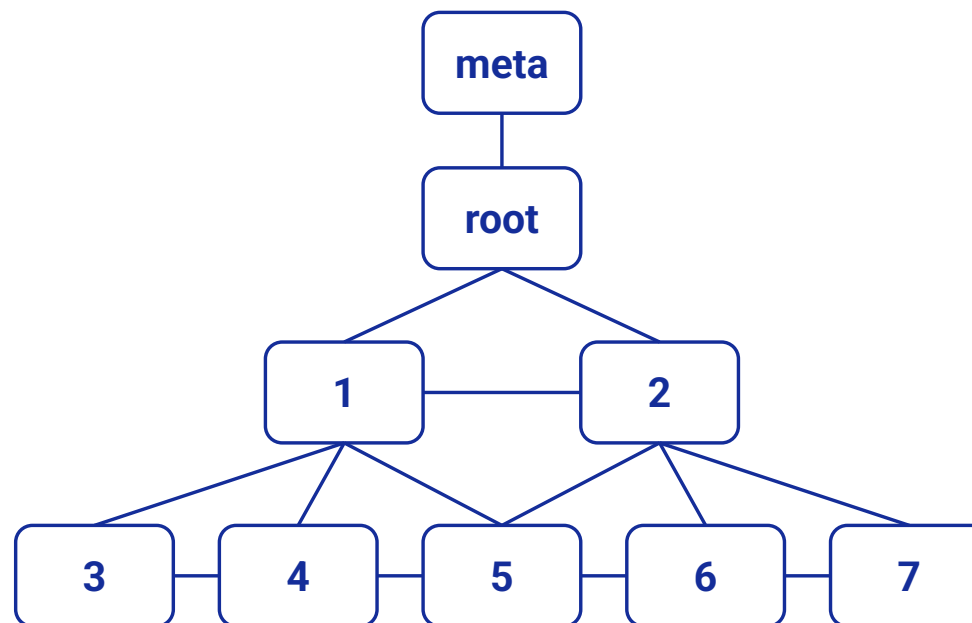
Поиск стартует с верхнего разреженного слоя и жадно «спускается» к нужной области, спускаясь слой за слоем и сужая круг соседей.

PostgreSQL, индексы...

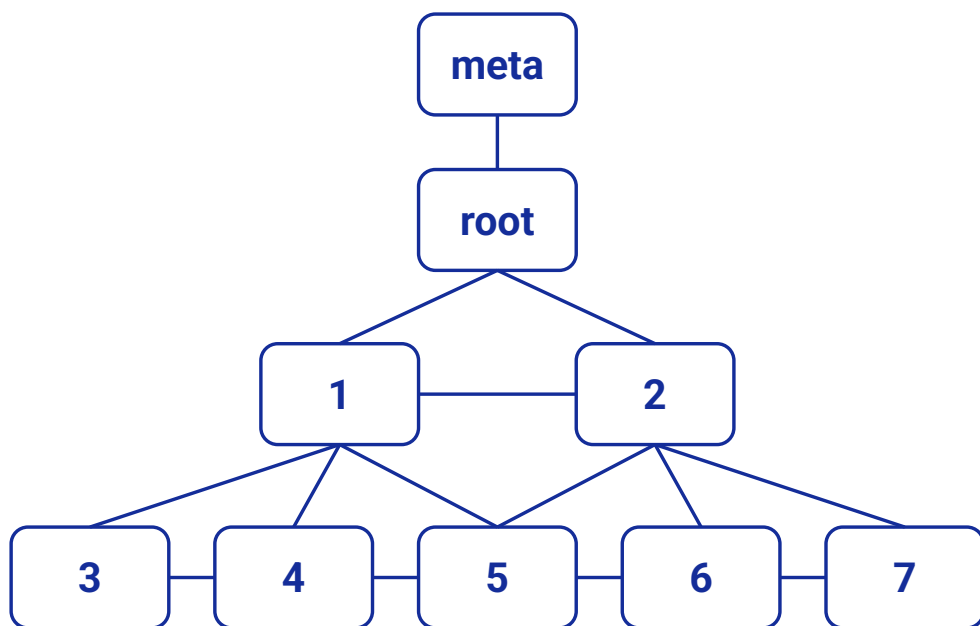
PostgreSQL, индексы...



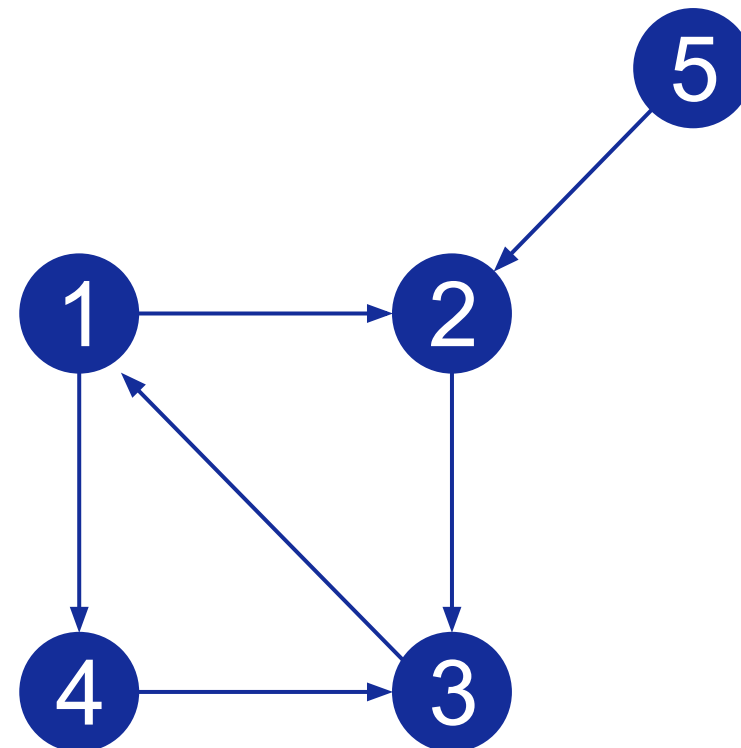
PostgreSQL, индексы...



PostgreSQL, индексы...



>>



Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree						
HNSW (граф)						
IVFFlat (центроиды)						

Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree	a < b a = b a > b					
HNSW (граф)	Метрика расстояния					
IVFFlat (центроиды)						

Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree	a < b a = b a > b	100%точное совпадение				
HNSW (граф)	Метрика расстояния	приближенная (recall <= 100%)				
IVFFlat (центроиды)						

Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree	a < b a = b a > b	100%точное совпадение	32–40 байт (ключ + Tuple ID)			
HNSW (граф)	Метрика расстояния	приближенная (recall <= 100%)	~6 Кб (вектор d1536)	~268 байт (служебные)		
IVFFlat (центроиды)				~12 байт (служебные)		

Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree	a < b a = b a > b	100%точное совпадение	32–40 байт (ключ + Tuple ID)		O(log N), простая	
HNSW (граф)	Метрика расстояния	приближенная (recall <= 100%)	~6 Кб (вектор d1536)	~268 байт (служебные)	O(log N)	
IVFFlat (центроиды)				~12 байт (служебные)	O(N/lists * probes + probes)	

Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree	a < b a = b a > b	100%точное совпадение	32–40 байт (ключ + Tuple ID)		O(log N), простая	O(NlogN)
HNSW (граф)	Метрика расстояния	приближенная (recall <= 100%)	~6 Кб (вектор d1536)	~268 байт (служебные)	O(log N)	O(NlogN)
IVFFlat (центроиды)				~12 байт (служебные)	O(N/lists * probes + probes)	O(lists * N * iter)

Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree	$a < b$ $a = b$ $a > b$	100%точное совпадение	32–40 байт (ключ + Tuple ID)		$O(\log N)$, простая	$O(N \log N)$
HNSW (граф)	Метрика расстояния	приближенная (recall $\leq 100\%$)	~6 Кб (вектор d1536)	~268 байт (служебные)	$O(\log N)$	$O(N \log N)$
IVFFlat (центроиды)				~12 байт (служебные)	$O(N / \text{lists} * \text{probes} + \text{probes})$	$O(\text{lists} * N * \text{iter})$

200 +

ключей B-Tree
на одной странице 8 КБ

Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree	$a < b$ $a = b$ $a > b$	100%точное совпадение	32–40 байт (ключ + Tuple ID)		$O(\log N)$, простая	$O(N \log N)$
HNSW (граф)	Метрика расстояния	приближенная (recall $\leq 100\%$)	~6 Кб (вектор d1536)	~268 байт (служебные)	$O(\log N)$	$O(N \log N)$
IVFFlat (центроиды)				~12 байт (служебные)	$O(N/\text{lists} * \text{probes} + \text{probes})$	$O(\text{lists} * N * \text{iter})$

200 +

ключей B-Tree
на одной странице 8 КБ

1

вектор на одной
странице 8 КБ

Почему B-Tree не болит, а векторы страдают?

Критерий	Сравнение	Точность	Запись		Поиск	Построение
B-Tree	$a < b$ $a = b$ $a > b$	100%точное совпадение	32–40 байт (ключ + Tuple ID)		$O(\log N)$, простая	$O(N \log N)$
HNSW (граф)	Метрика расстояния	приближенная (recall $\leq 100\%$)	~6 Кб (вектор d1536)	~268 байт (служебные)	$O(\log N)$	$O(N \log N)$
IVFFlat (центроиды)				~12 байт (служебные)	$O(N/\text{lists} * \text{probes} + \text{probes})$	$O(\text{lists} * N * \text{iter})$

200 +

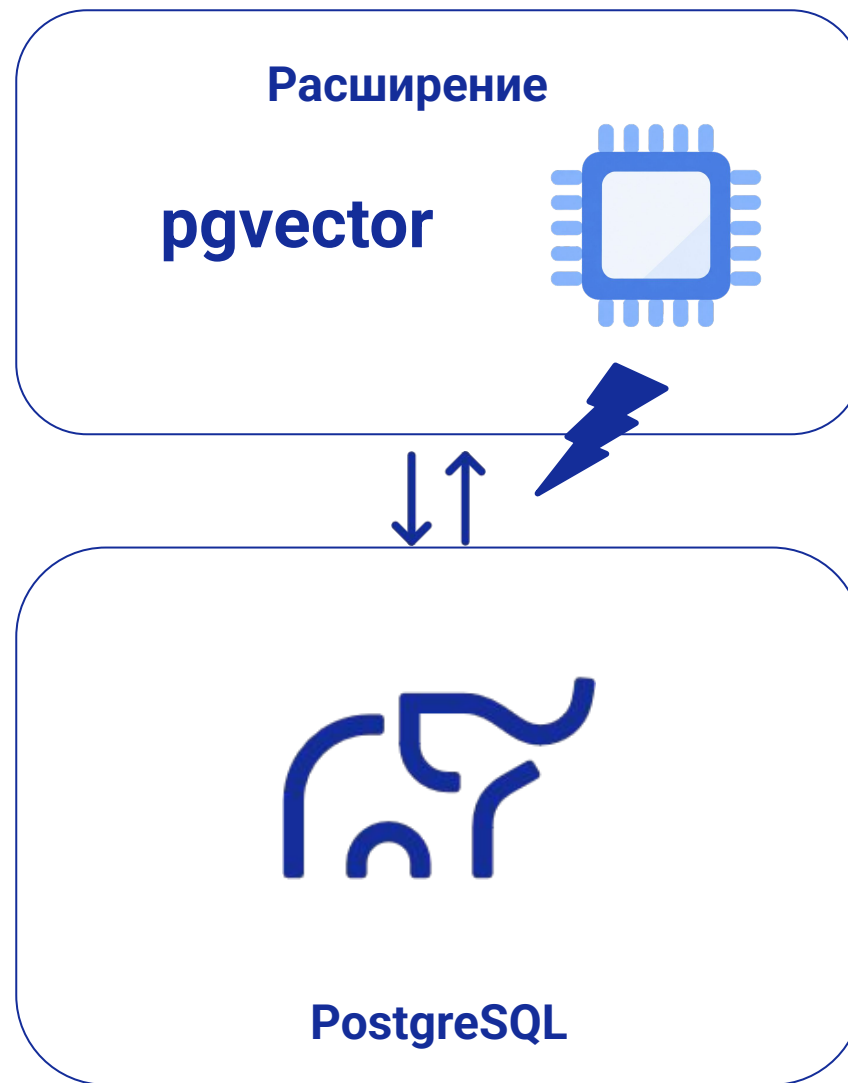
ключей B-Tree
на одной странице 8 КБ

1

вектор на одной
странице 8 КБ

96

кэш-линий CPU нужно
прочитать на 1 вектор



Анатомия pgvector: Access Method API

Как Postgres «дружит» с векторами

Системная интеграция: регистрация INDEX ACCESS METHOD



1. Регистрация в системном каталоге pg_am:

Операторы расстояния (<->, <=>, <#>) → Вызов HNSW / IVFFlat

Как Postgres «дружит» с векторами

Системная интеграция: регистрация INDEX ACCESS METHOD



1. Регистрация в системном каталоге pg_am:

Операторы расстояния (<->, <=>, <#>) → Вызов HNSW / IVFFlat

2. Исполнение: Модель **Volcano**

Как Postgres «дружит» с векторами

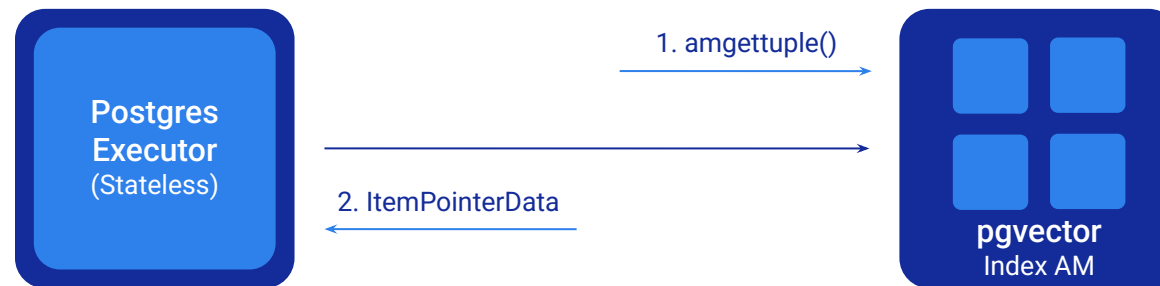
Системная интеграция: регистрация INDEX ACCESS METHOD



1. Регистрация в системном каталоге pg_am:

Операторы расстояния (<->, <=>, <#>) → Вызов HNSW / IVFFlat

2. Исполнение: Модель **Volcano**



Как Postgres «дружит» с векторами

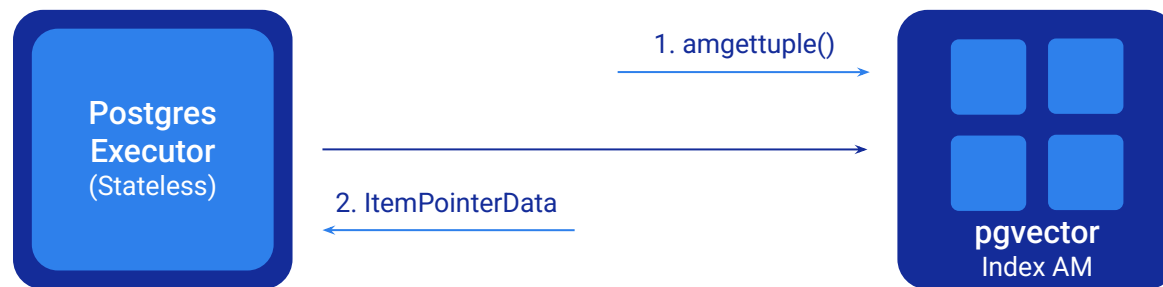
Системная интеграция: регистрация INDEX ACCESS METHOD



1. Регистрация в системном каталоге pg_am:

Операторы расстояния (<->, <=>, <#>) → Вызов HNSW / IVFFlat

2. Исполнение: Модель **Volcano**



Ключевая проблема: ехеситор «забывает» всё между вызовами.
Вся ответственность за хранение позиции графа лежит на pgvector.

Как Postgres «дружит» с векторами

Внутренний контекст сканирования



Как Postgres «дружит» с векторами

Внутренний контекст сканирования



Каждый раз pgvector тащит за собой в памяти:

1. Pairing Heap – 2 приоритетных очереди

- **C** – кандидаты для дальнейшего исследования (min-heap по расстоянию)
- **W** – уже найденные ближайшие (max-heap по расстоянию)

Как Postgres «дружит» с векторами

Внутренний контекст сканирования



Каждый раз pgvector тащит за собой в памяти:

1. Pairing Heap – 2 приоритетных очереди

- **C** – кандидаты для дальнейшего исследования (min-heap по расстоянию)
- **W** – уже найденные ближайшие (max-heap по расстоянию)

2. Visited Hash Table – защита от закливания при обходе связей

Как Postgres «дружит» с векторами

Внутренний контекст сканирования



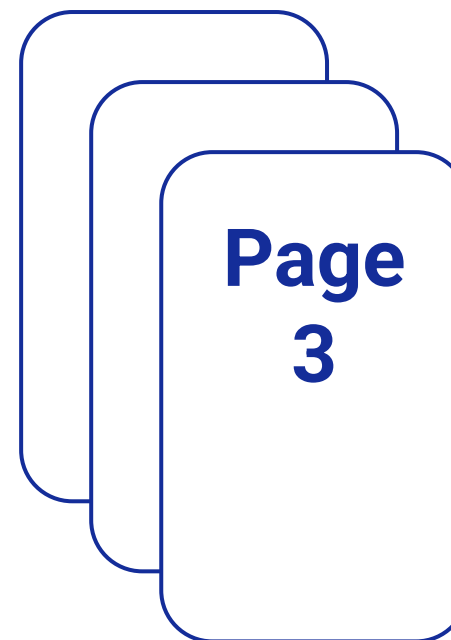
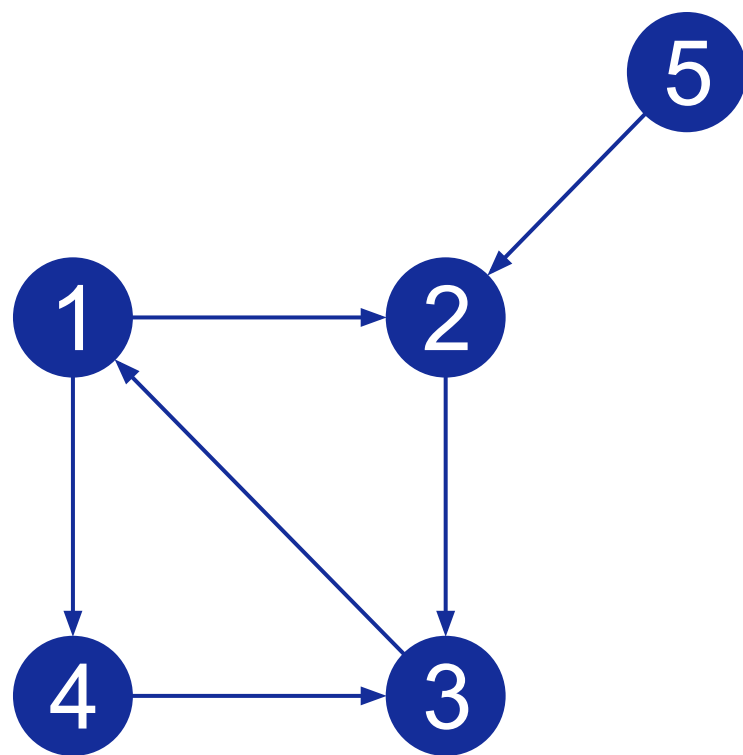
Каждый раз pgvector тащит за собой в памяти:

1. Pairing Heap – 2 приоритетных очереди

- **C** – кандидаты для дальнейшего исследования (min-heap по расстоянию)
- **W** – уже найденные ближайшие (max-heap по расстоянию)

2. Visited Hash Table – защита от закливания при обходе связей

Следствие: постоянная аллокация, протаскивание по памяти и обновление метаданных на каждый шаг конвейера. Процессор тратит такты на служебный контекст.



Хранение: вектор в мире 8-килобайтных страниц

Page 0 (Метастраница)

Заголовок:

- magic: 0xA953A953
- version: 1
- dimensions: 1536

Граф:

- m: 16
- ef_construction: 200
- entry point: 42 (TID)

Хранение: вектор в мире 8-килобайтных страниц

Page 0 (Метастраница)

Заголовок:

- magic: 0xA953A953
- version: 1
- dimensions: 1536

Граф:

- m: 16
- ef_construction: 200
- entry point: 42 (TID)

Вектор (ID: 42)

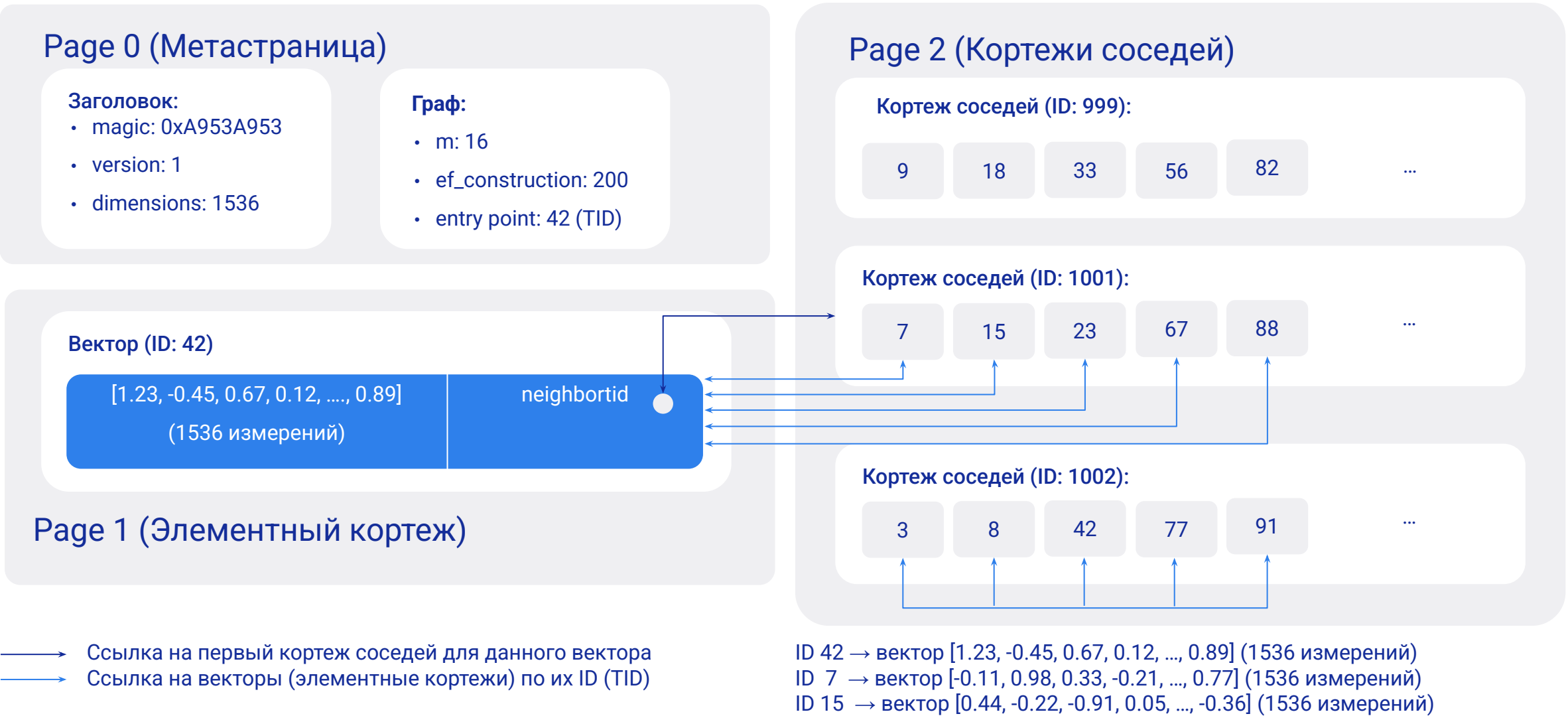
[1.23, -0.45, 0.67, 0.12, ..., 0.89]
(1536 измерений)

neighbortid



Page 1 (Элементный кортеж)

Хранение: вектор в мире 8-килобайтных страниц



Хранение: вектор в мире 8-килобайтных страниц

«Интерстеллар» - запрос

Хранение: вектор в мире 8-килобайтных страниц

«Интерстеллар» - запрос

1

Страница А (Элемент)

Текущий узел: «Дюна»

→ Указатель neighbortid -> Page B

Вычислить distance

← («Интерстеллар», «Дюна»)

Disc Read #1

«Где соседи?»

Хранение: вектор в мире 8-килобайтных страниц

«Интерстеллар» - запрос

1

Страница А (Элемент)

Текущий узел: «Дюна»

→ Указатель neighbortid -> Page B

Вычислить distance

← («Интерстеллар», «Дюна»)

Disc Read #1

«Где соседи?»

2

Страница В (Соседи)

→ Page C («Прибытие»), Off 1

→ Page D («Марсианин»), Off 3

→ Page E («Гравитация»), Off 2

Выбрать ближайшего
← непосещенного

Disc Read #2

«Идем к Page C»

Хранение: вектор в мире 8-килобайтных страниц

«Интерстеллар» - запрос

1

Страница А (Элемент)

Текущий узел: «Дюна»

→ Указатель neighbortid -> Page B

Вычислить distance
← («Интерстеллар», «Дюна»)

Disc Read #1

«Где соседи?»

2

Страница В (Соседи)

→ Page C («Прибытие»), Off 1

→ Page D («Марсианин»), Off 3

→ Page E («Гравитация»), Off 2

Выбрать ближайшего
← непосещенного

Disc Read #2

«Идем к Page C»

3

Страница С (Элемент)

Текущий узел: «Прибытие»

→ указатель neighbortid -> Page F

Новое вычисление
← distance(«Интерстеллар»,
«Прибытие»)

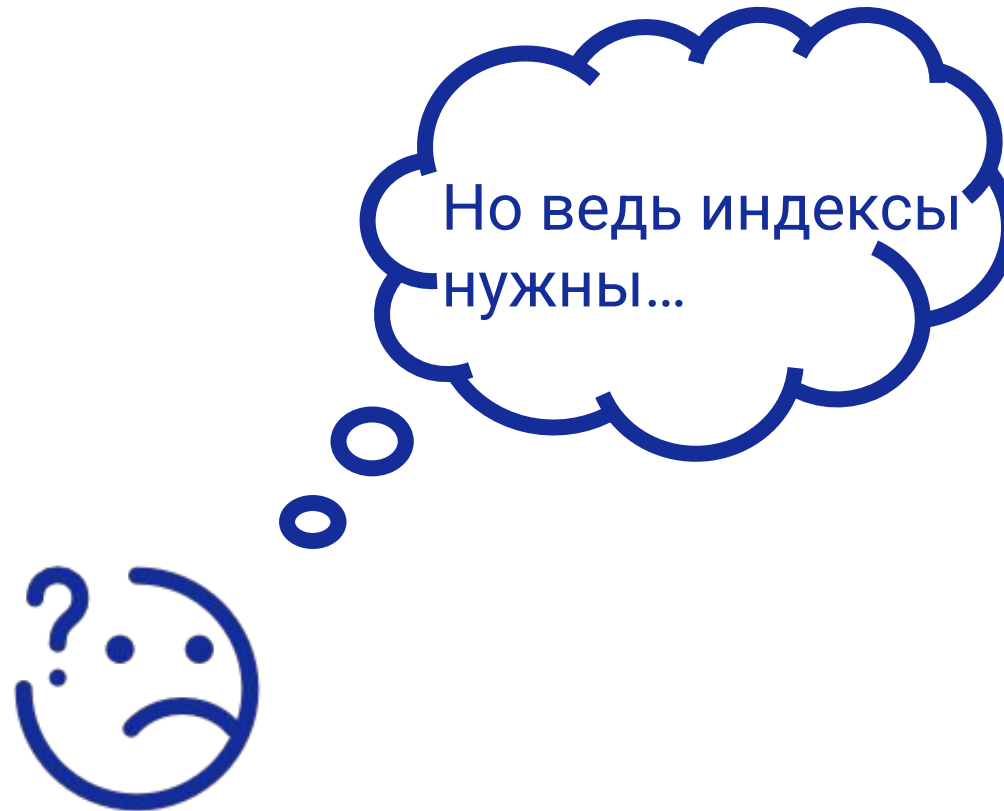
Disc Read #3

Повторить цикл от Page C -
«Прибытие»

1 шаг графа = 3 I/O

100 шагов поиска = 300 I/O

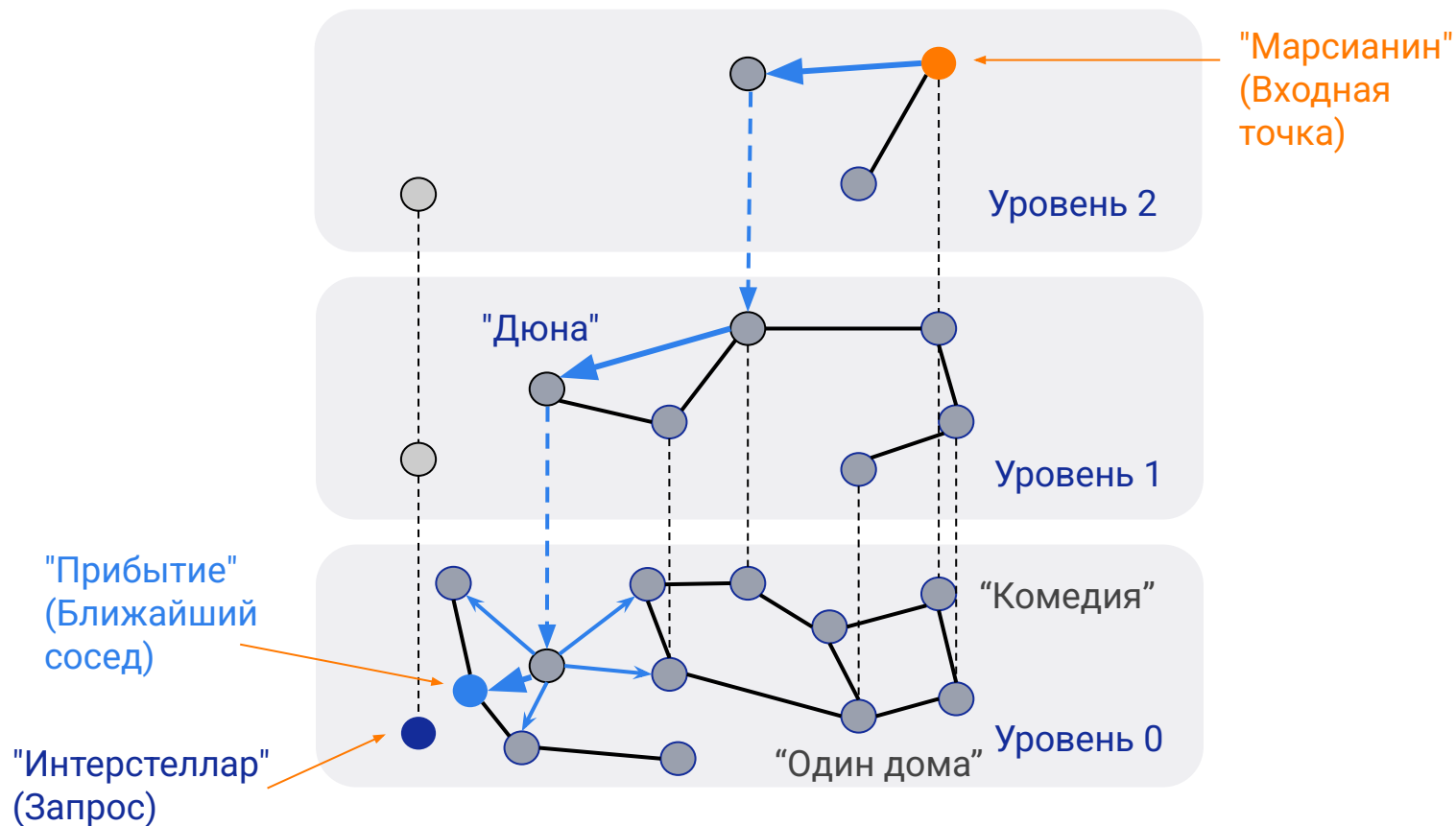
Хранение: вектор в мире 8-килобайтных страниц



HNSW: анатомия графа и погоня за указателями

HNSW

(иерархический маленький мир)

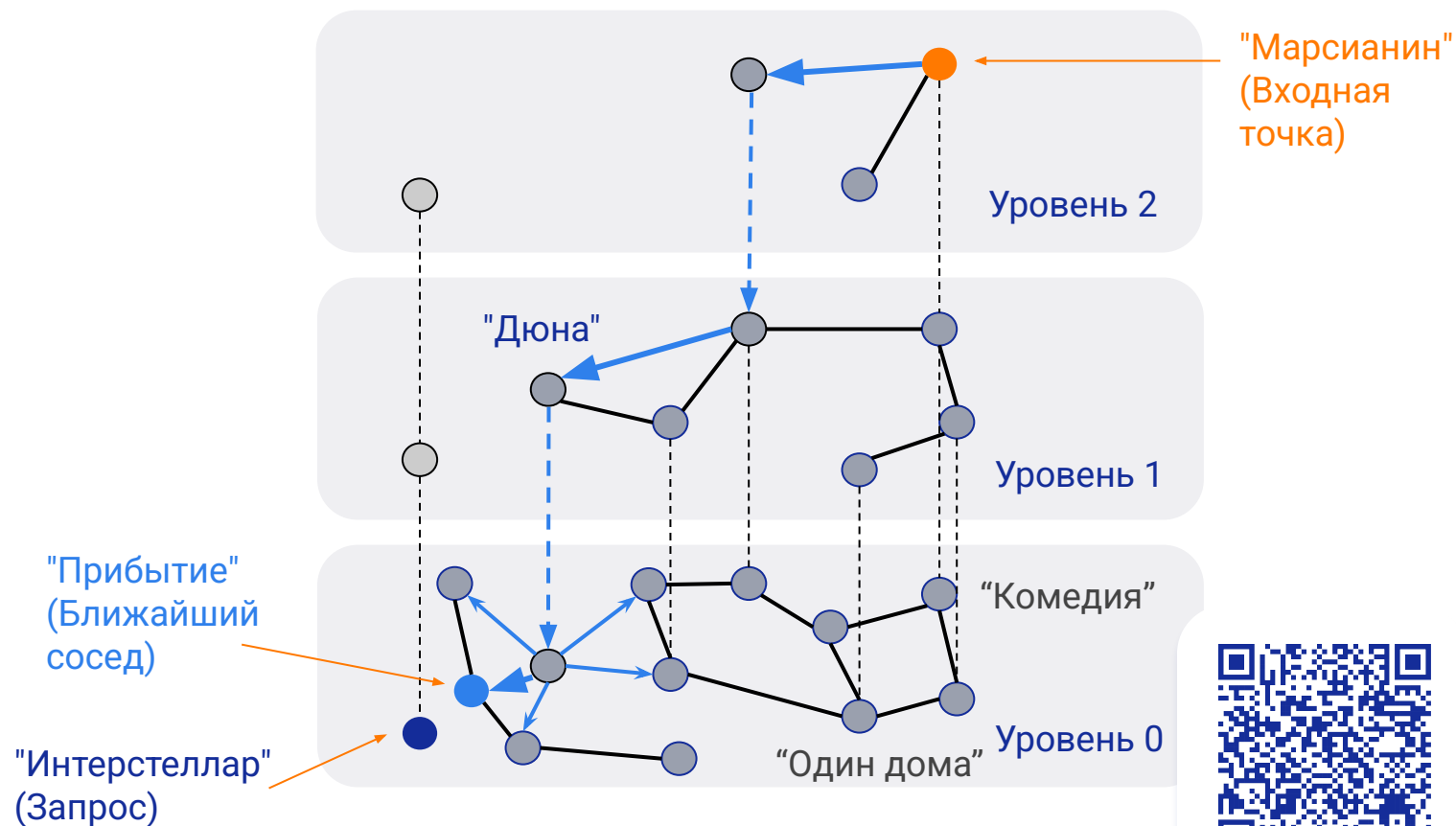


HNSW

(иерархический маленький мир)

Параметры индекса:

- M
- ef_construction



HNSW

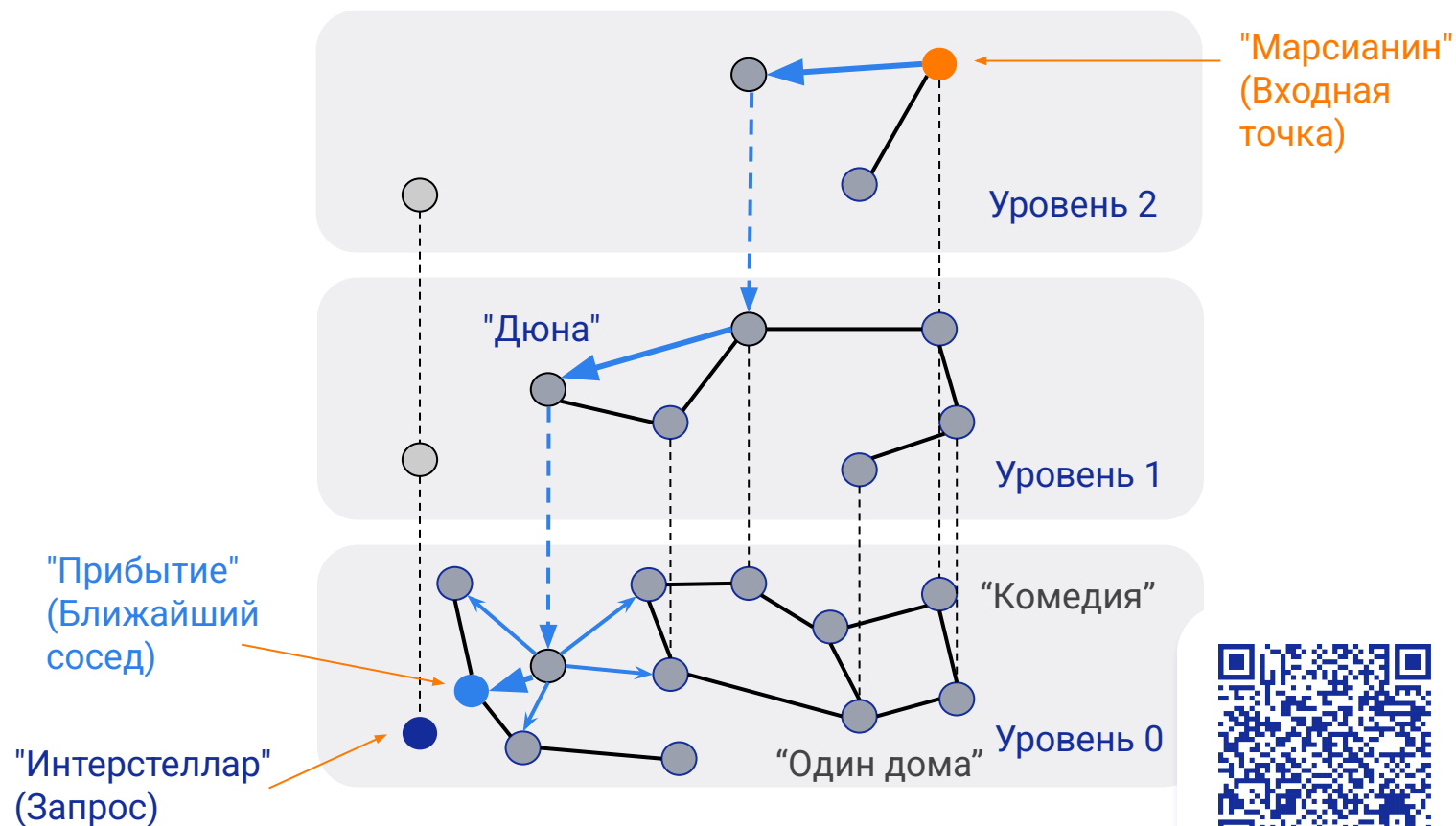
(иерархический маленький мир)

Параметры индекса:

- M
- ef_construction

Параметры поиска:

- ef_search



Структура HnswElementData

Распределение памяти индекса (5M векторов)

Векторы (данные)



30.0 ГБ



Структура HnswElementData

Распределение памяти индекса (5M векторов)

Векторы (данные)



30.0 ГБ

Ребра уровня 0

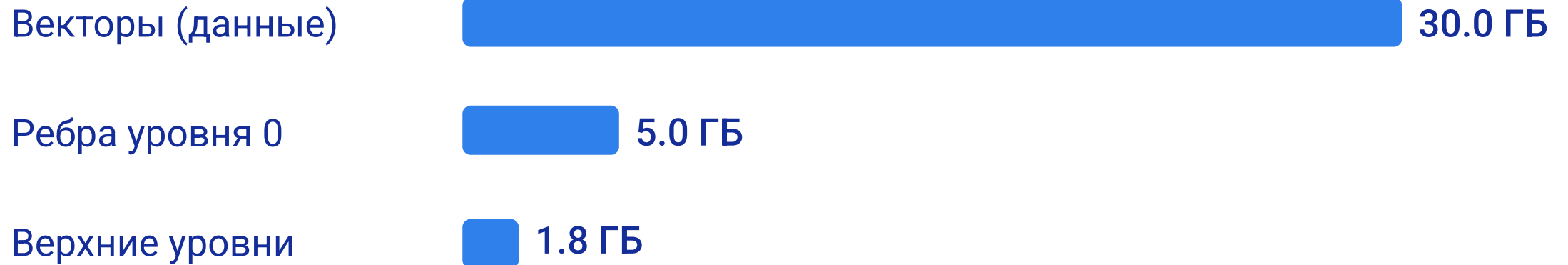


5.0 ГБ



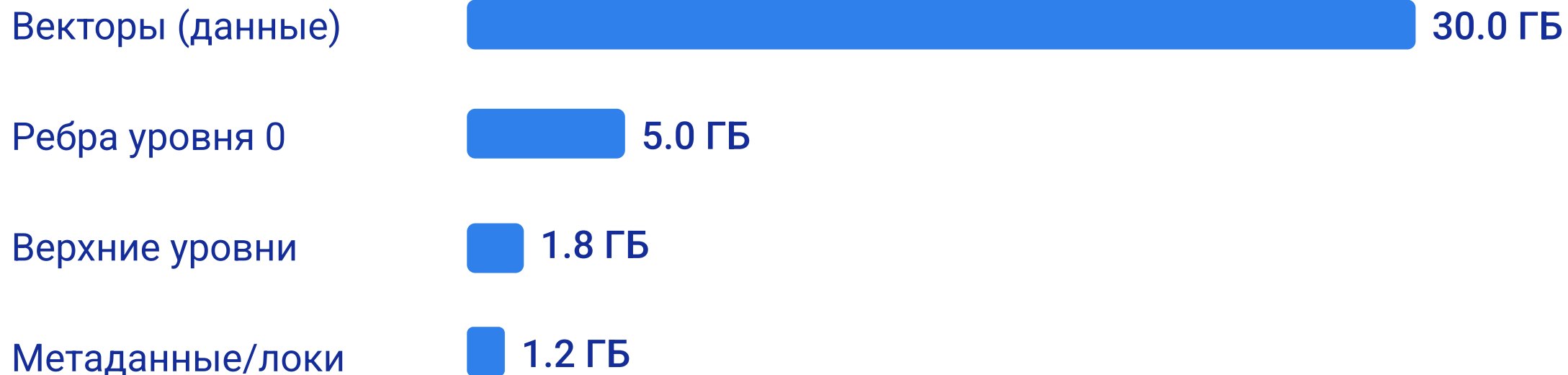
Структура HnswElementData

Распределение памяти индекса (5M векторов)



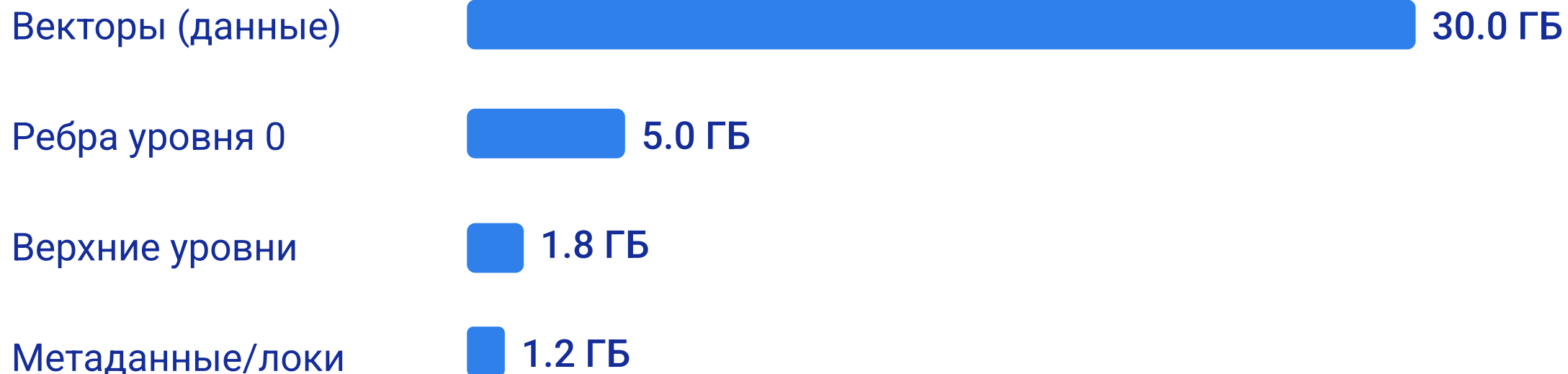
Структура HnswElementData

Распределение памяти индекса (5M векторов)



Структура HnswElementData

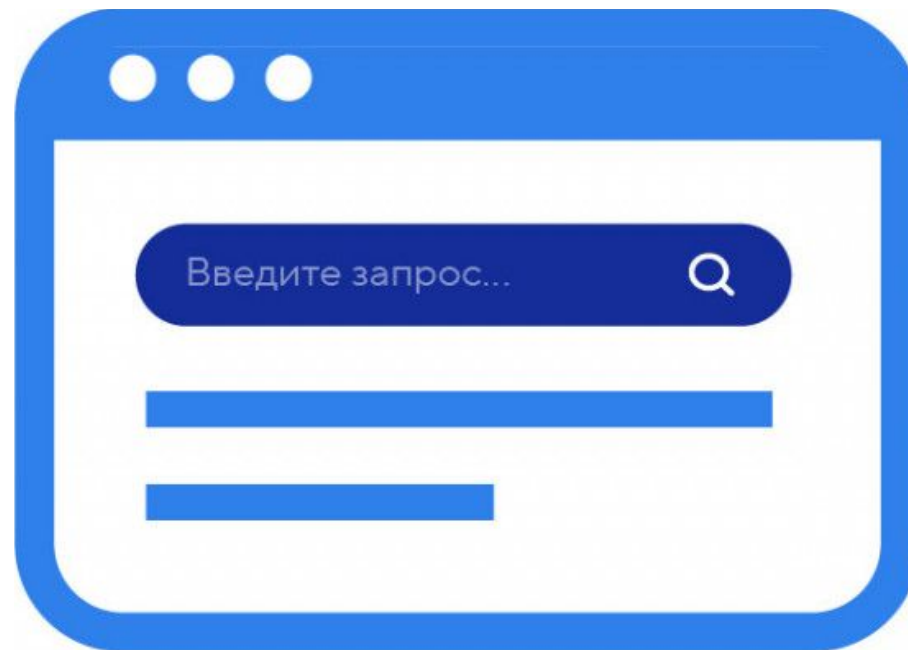
Распределение памяти индекса (5М векторов)



30 ГБ чистых данных → 38 ГБ индекс:
8 ГБ уходит на рёбра, метаданные и локи Access Method API

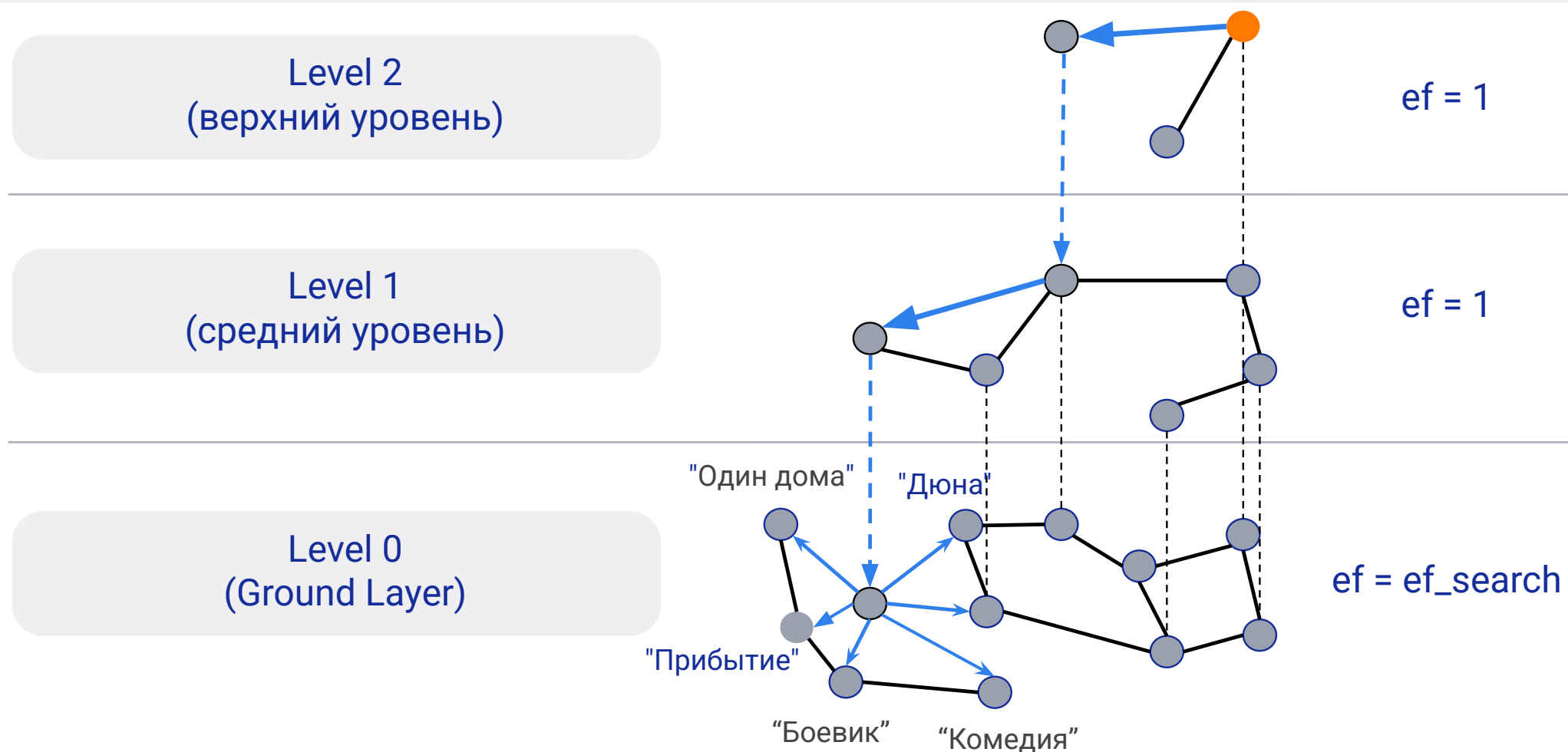


Ради чего все это?



HnswSearchLayer:

Цикл, который убивает кэш

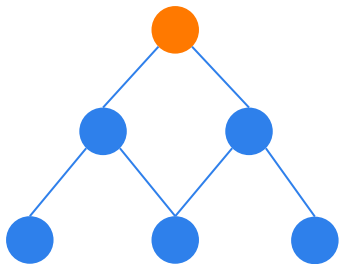


HnswSearchLayer: Цикл, который убивает кэш

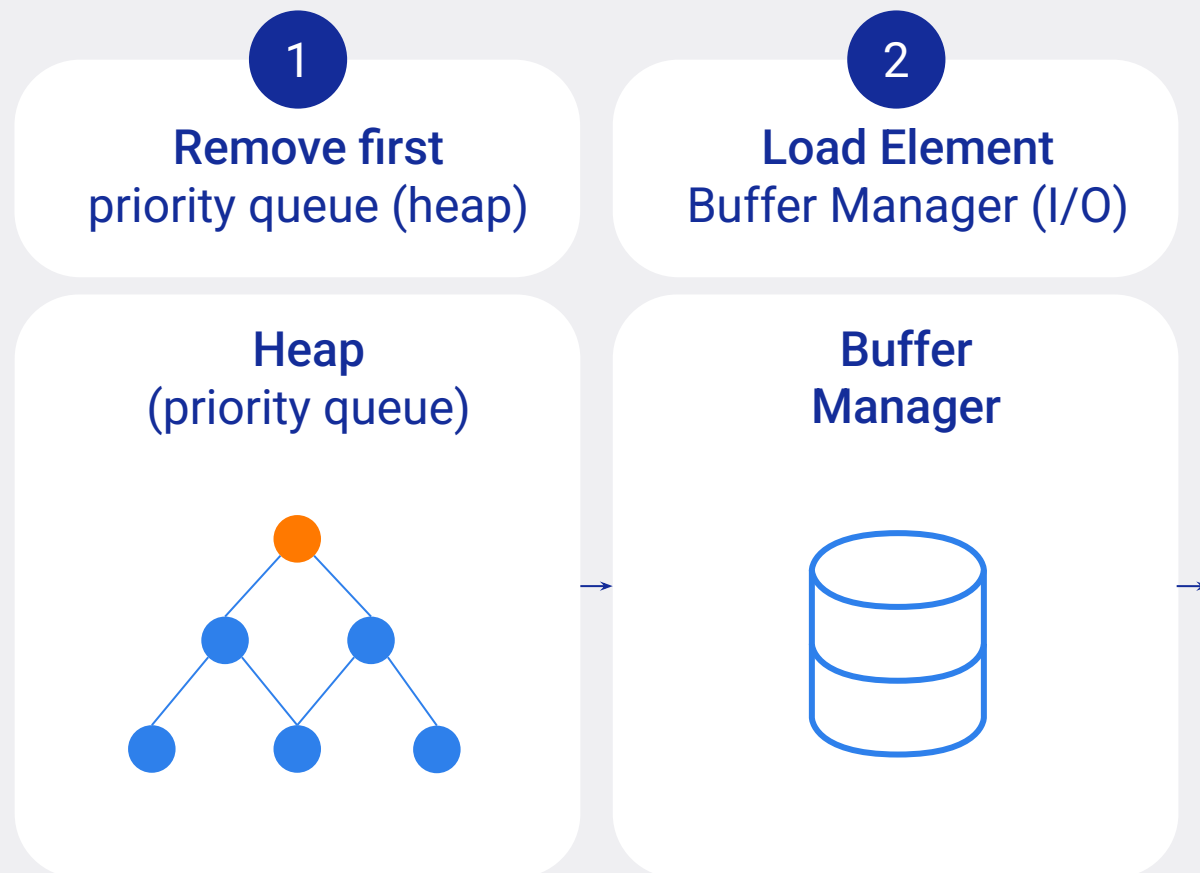
1

Remove first
priority queue (heap)

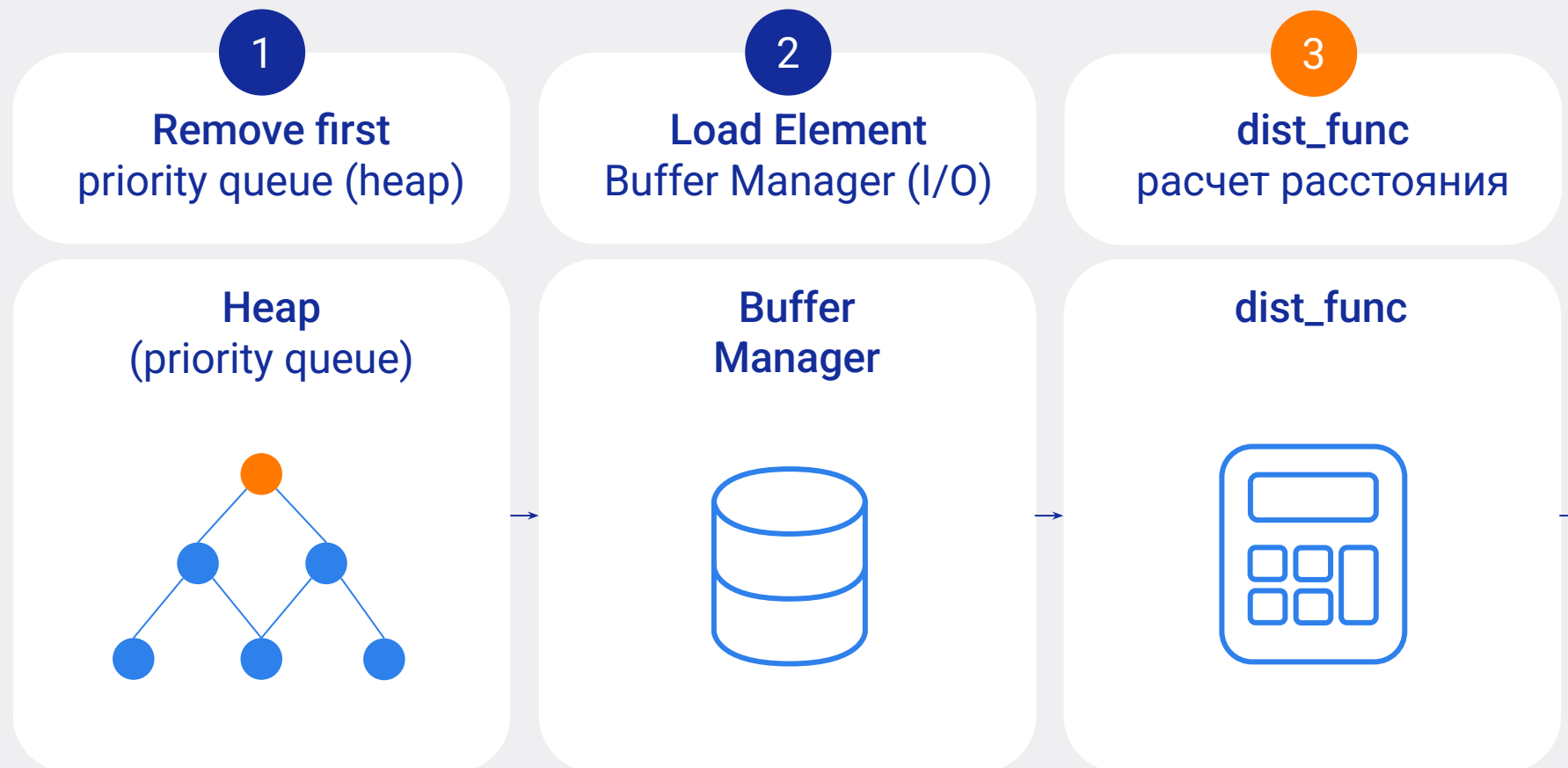
Heap
(priority queue)



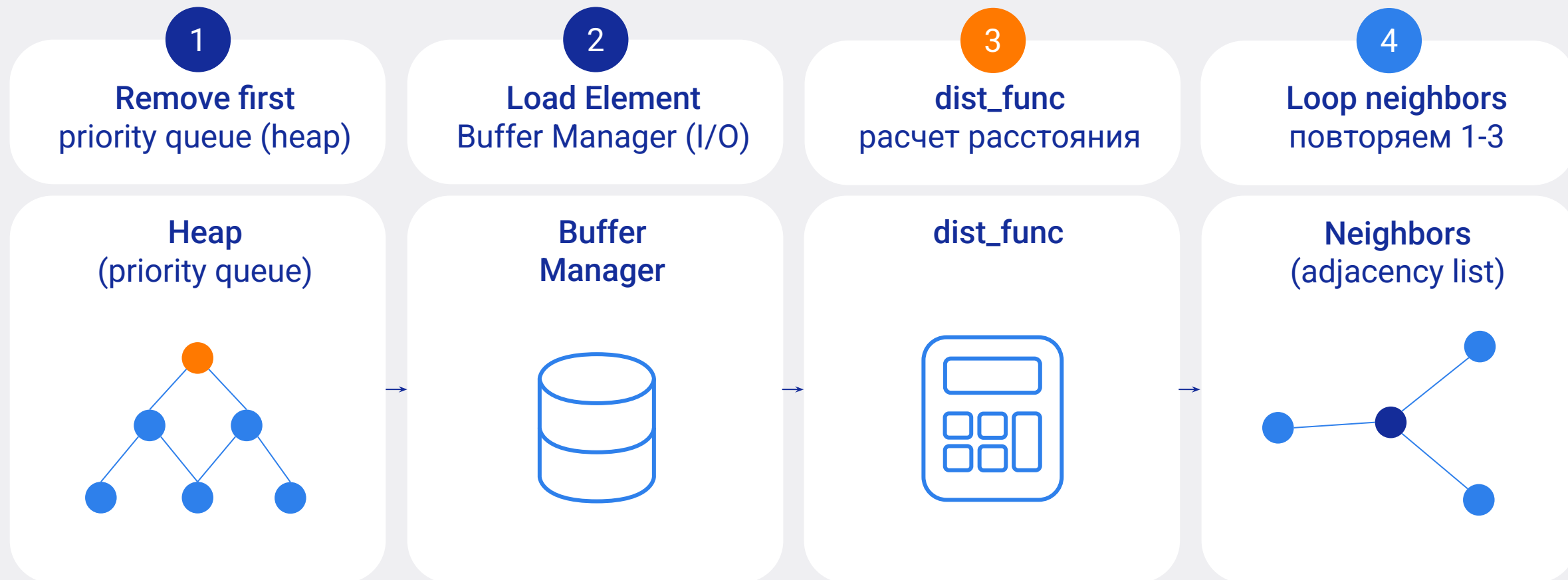
HnswSearchLayer: Цикл, который убивает кэш



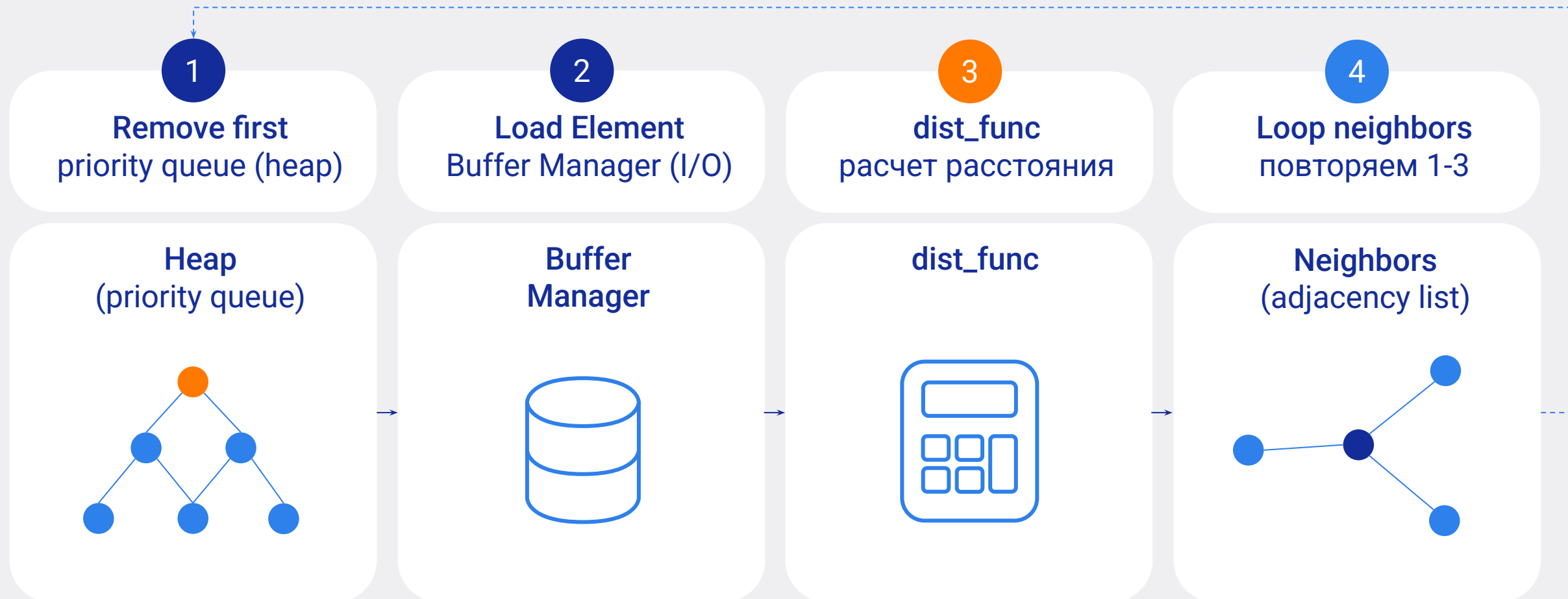
HnswSearchLayer: Цикл, который убивает кэш



HnswSearchLayer: Цикл, который убивает кэш

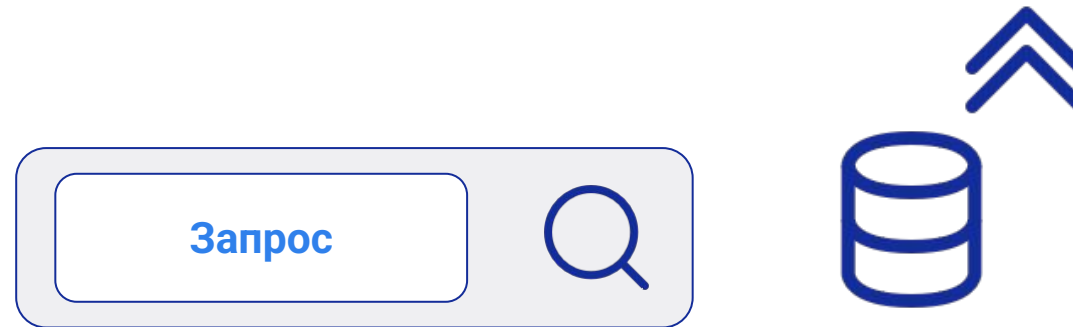


HnswSearchLayer: Цикл, который убивает кэш

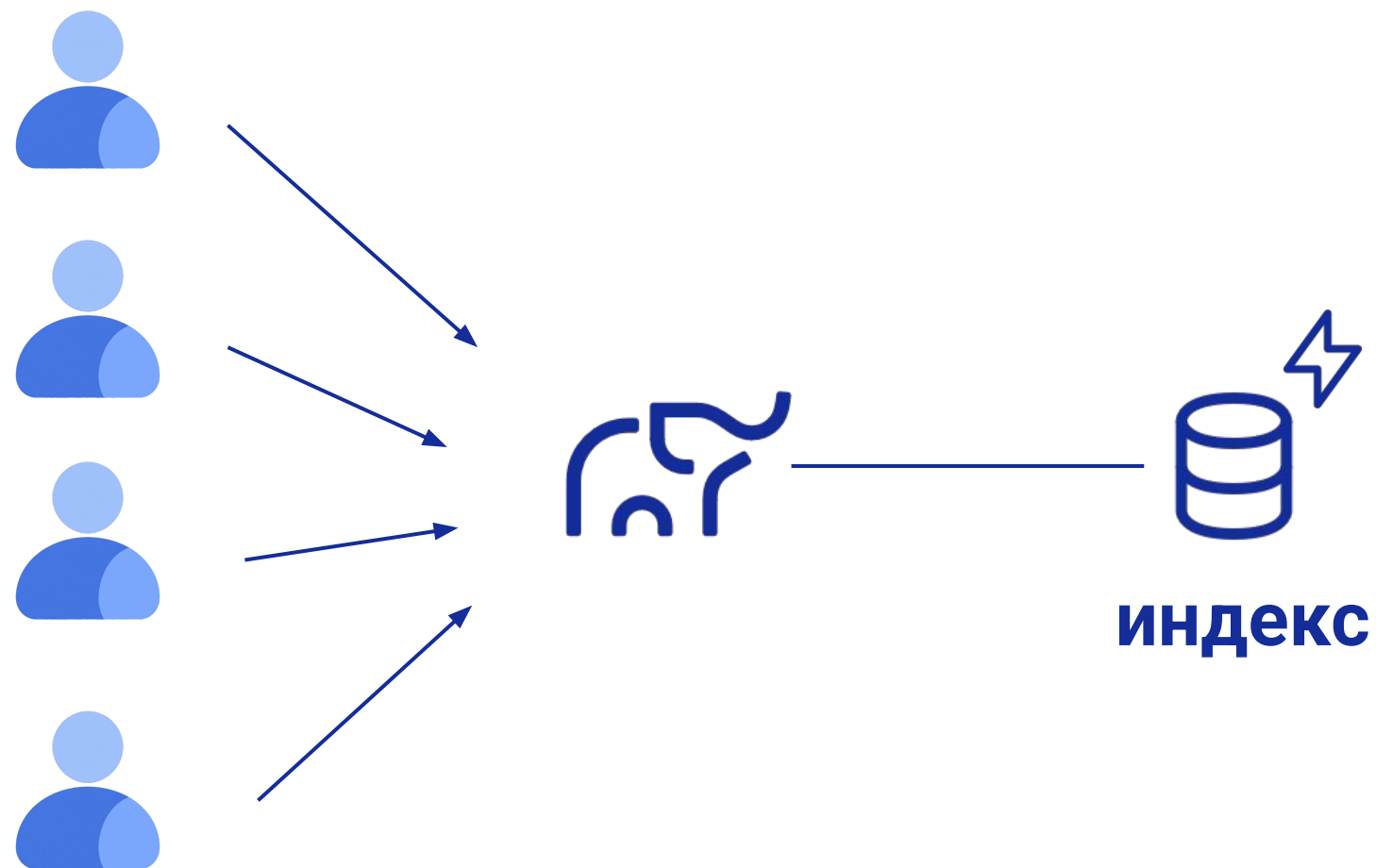


Pop closest ----->

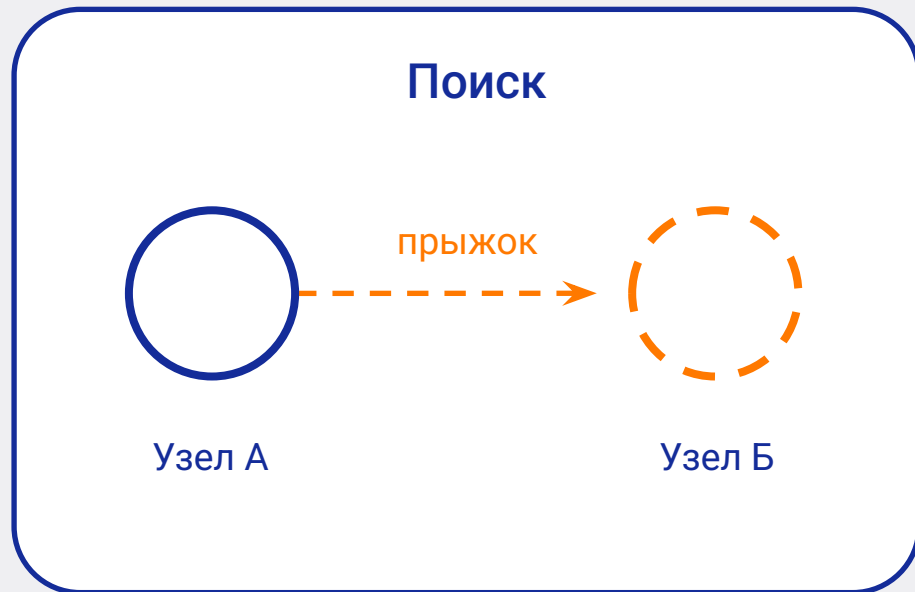
Это был только 1 запрос...



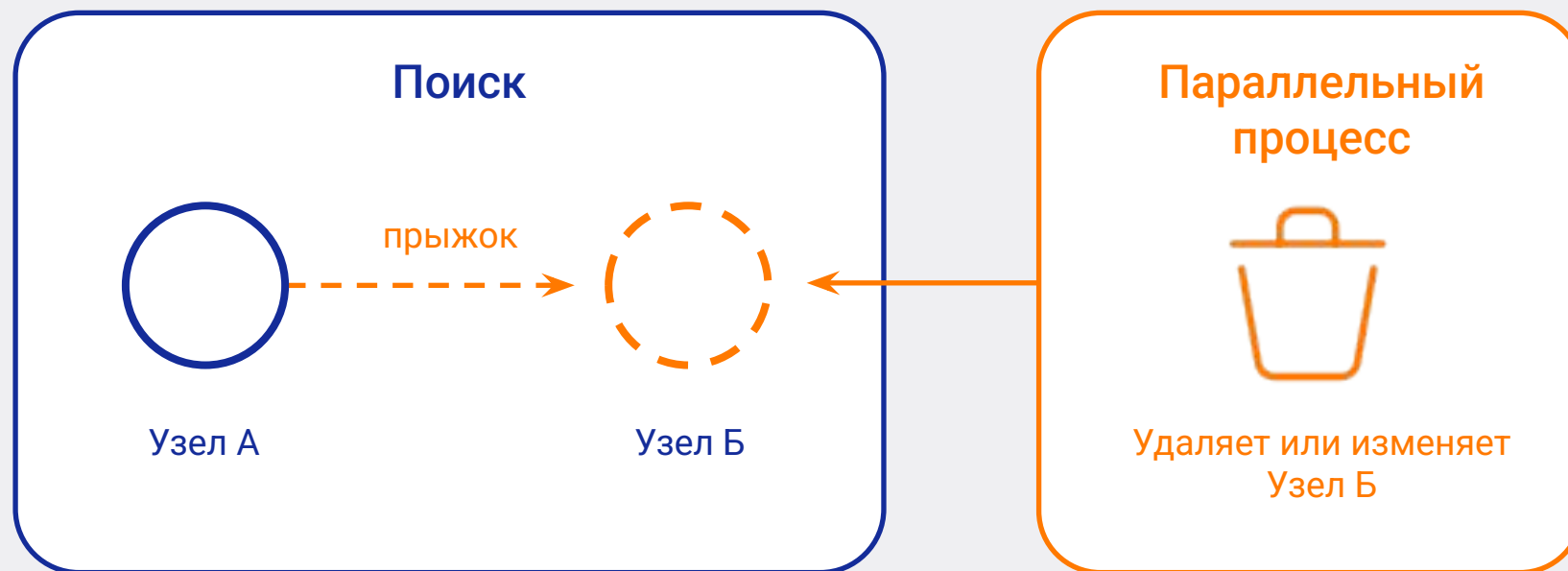
А если запросов много?



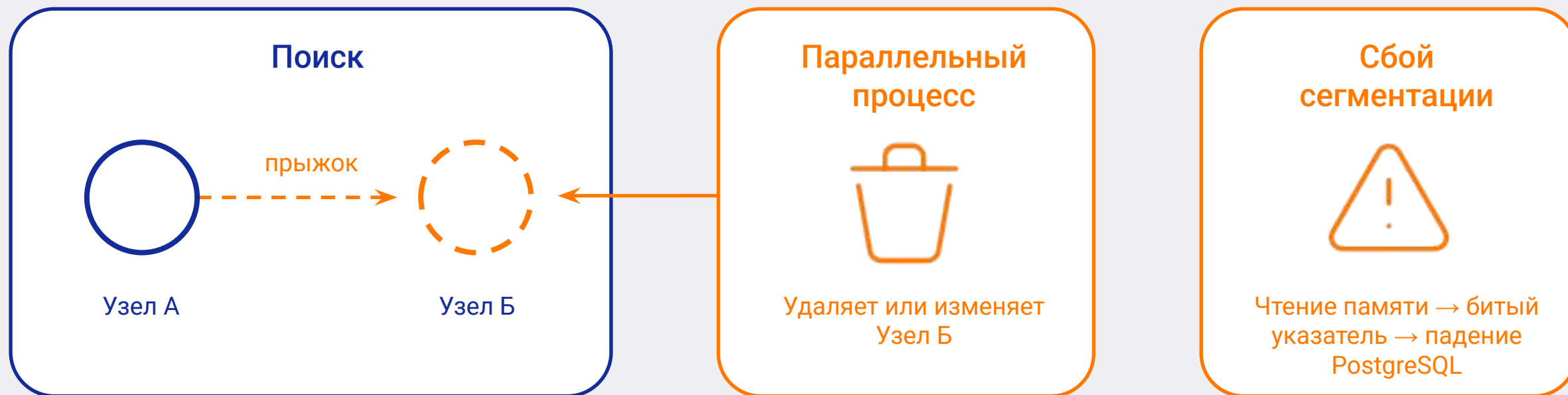
А если запросов много?



А если запросов много?



А если запросов много?



А если запросов много?

1

MVCC снимки

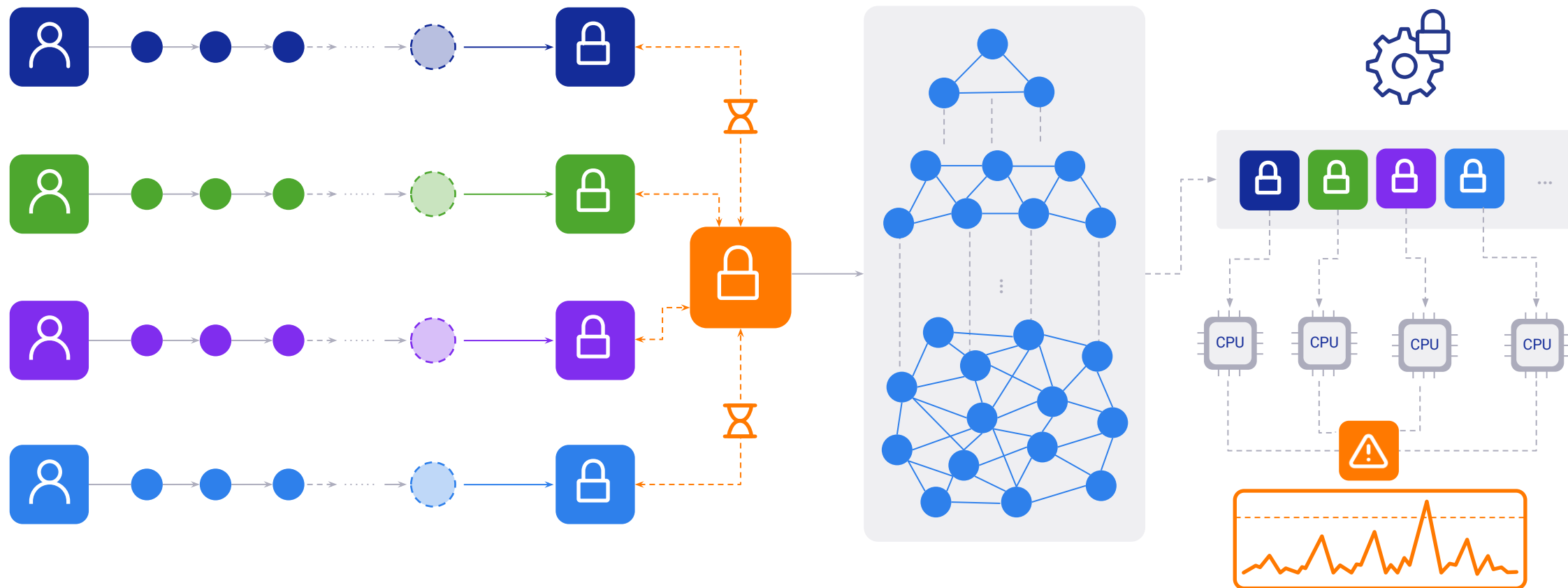


MVCC — разрешены
не-MVCC — запрещены

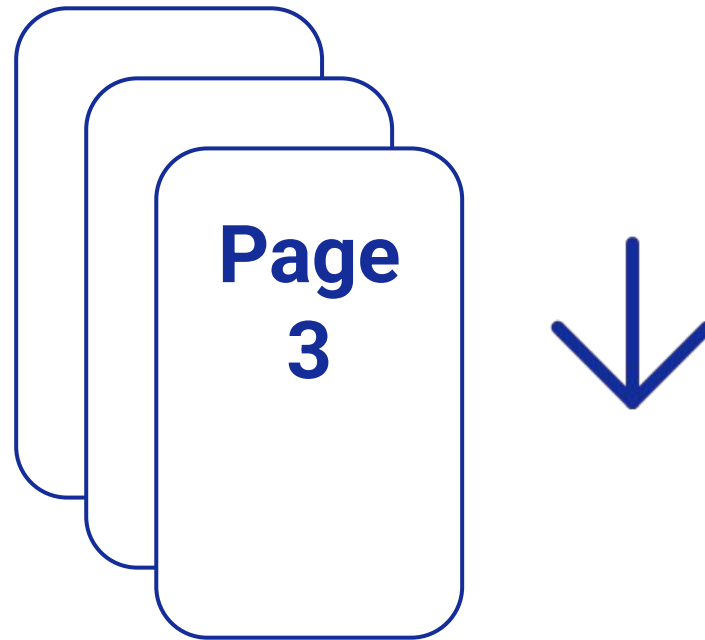


pgvector не держит
пины буферов

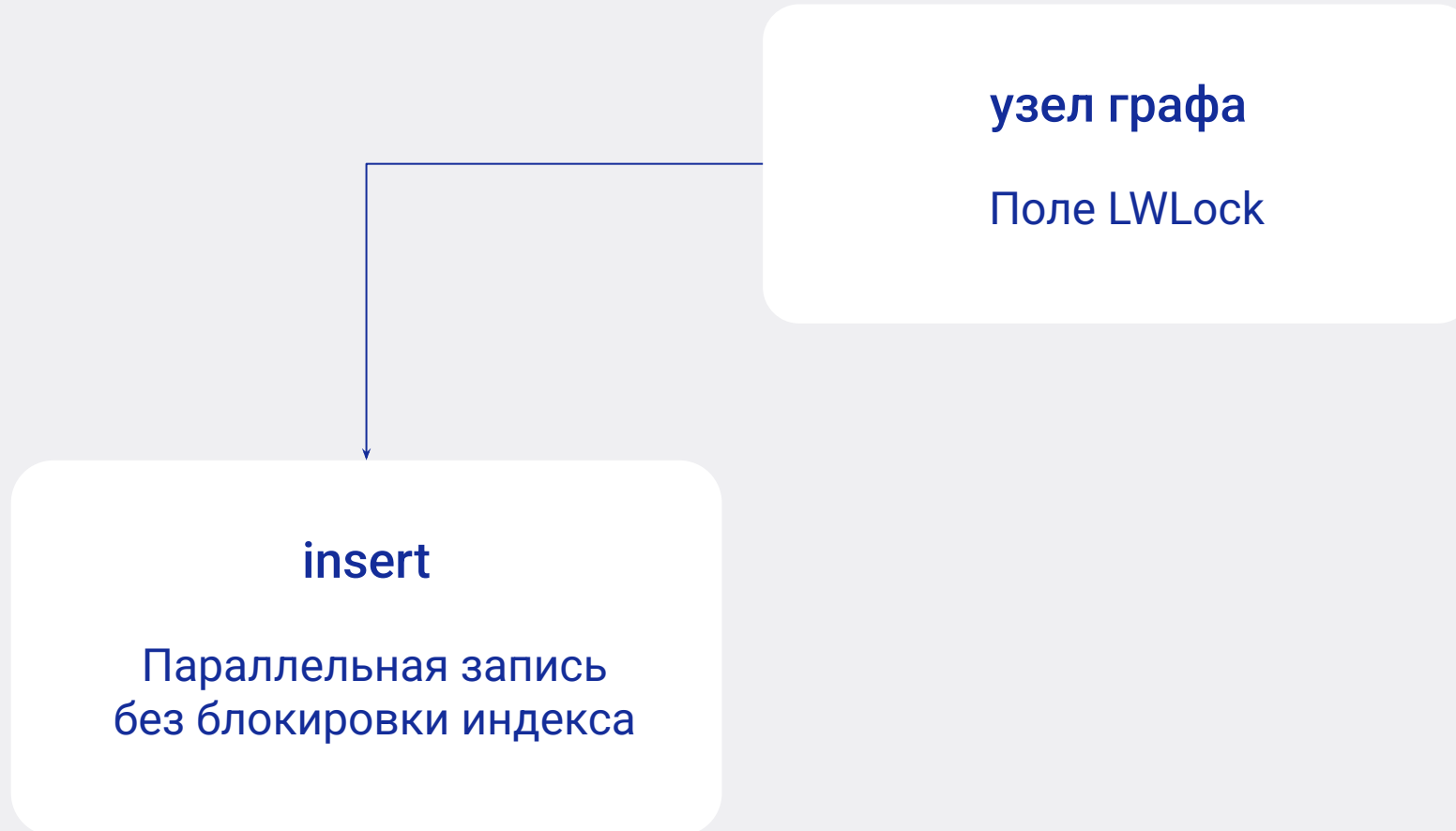
А если запросов много?



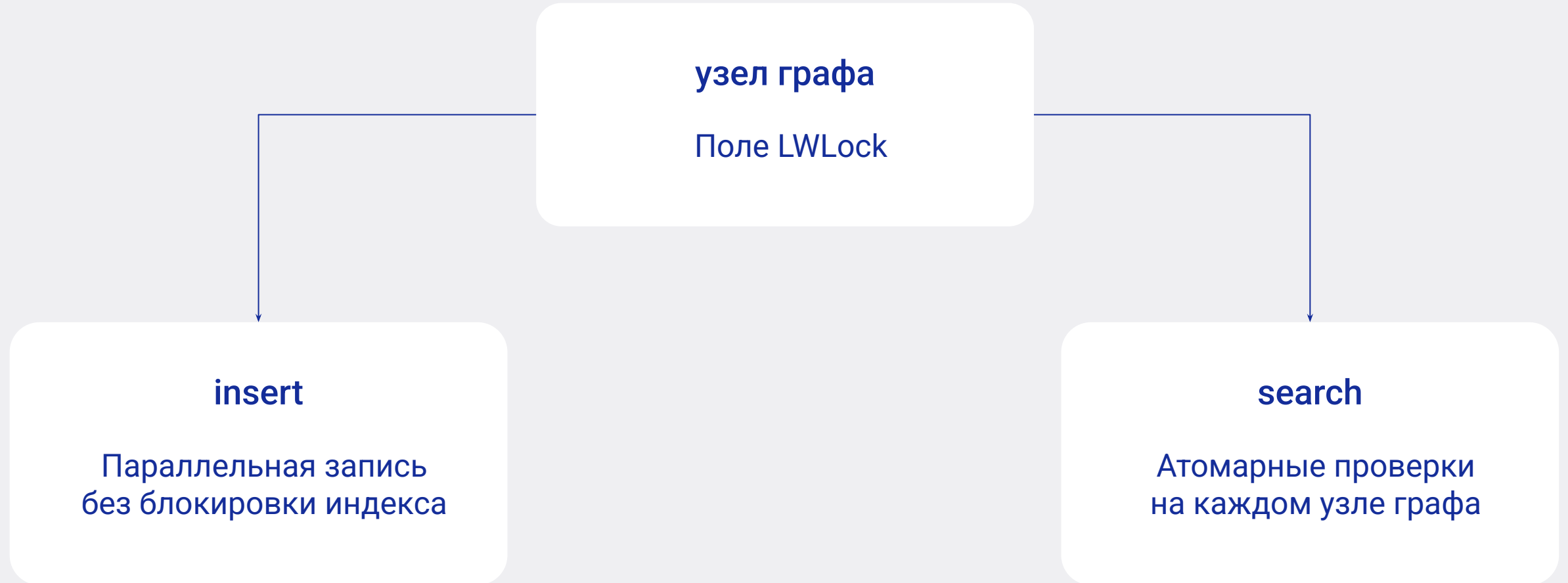
Структура HnswElementData



Структура HnswElementData



Структура HnswElementData



Структура HnswElementData



А если запросов много?

1

MVCC снимки



MVCC — разрешены
не-MVCC — запрещены



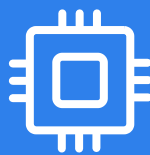
pgvector не держит
пины буферов

2

ShareLock



Lock страниц индекса
в режиме SHARE



Каждый шаг поиска
= системный вызов

А если запросов много?

	visited_hash / Pairing Heap
Где живёт	Memory context запроса
Между запросами	Уничтожается, следующий запрос строит с нуля
Эффект прогрева кэша	Не работает — прогрева не существует в принципе

А если запросов много?

	visited_hash / Pairing Heap
Где живёт	Memory context запроса
Между запросами	Уничтожается, следующий запрос строит с нуля
Эффект прогрева кэша	Не работает — прогрева не существует в принципе

```
// Проверка visited – вызывается на  
КАЖДОЙ итерации
```

```
if visited(neighbor) → skip  
else → mark visited, add to unvisited
```

А если запросов много?

1

MVCC снимки



MVCC — разрешены
не-MVCC — запрещены



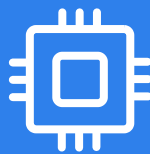
pgvector не держит
пины буферов

2

ShareLock



Lock страниц индекса
в режиме SHARE



Каждый поиск
= системный вызов

3

Сброс контекста



На каждом
возобновлении



Хэш-таблица узлов
сбрасывается

А если запросов много?

1

MVCC снимки



MVCC — разрешены
не-MVCC — запрещены

2

ShareLock



Lock страниц индекса
в режиме SHARE

3

Сброс контекста



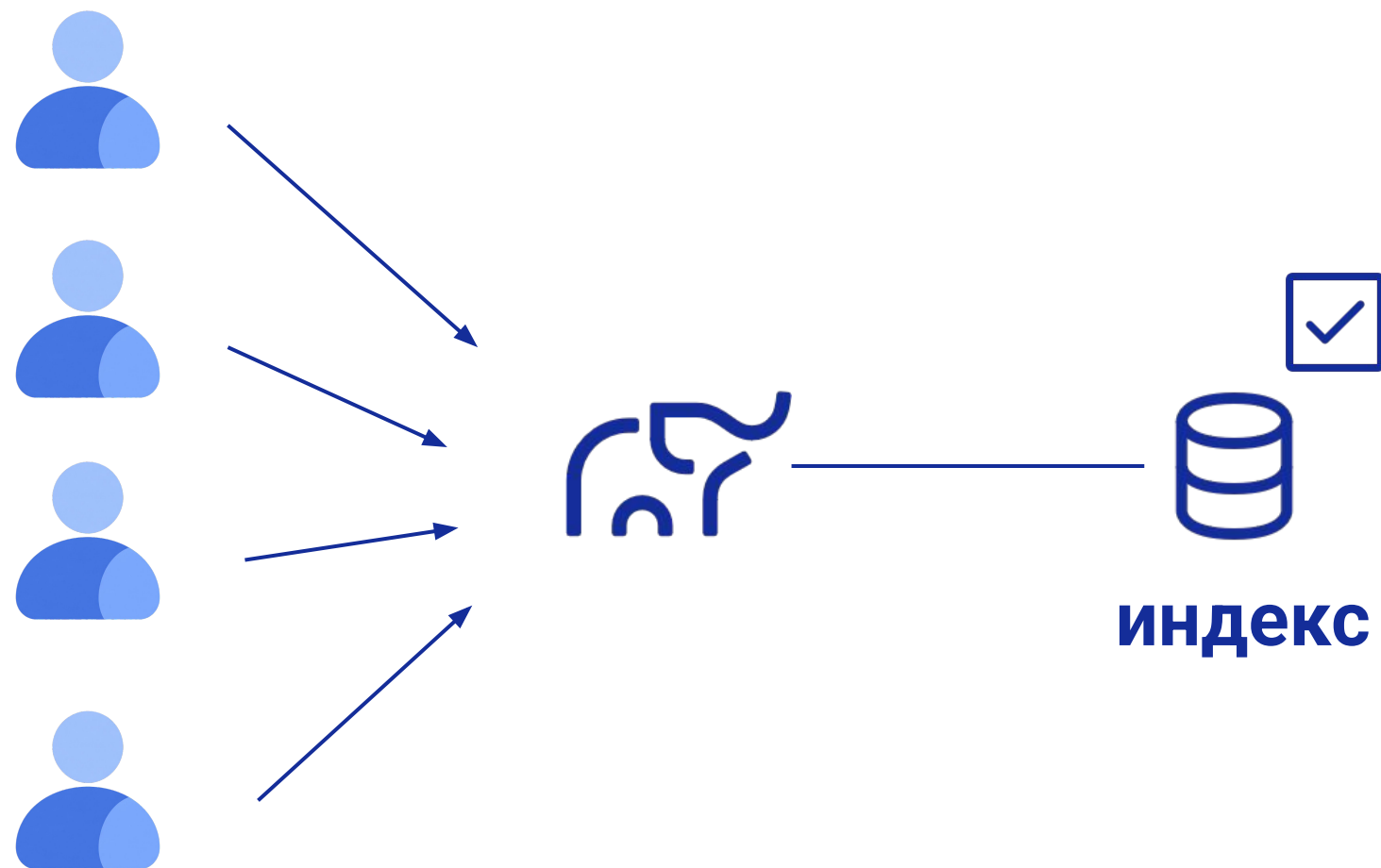
На каждом
возобновлении

Поиск всегда видит целостные данные,
даже при параллельных изменениях

Защита от конкурентных изменений
страниц

Защита от утечек и переполнения
памяти

А если запросов много?



И все же, что с графами?

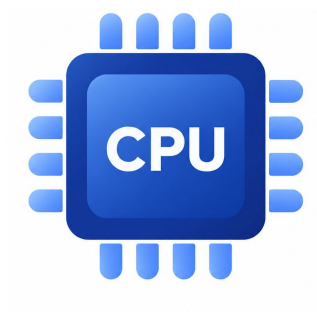
Три жита слона



Три кита слона



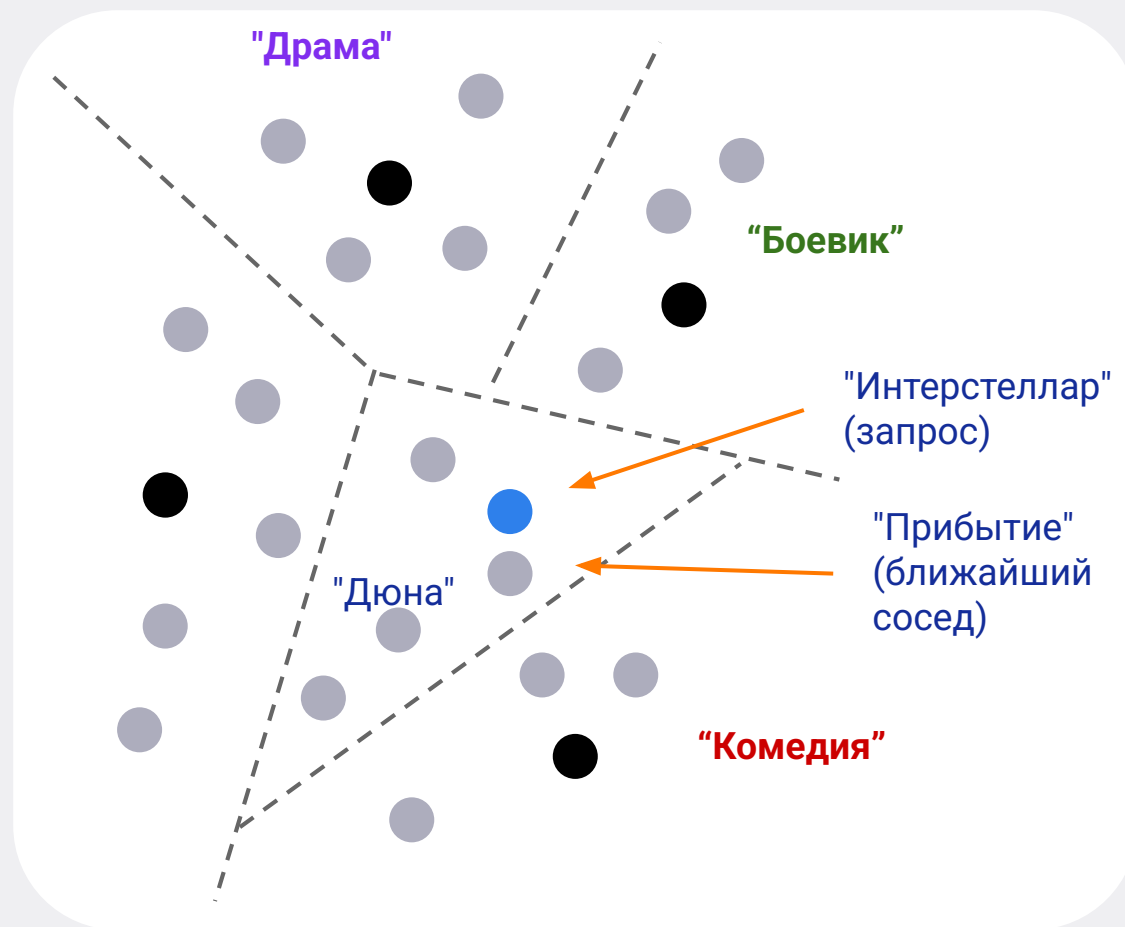
Три кита слона



IVFFlat: стриминг против прыжков

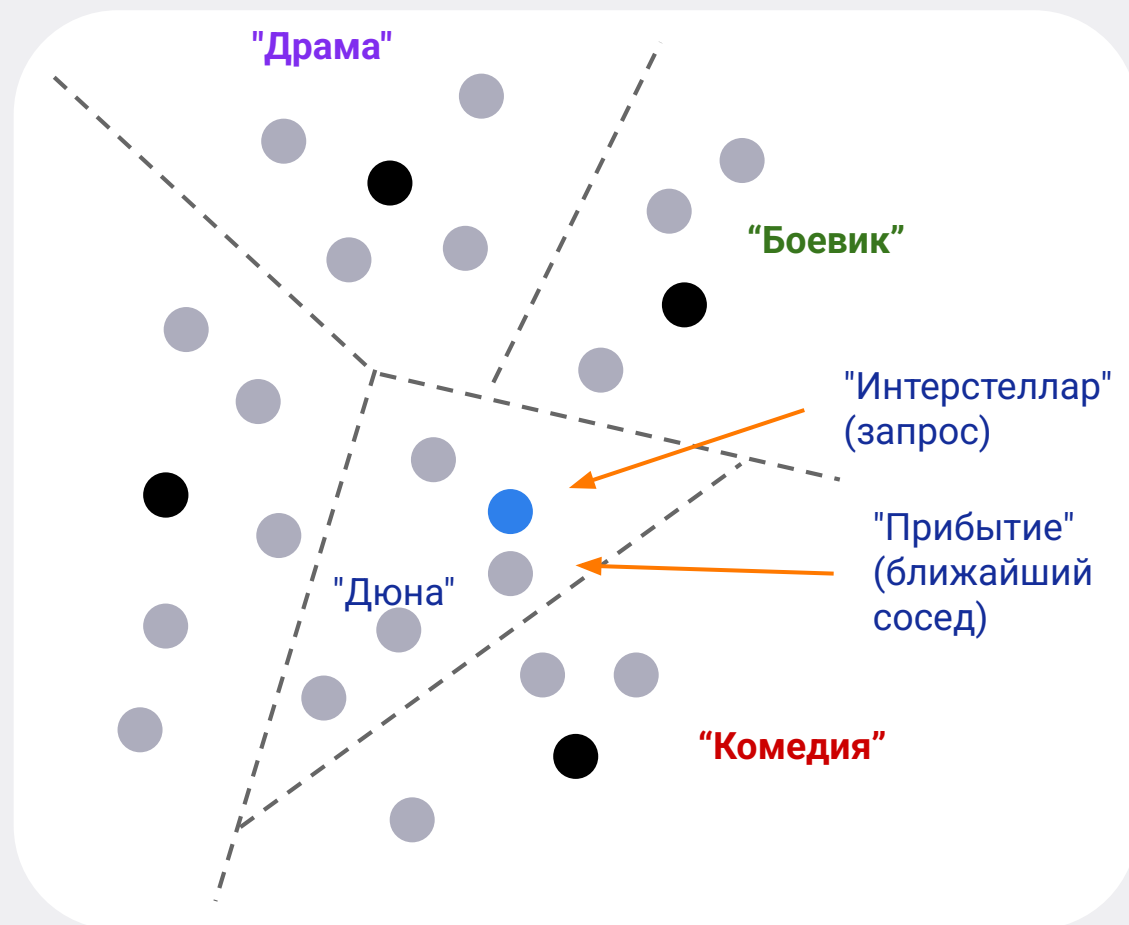
IVFFlat

(Инвертированный файл с плоским сжатием)



IVFFlat

(Инвертированный файл с плоским сжатием)



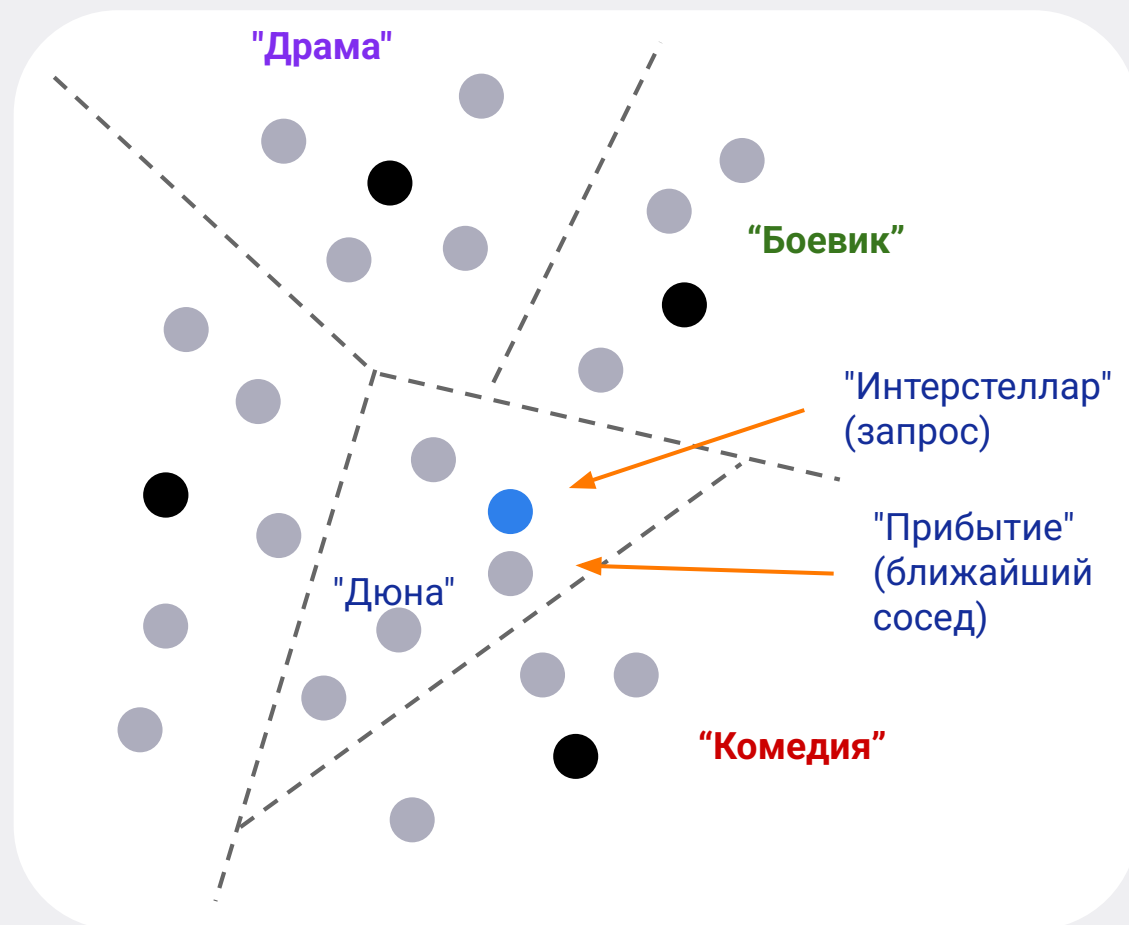
Параметры индекса:

- `lists` = число кластеров
больше → мельче кластеры
меньше → крупнее кластеры



IVFFlat

(Инвертированный файл с плоским сжатием)



Параметры индекса:

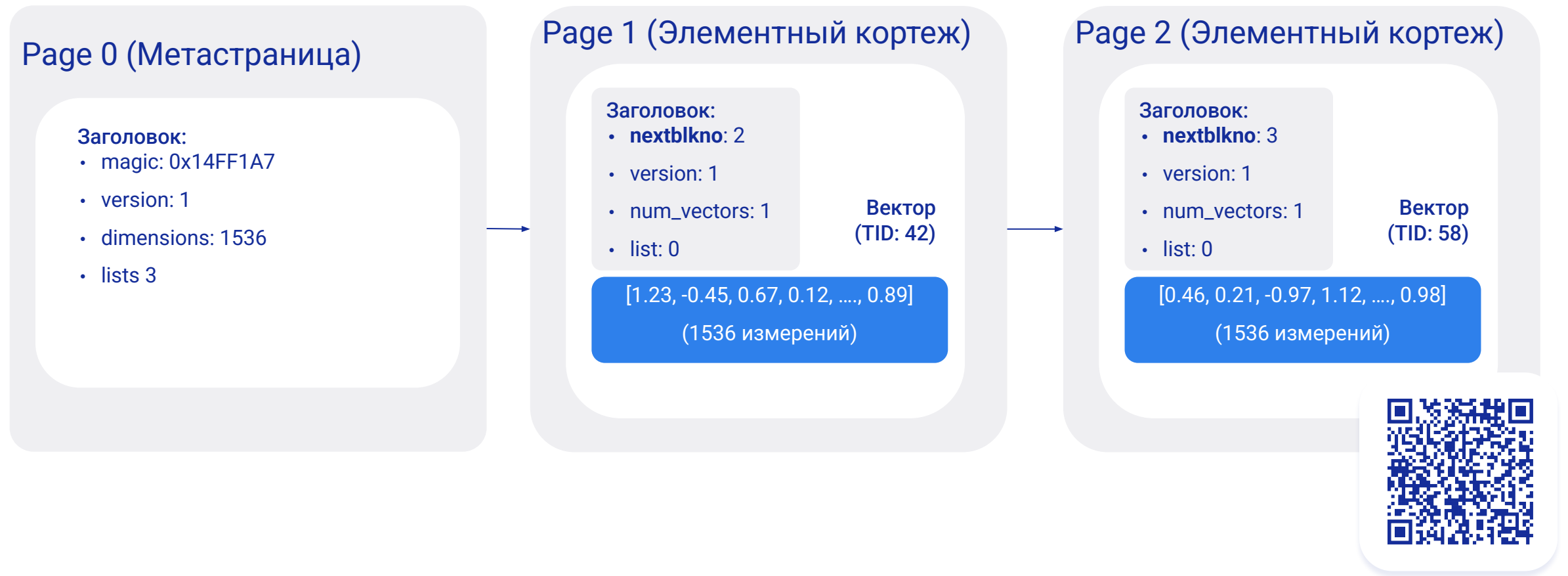
- `lists` = число кластеров
больше → мельче кластеры
меньше → крупнее кластеры

Параметры поиска:

- `probes` = сколько кластеров проверять на запросе
 $\uparrow \text{probes} \rightarrow \uparrow \text{recall}, \uparrow \text{время}$



Структура IVFFlat



Структура IVFFlat

Страницы идут последовательно →

Page 0 (Метастраница)

Заголовок:

- magic: 0x14FF1A7
- version: 1
- dimensions: 1536
- lists 3

Page 1 (Элементный кортеж)

Заголовок:

- **nextblkno**: 2
- version: 1
- num_vectors: 1
- list: 0

Вектор
(TID: 42)

[1.23, -0.45, 0.67, 0.12, ..., 0.89]
(1536 измерений)

Page 2 (Элементный кортеж)

Заголовок:

- **nextblkno**: 3
- version: 1
- num_vectors: 1
- list: 0

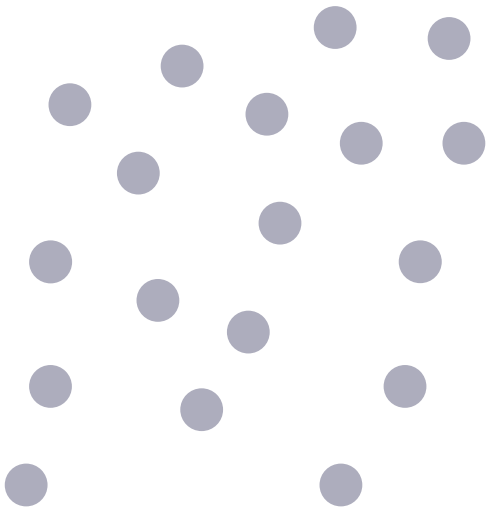
Вектор
(TID: 58)

[0.46, 0.21, -0.97, 1.12, ..., 0.98]
(1536 измерений)



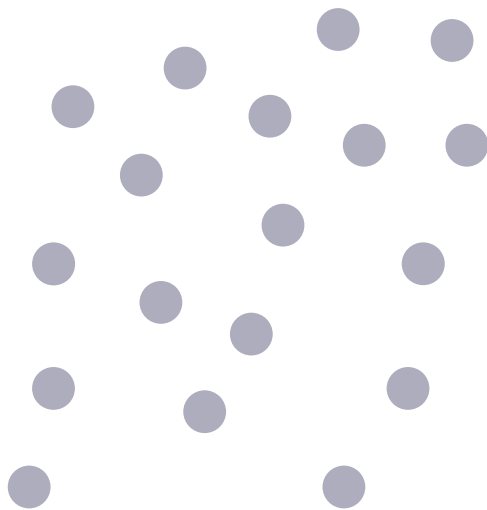
IVFFlat: когда простота — это SeqScan

Исходные
вектора

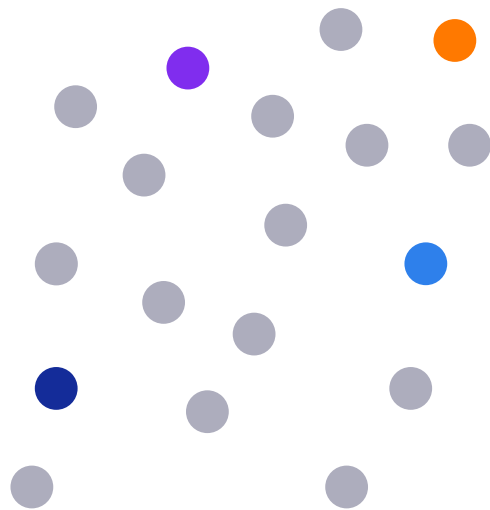


IVFFlat: когда простота — это SeqScan

Исходные
вектора

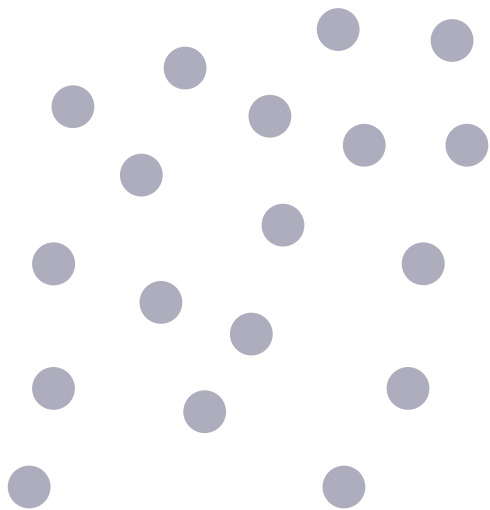


Выбор k
центроидов

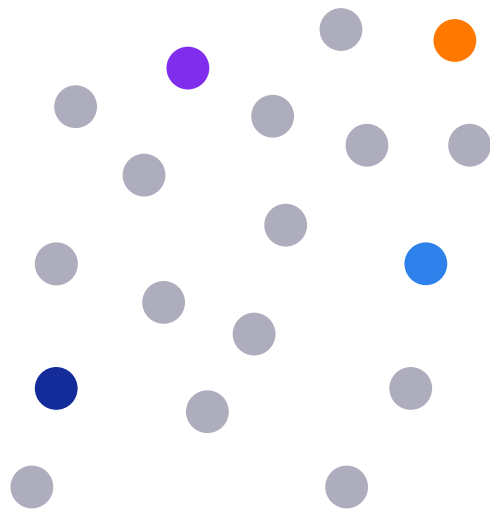


IVFFlat: когда простота — это SeqScan

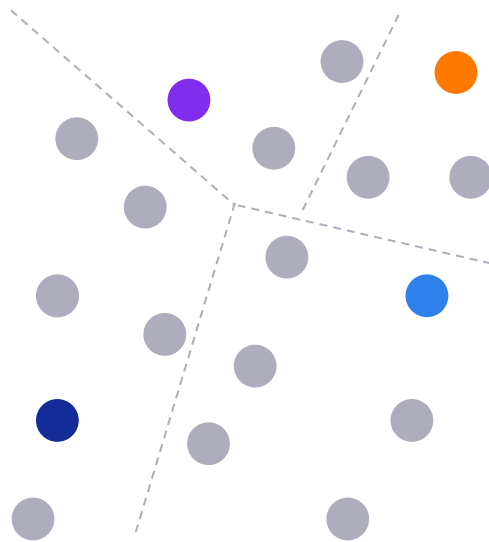
Исходные
вектора



Выбор k
центроидов

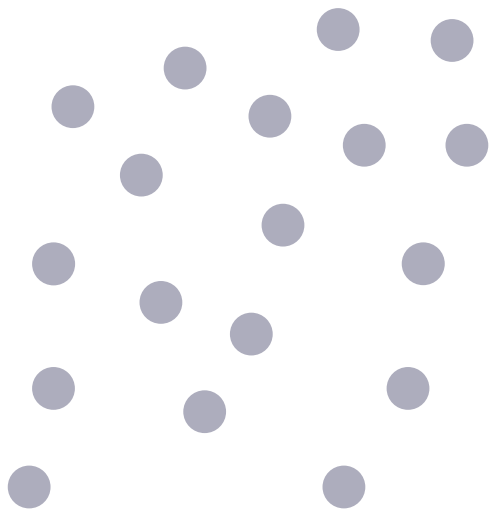


Назначаем векторы
к центроидам

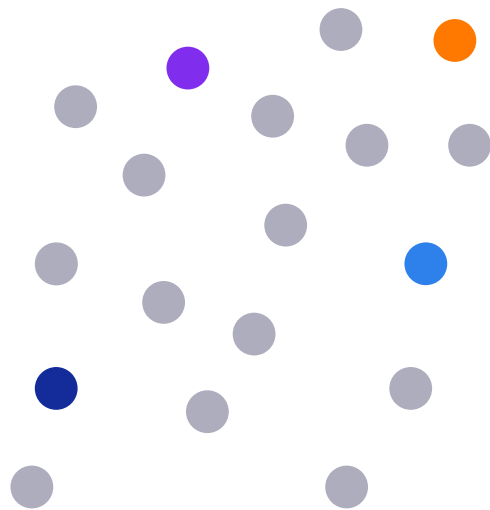


IVFFlat: когда простота — это SeqScan

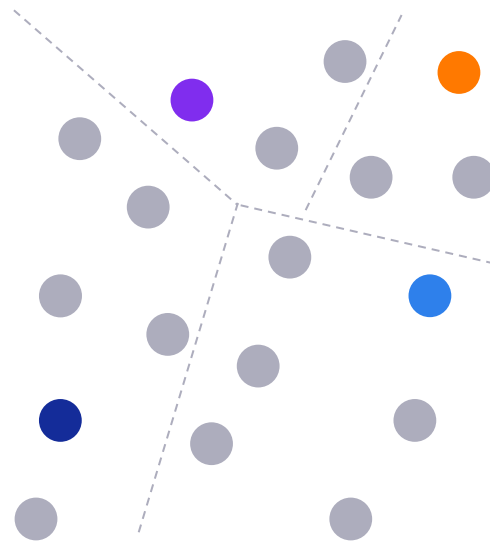
Исходные
вектора



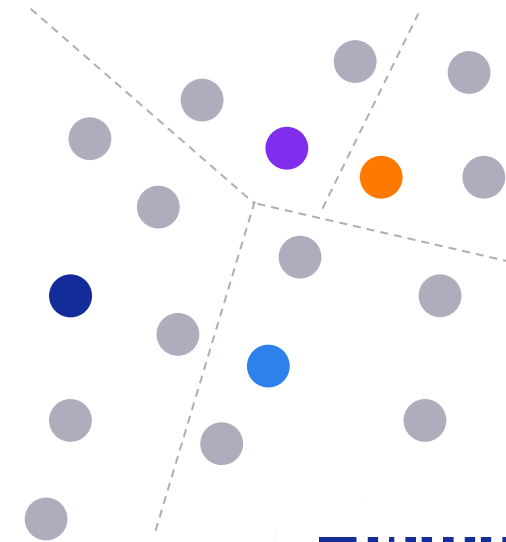
Выбор k
центроидов



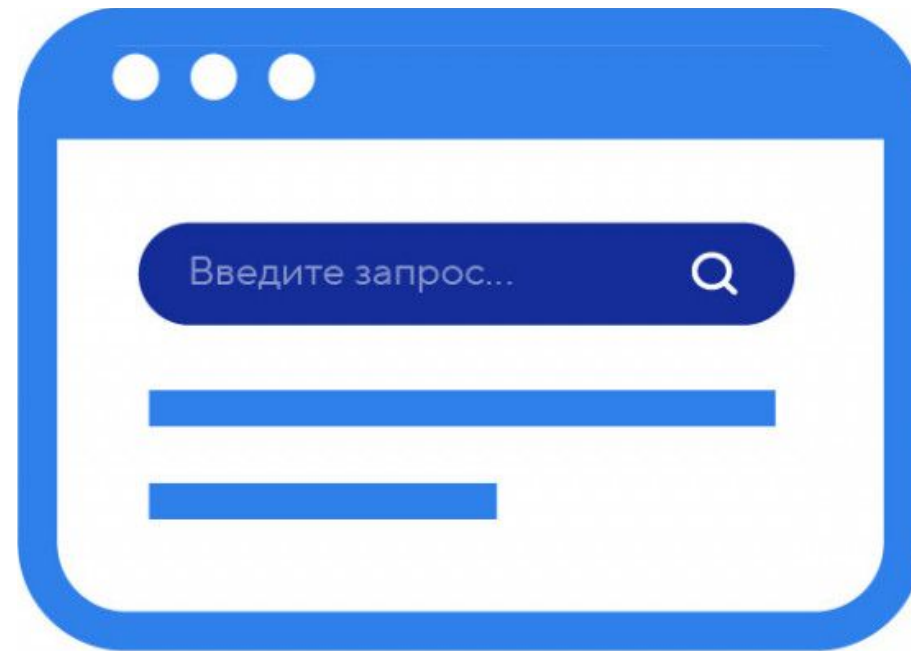
Назначаем векторы
к центроидам



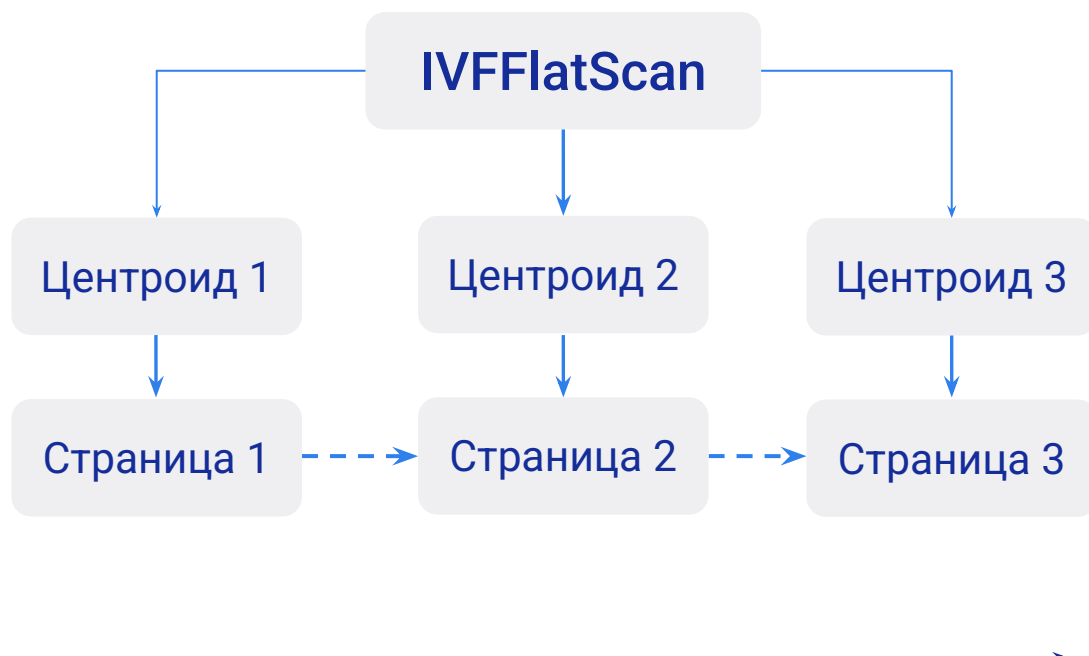
Перерасчет
центров



А поиск?



IVFFlat: когда простота — это SeqScan



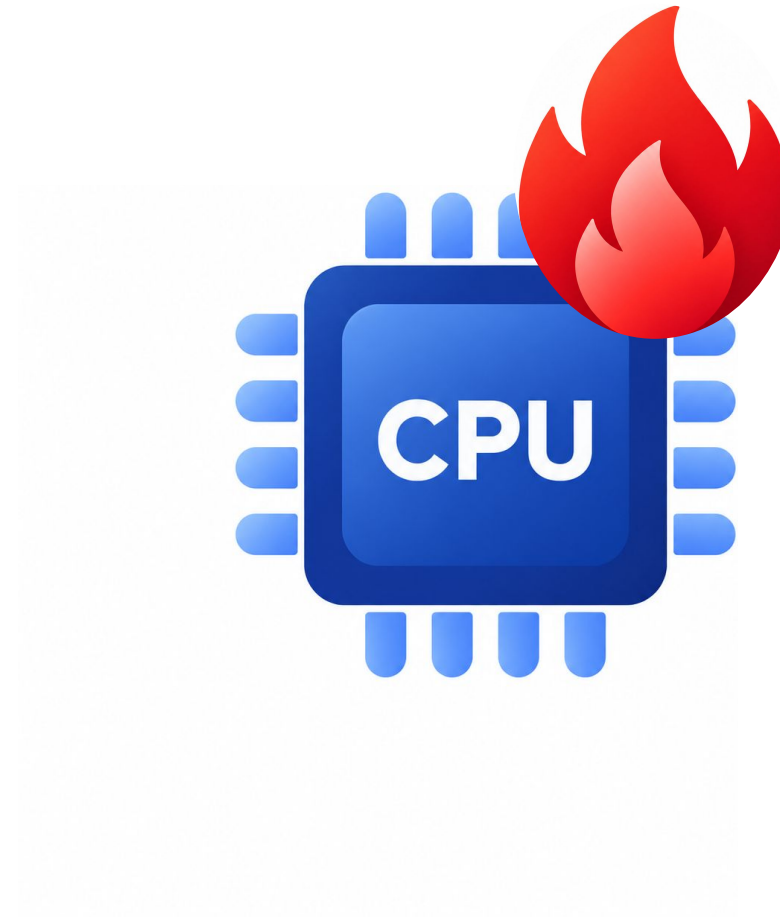
Последовательное чтение страниц

IVFFlat: стриминг

- Кластеризация (K-Means) на этапе построения
- Поиск: чтение списков подряд
- Sequential access, prefetcher работает
- Сложность: $O(N/\text{lists} * \text{probes})$ — но линейное чтение
- На практике: sequential scan по кластерам



Главная проблема IVFFlat



SIMD в vector_dist.c

Миф AVX

```
__m256 a = _mm256_load_ps(...)  
__m256 b = _mm256_load_ps(...)  
__m256 diff = _mm256_sub_ps(a, b)  
__m256 sq = _mm256_mul_ps(diff, diff)  
result = _mm256_add_ps(...)
```

Миф AVX

```
__m256 a = _mm256_load_ps(...)  
__m256 b = _mm256_load_ps(...)  
__m256 diff = _mm256_sub_ps(a, b)  
__m256 sq = _mm256_mul_ps(diff, diff)  
result = _mm256_add_ps(...)
```

Правда

```
VECTOR_TARGET_CLONES static float  
VectorL2SquaredDistance (int dim, float*  
a, float* b) {  
    distance = 0.0;  
    for (int i = 0; i < dim; i++) {  
        diff = a[i] - b[i];  
        distance += diff * diff;  
    }  
    return distance;  
}
```

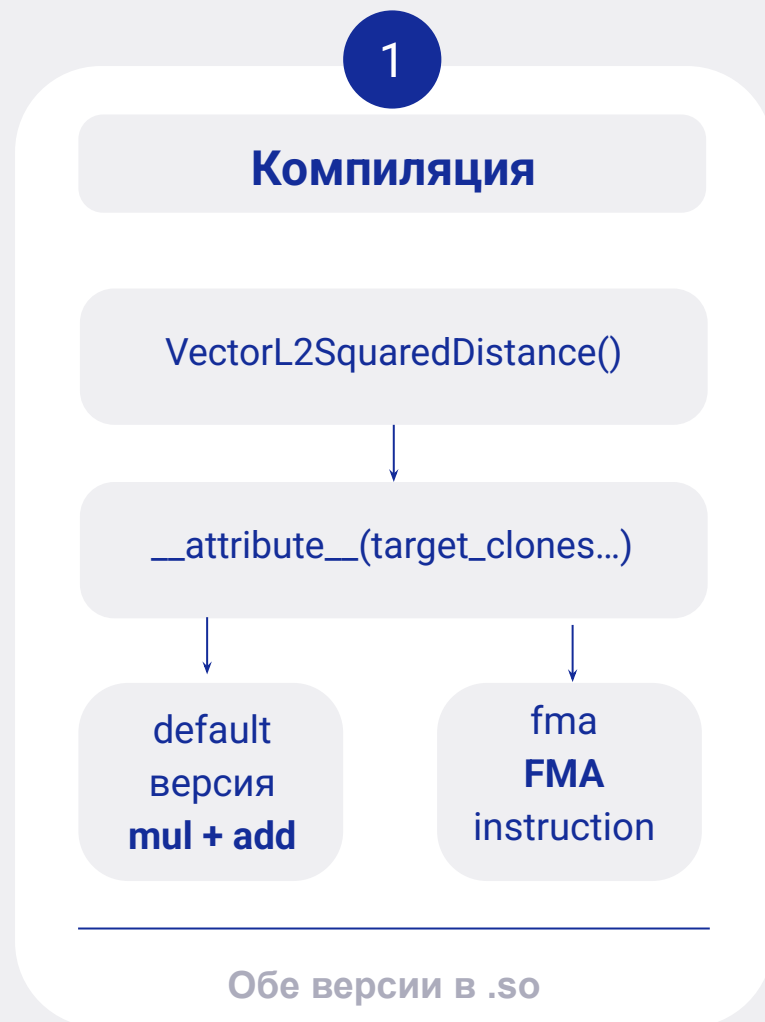
Миф AVX

```
__m256 a = _mm256_load_ps(...)  
  
__m256 b = _mm256_load_ps(...)  
  
__m256 diff = _mm256_sub_ps(a, b)  
  
__m256 sq = _mm256_mul_ps(diff, diff)  
  
result = _mm256_add_ps(...)
```

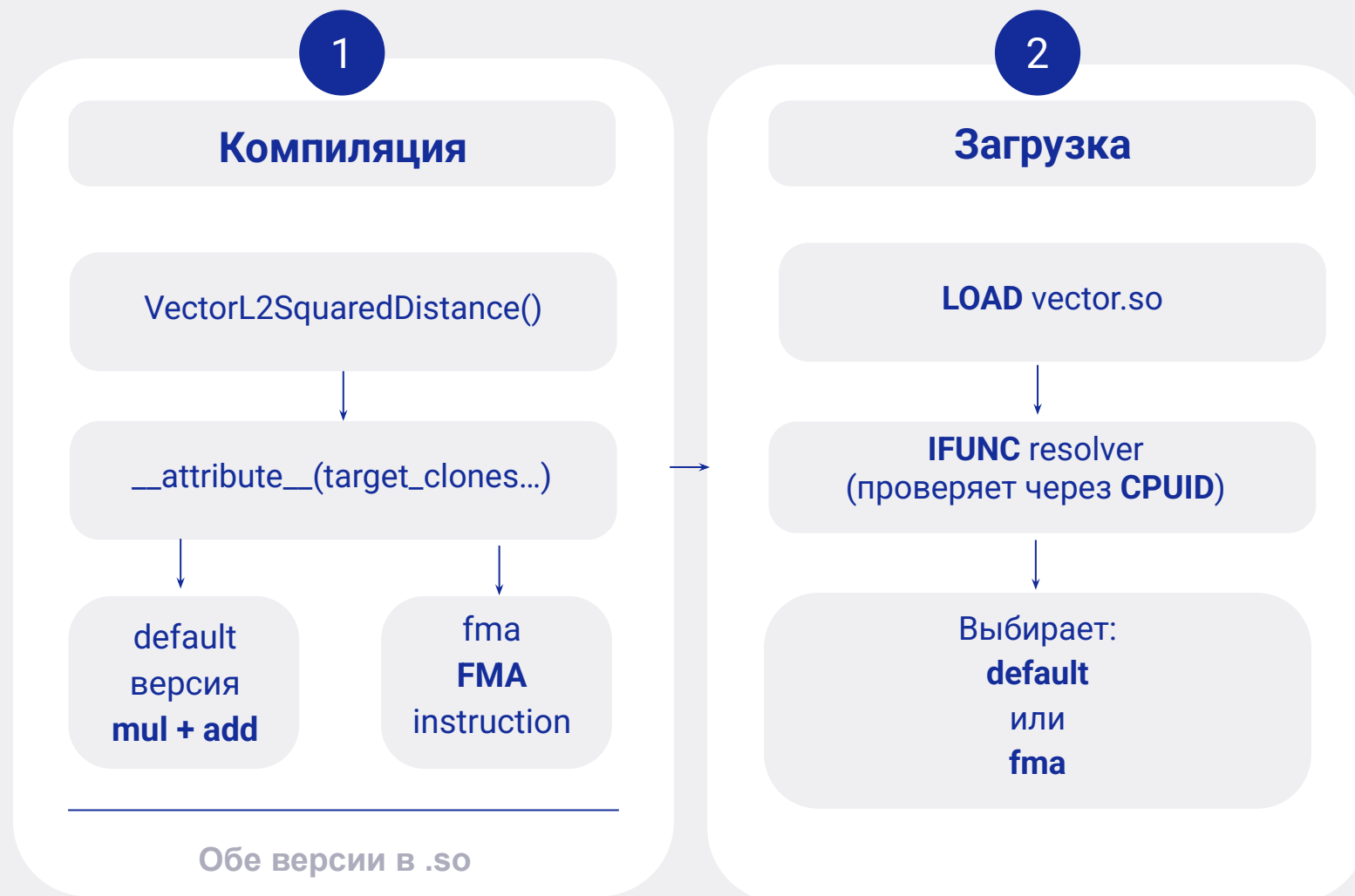
Правда

```
VECTOR_TARGET_CLONES static float  
VectorL2SquaredDistance (int dim, float*  
a, float* b) {  
    distance = 0.0;  
    for (int i = 0; i < dim; i++) {  
        diff = a[i] - b[i];  
        distance += diff * diff;  
    }  
    return distance;  
}
```

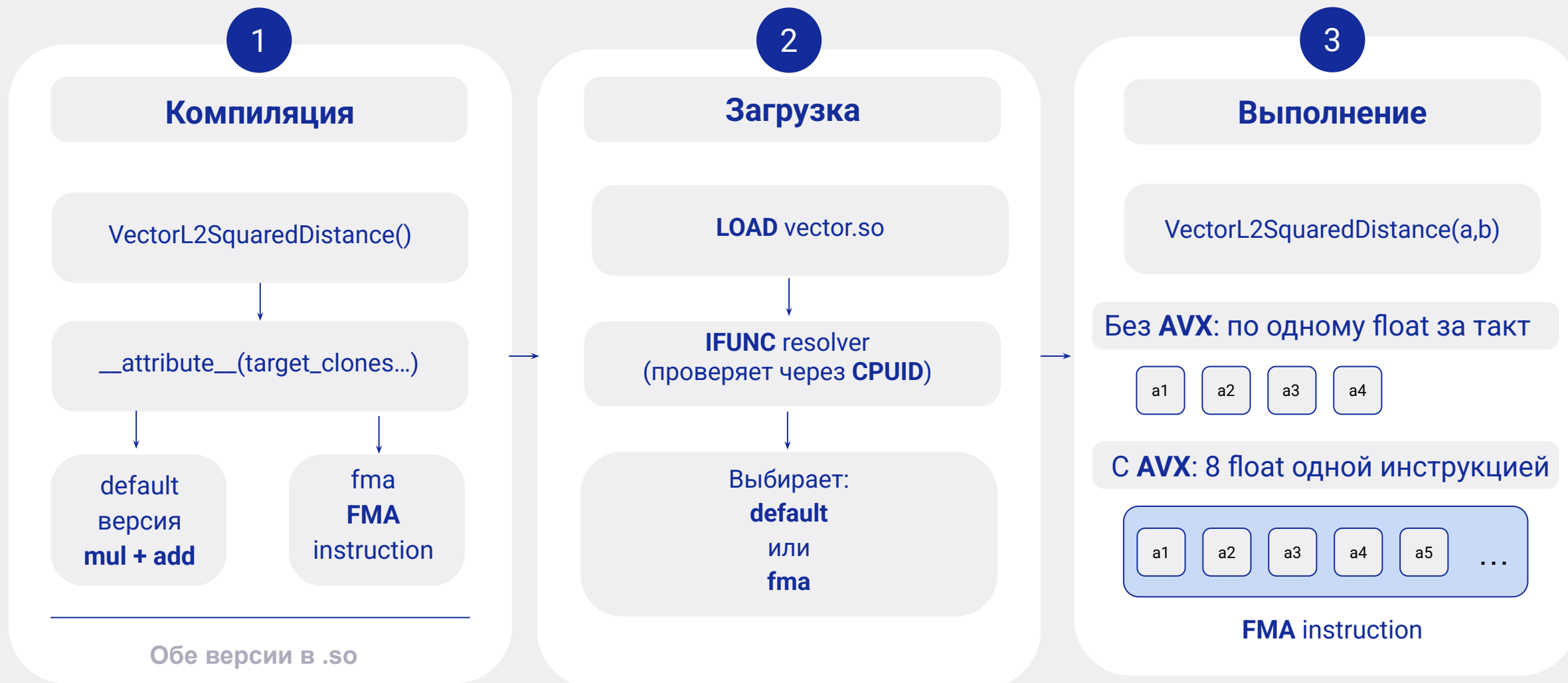
SIMD в vector_dist.c



SIMD в vector_dist.c



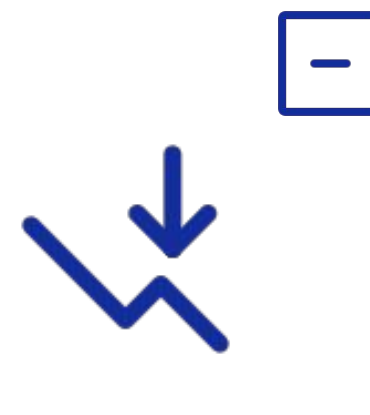
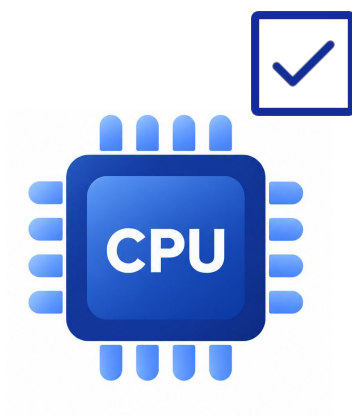
SIMD в vector_dist.c



Значит IVFFlat наш выбор?

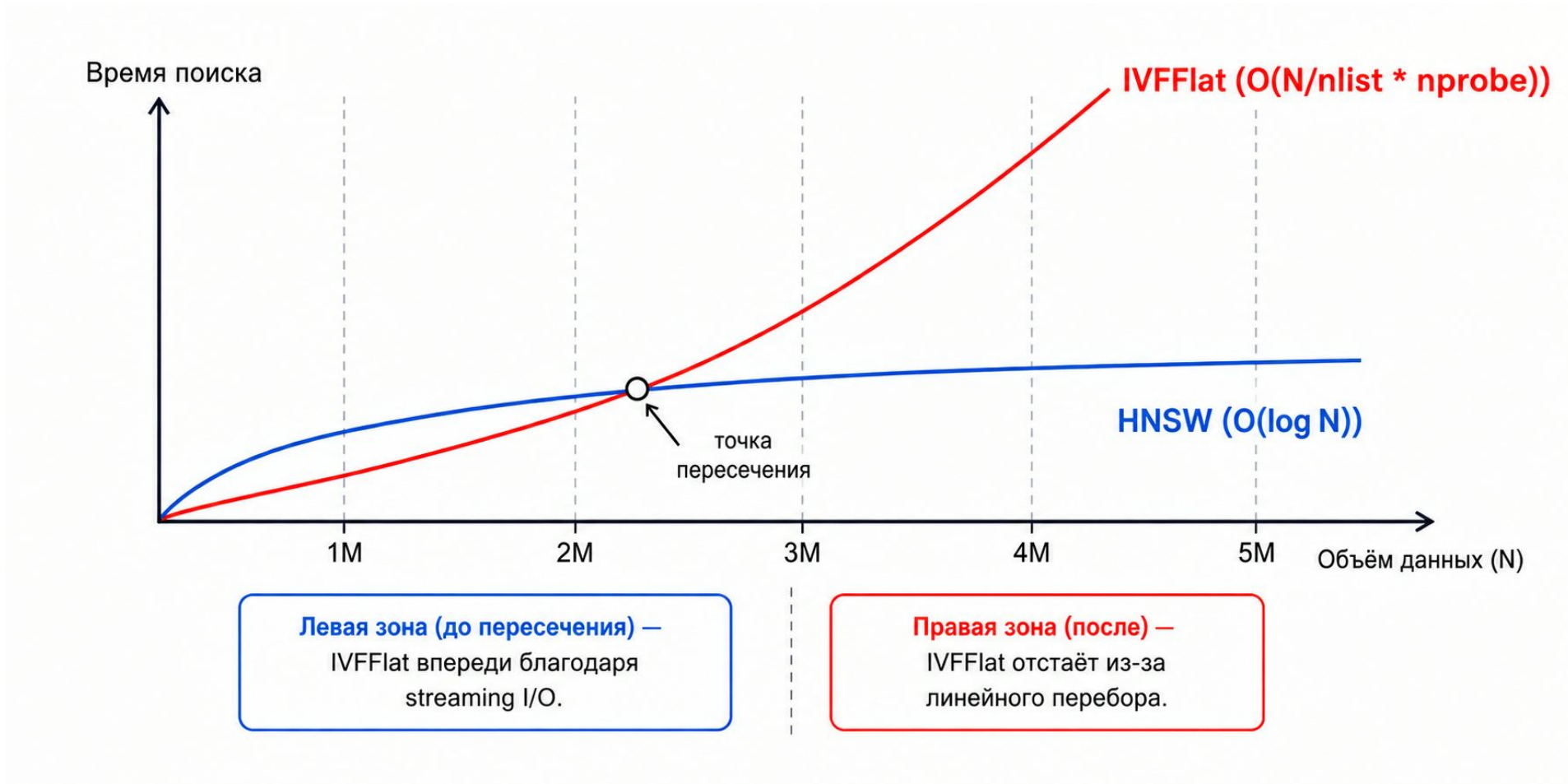


Значит IVFFlat наш выбор?



IVFFlat: сложность побеждает

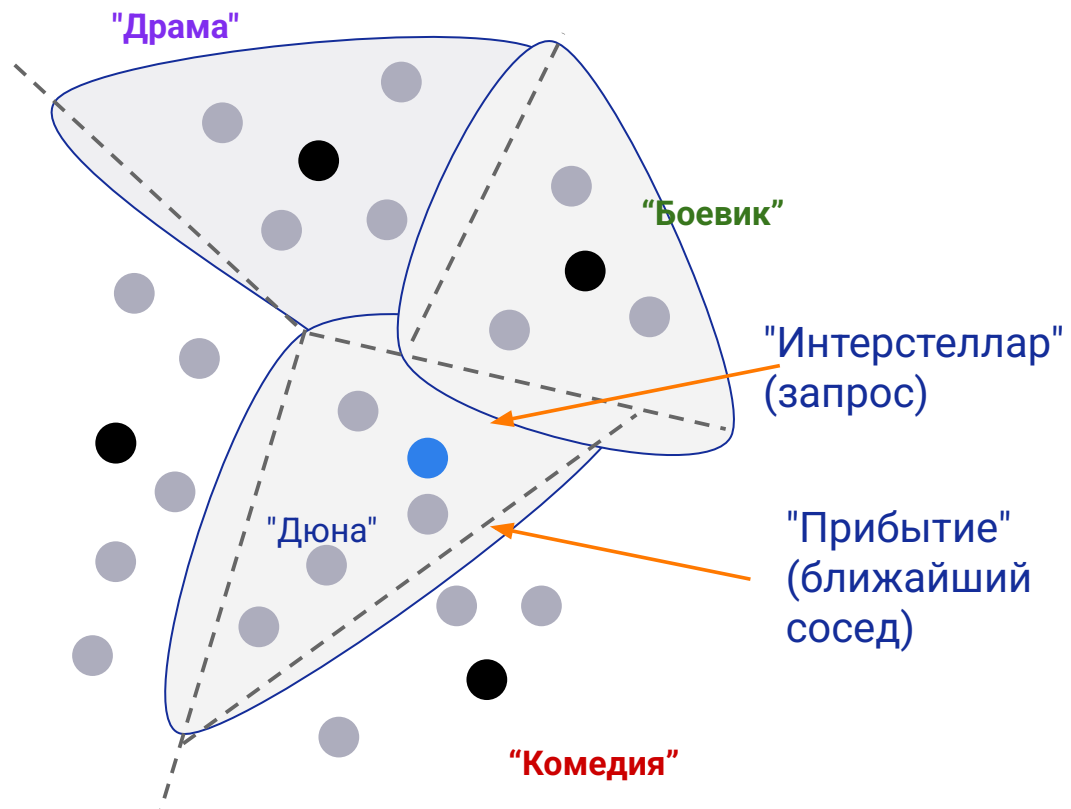
Слабое место IVFFlat



Слабое место IVFFlat

Шаг 1: probes = 3

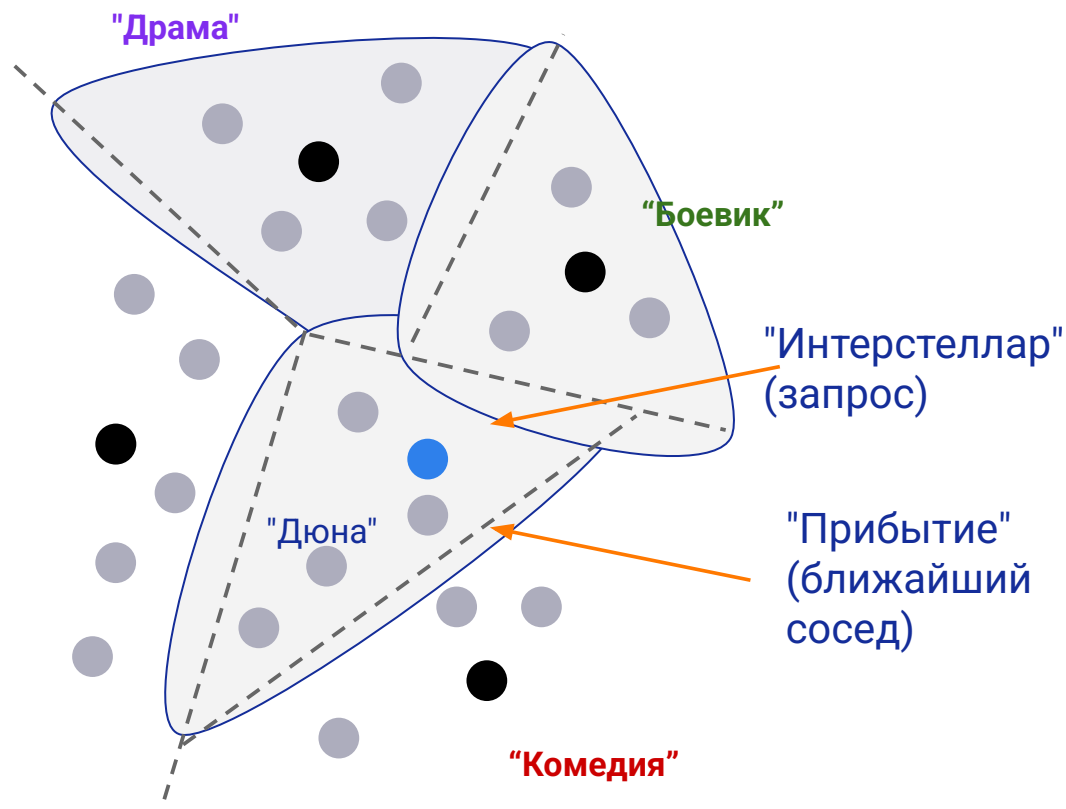
Выбираем 3 ближайших кластера по метрике



Слабое место IVFFlat

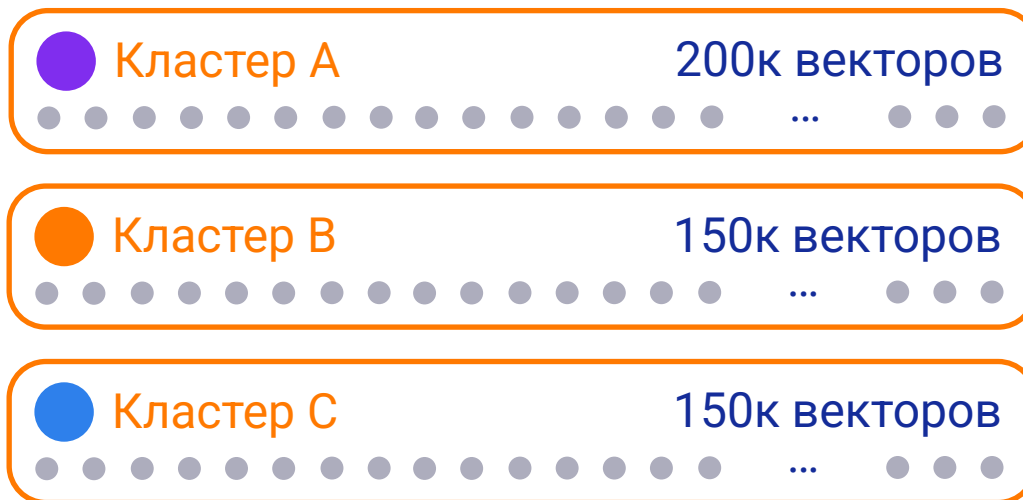
Шаг 1: probes = 3

Выбираем 3 ближайших кластера по метрике



Шаг 2: внутри кластеров

Полный перебор (brute-force)



Всего: 500K brute-force

Что в итоге с индексами?

IVFFLAT

HNSW

IVFFLAT

- Прост в реализации

HNSW

IVFFLAT

- Прост в реализации

HNSW

- Сложнее для кеша

IVFFLAT

- Прост в реализации
- Хорош при малых данных

HNSW

- Сложнее для кеша

IVFFLAT

- Прост в реализации
- Хорош при малых данных

HNSW

- Сложнее для кеша
- Больше случайных переходов

IVFFLAT

- Прост в реализации
- Хорош при малых данных
- Дegradiрует на больших данных ~ $O(N/\text{lists} * \text{probes})$

HNSW

- Сложнее для кеша
- Больше случайных переходов

IVFFLAT

- Прост в реализации
- Хорош при малых данных
- Дegradiрует на больших данных ~ $O(N/\text{lists} * \text{probes})$

HNSW

- Сложнее для кеша
- Больше случайных переходов
- Хорошо масштабируется

IVFFLAT

- Прост в реализации
- Хорош при малых данных
- Дegradiрует на больших данных $\sim O(N/\text{lists} * \text{probes})$
- Упирается в память и I/O

HNSW

- Сложнее для кеша
- Больше случайных переходов
- Хорошо масштабируется

IVFFLAT

- Прост в реализации
- Хорош при малых данных
- Дegradiрует на больших данных ~ $O(N/\text{lists} * \text{probes})$
- Упирается в память и I/O

HNSW

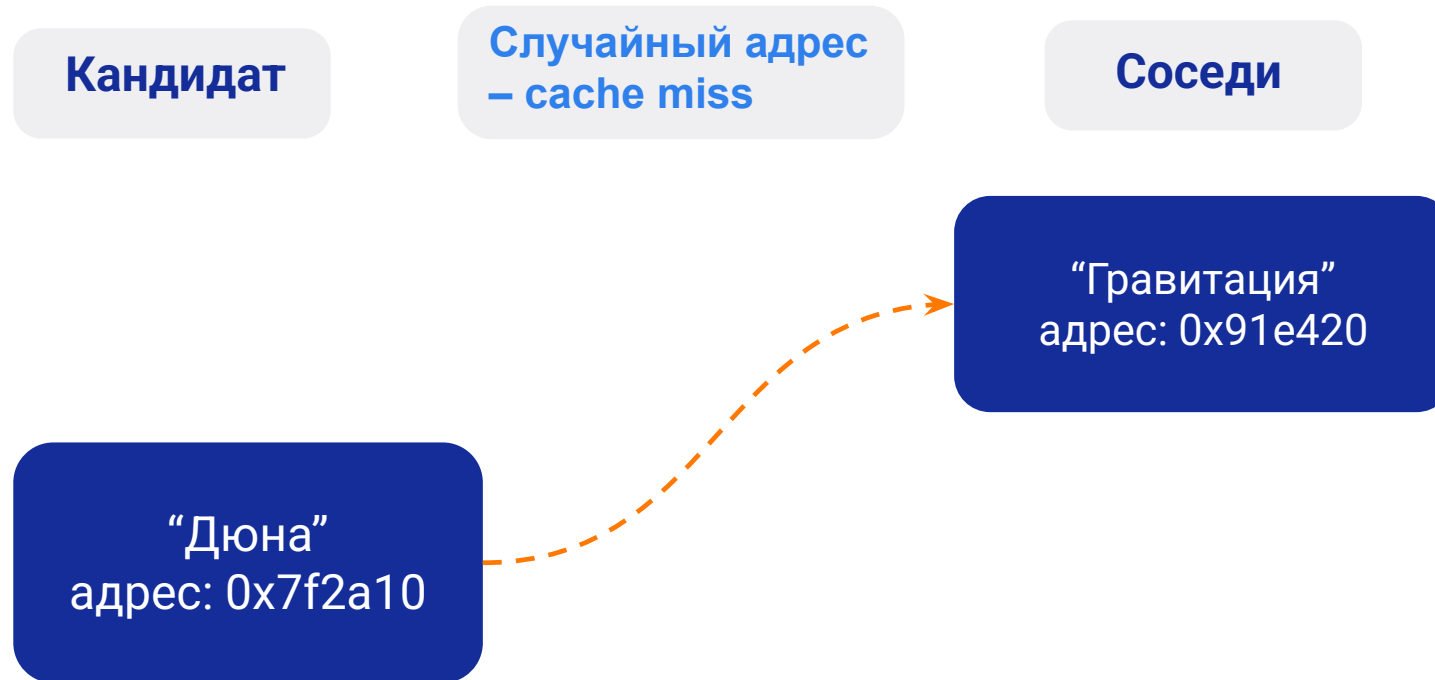
- Сложнее для кеша
- Больше случайных переходов
- Хорошо масштабируется
- Стабильно работает на больших данных

Теперь к практике



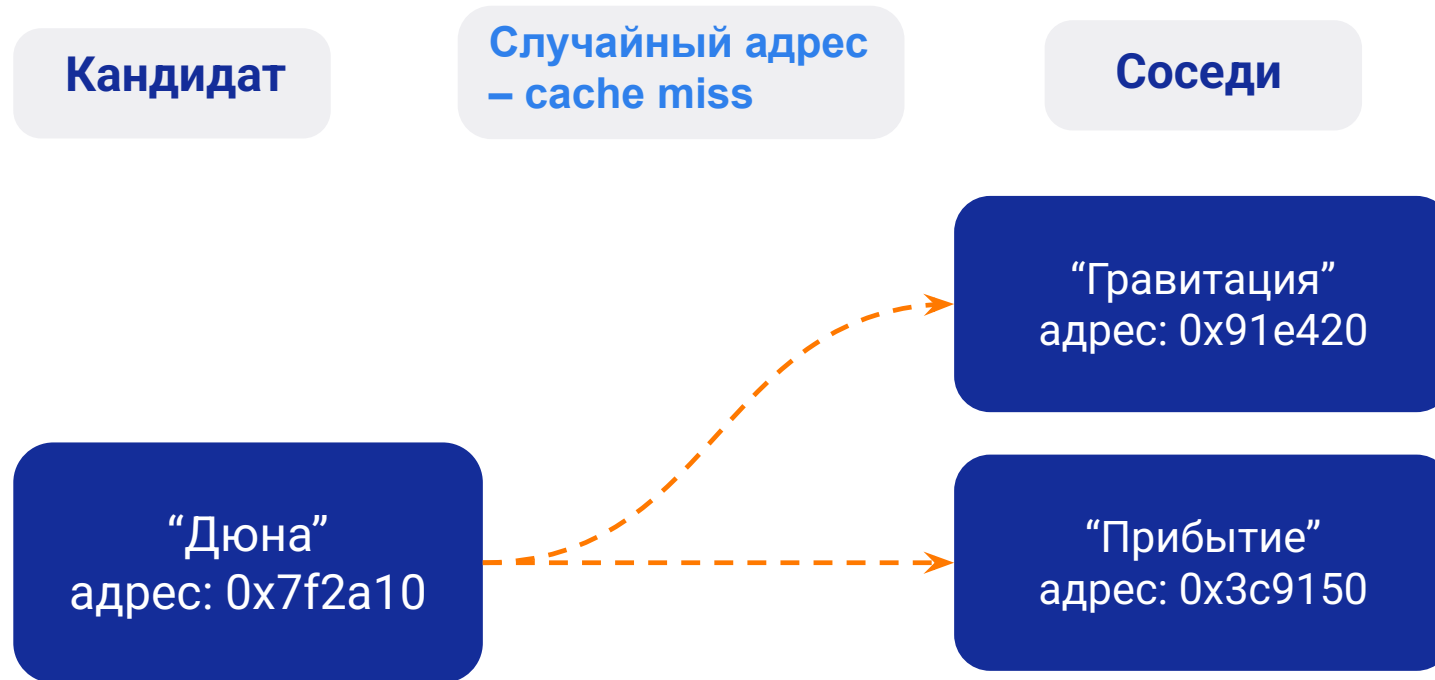
Профилирование: код против железа

Доказательство: Perf и 43% Cache Misses



```
$ perf stat -e cache-misses,instructions,cycles pgvector_query
```

Доказательство: Perf и 43% Cache Misses



```
$ perf stat -e cache-misses,instructions,cycles pgvector_query
```

Доказательство: Perf и 43% Cache Misses



```
$ perf stat -e cache-misses,instructions,cycles pgvector_query
```

Доказательство: Perf и 43% Cache Misses



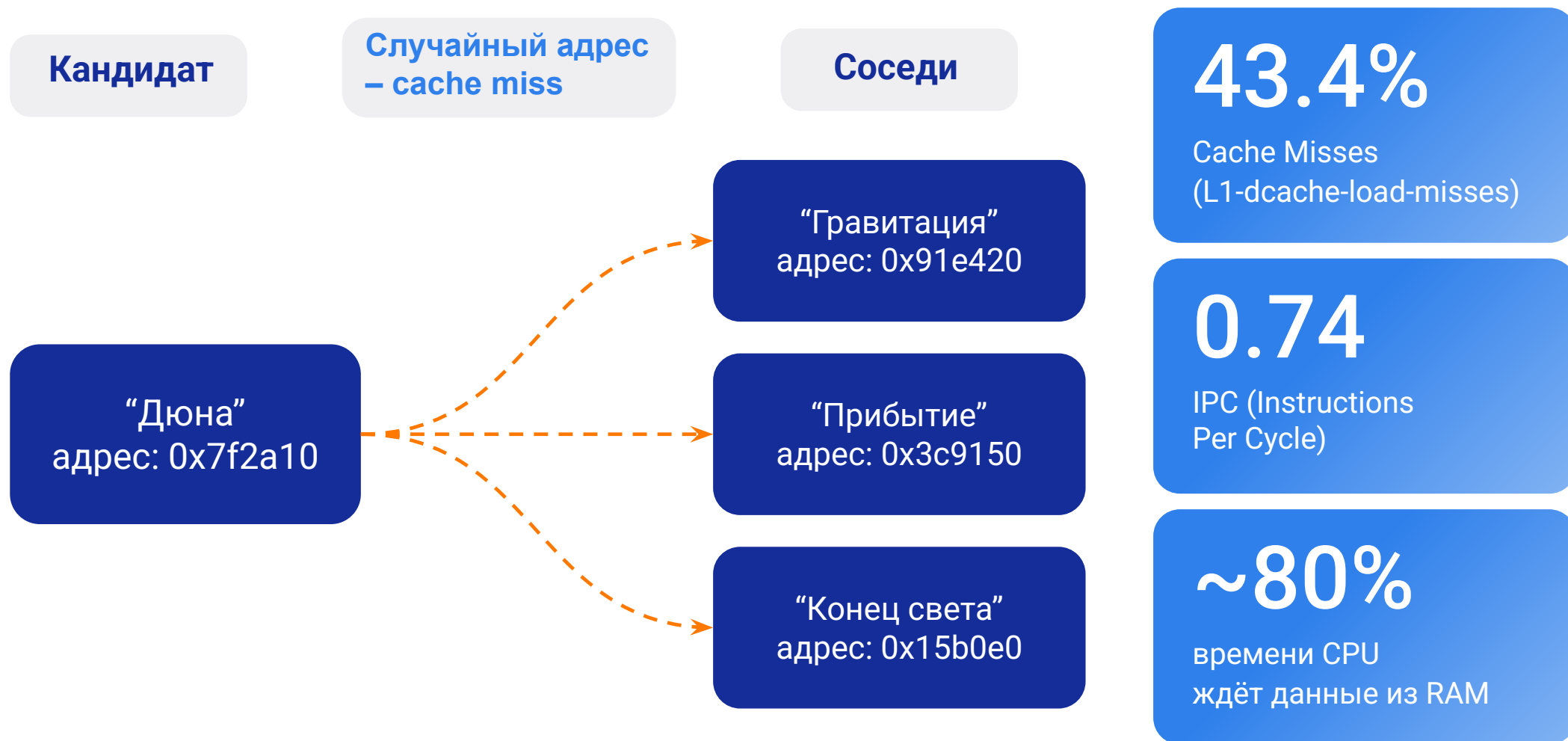
```
$ perf stat -e cache-misses,instructions,cycles pgvector_query
```

Доказательство: Perf и 43% Cache Misses



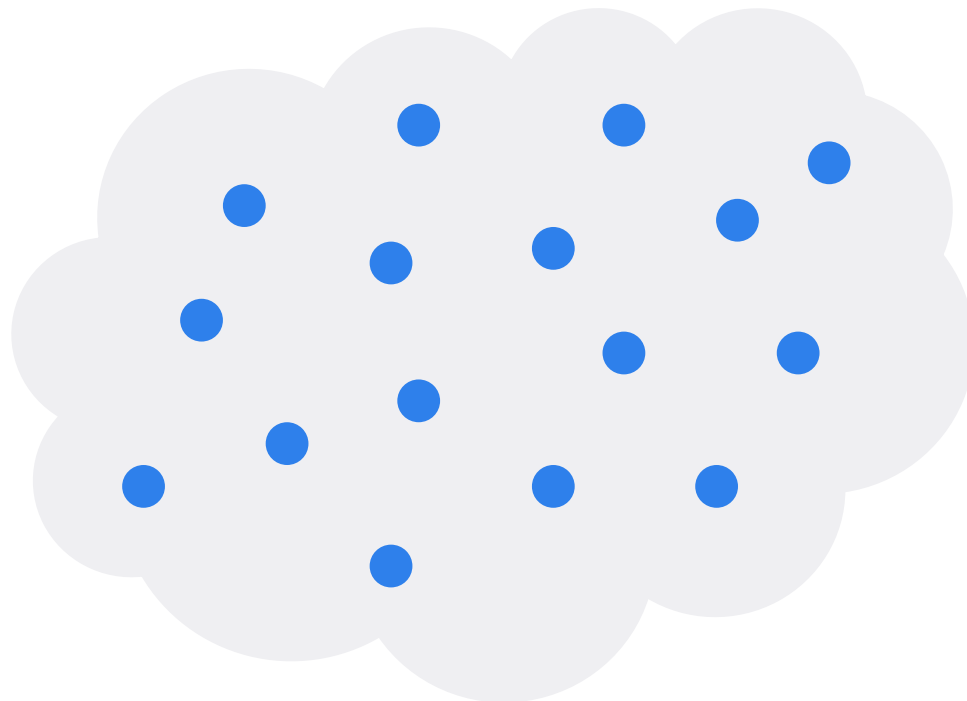
```
$ perf stat -e cache-misses,instructions,cycles pgvector_query
```

Доказательство: Perf и 43% Cache Misses

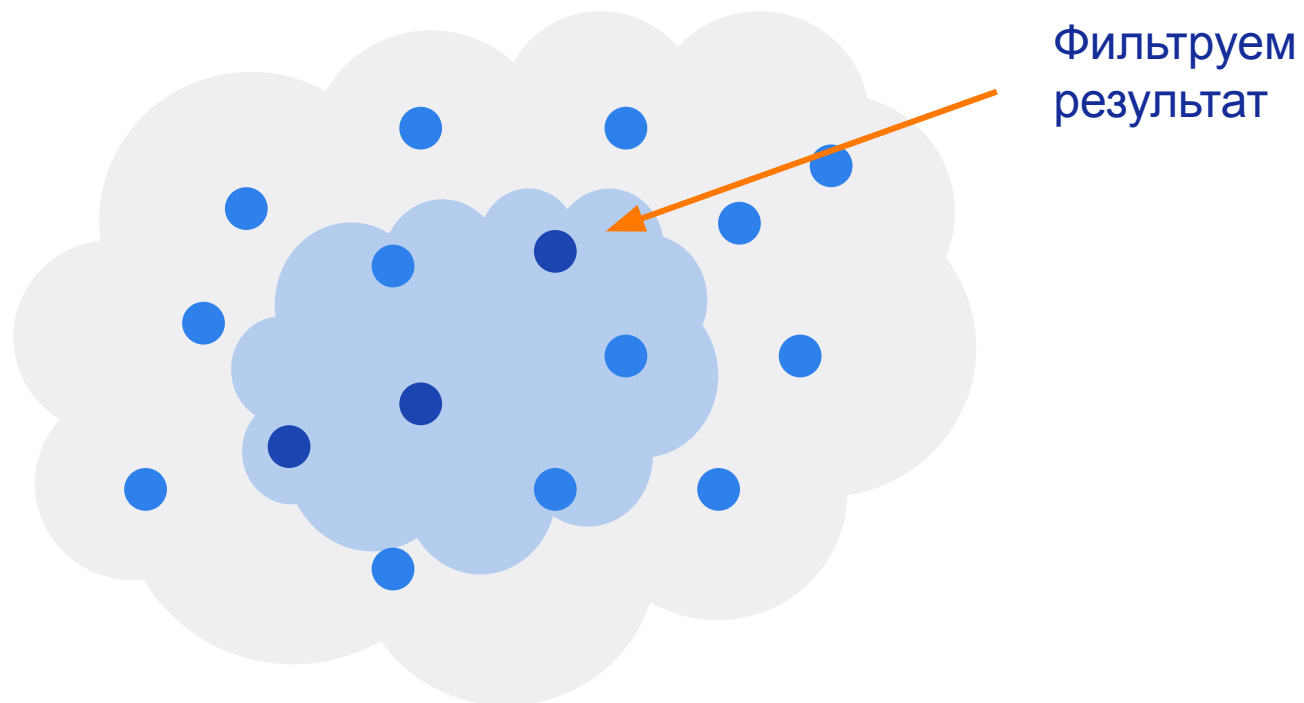


```
$ perf stat -e cache-misses,instructions,cycles pgvector_query
```

Стандартная ситуация



Стандартная ситуация



Загадка о потерянном recall

Загадка о потерянном recall

SQL - запрос

SELECT

id,
title,
platform

FROM movies

ORDER BY embedding

<=> :query_embedding

LIMIT 100;



Загадка о потерянном recall

SQL - запрос

```
SELECT
  id,
  title,
  platform
FROM movies
WHERE release=2026
ORDER BY embedding
<=> :query_embedding
LIMIT 100;
```



Загадка о потерянном recall

SQL - запрос

```
SELECT
  id,
  title,
  platform
FROM movies
WHERE release=2026
ORDER BY embedding
<=> :query_embedding
LIMIT 100;
```



Загадка о потерянном recall

SQL - запрос

SELECT

id,
title,
platform

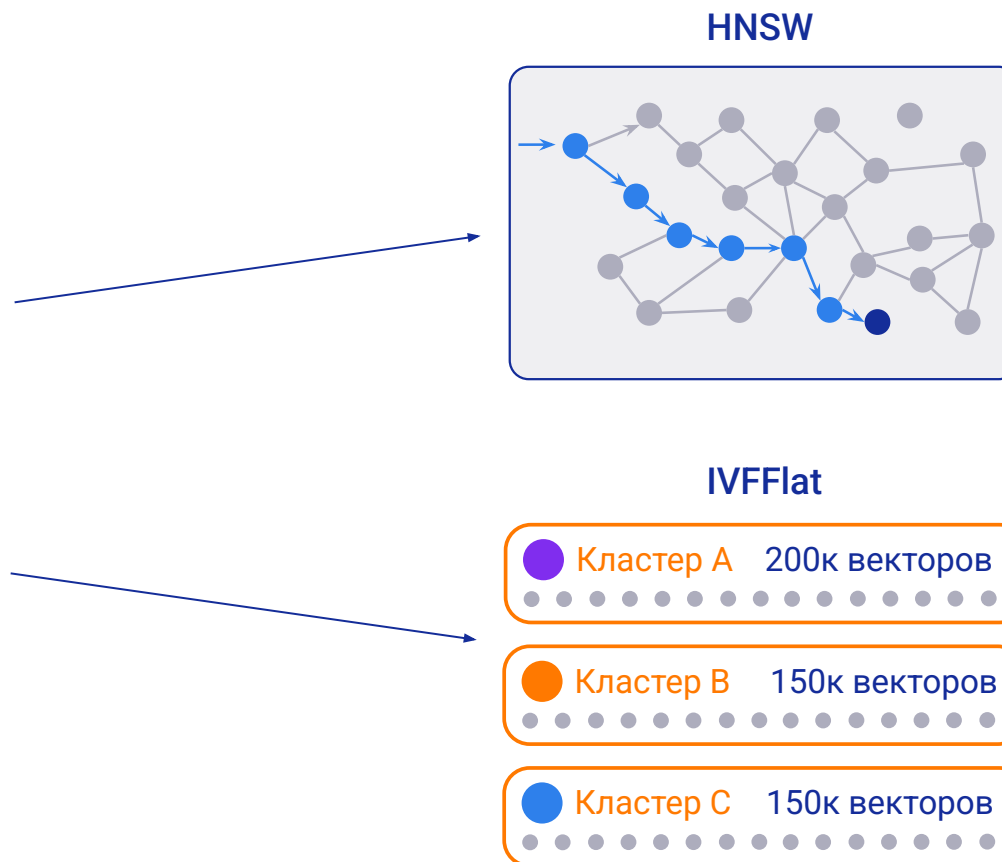
FROM movies

WHERE release=2026

ORDER BY embedding

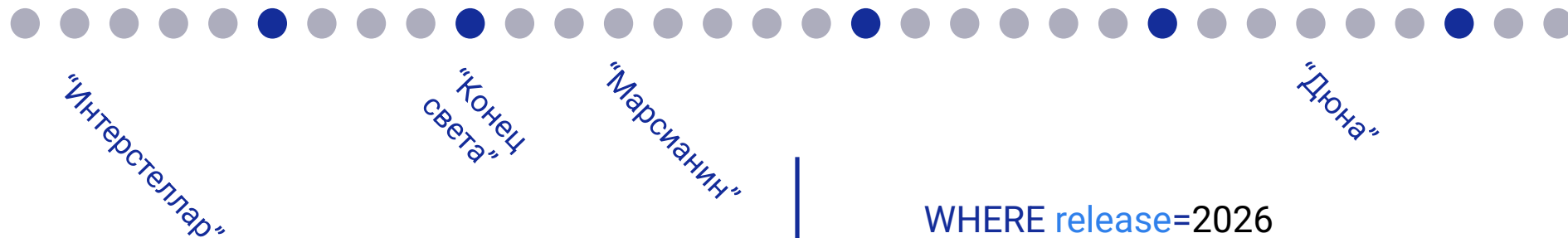
<=> :query_embedding

LIMIT 100;

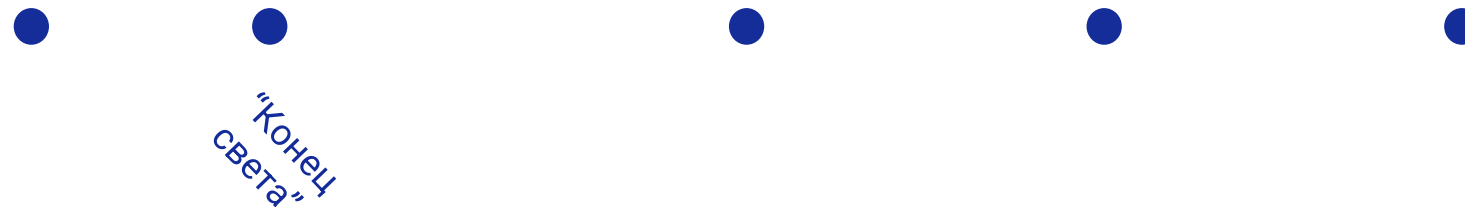


Загадка о потерянном recall

Шаг 1. Топ-100 ближайших соседей



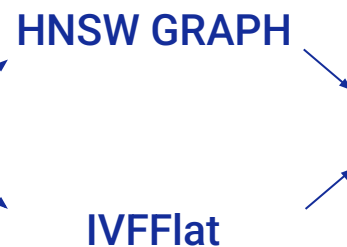
Шаг 2. Применяем фильтрацию



Загадка о потерянном recall

SQL - запрос

```
SELECT
  id,
  title,
  platform
FROM movies
WHERE release=2026
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```



Загадка о потерянном recall

SQL - запрос

```
SELECT
  id,
  title,
  platform
FROM movies
WHERE release=2026
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```



HNSW GRAPH

IVFFlat

Recall

WARNING

Загадка о потерянном recall

SQL - запрос

```
SELECT
  id,
  title,
  platform
FROM movies
WHERE release=2026
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```



HNSW GRAPH

IVFFlat

resume_scan()

...



Загадка о потерянном recall

SQL - запрос

```
SELECT
  id,
  title,
  platform
FROM movies
WHERE release=2026
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```



HNSW GRAPH

IVFFlat

resume_scan()

...



GUARD RAILS

hnsw_max_scan_tuples



work_mem x
scan_mem_multiplier



max_probes



Загадка о потерянном recall

SQL - запрос

```
SELECT
  id,
  title,
  platform
FROM movies
WHERE release=2026
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```



HNSW GRAPH

IVFFlat

resume_scan()

...



GUARD RAILS

hnsw_max_scan_tuples



work_mem x
scan_mem_multiplier



max_probes



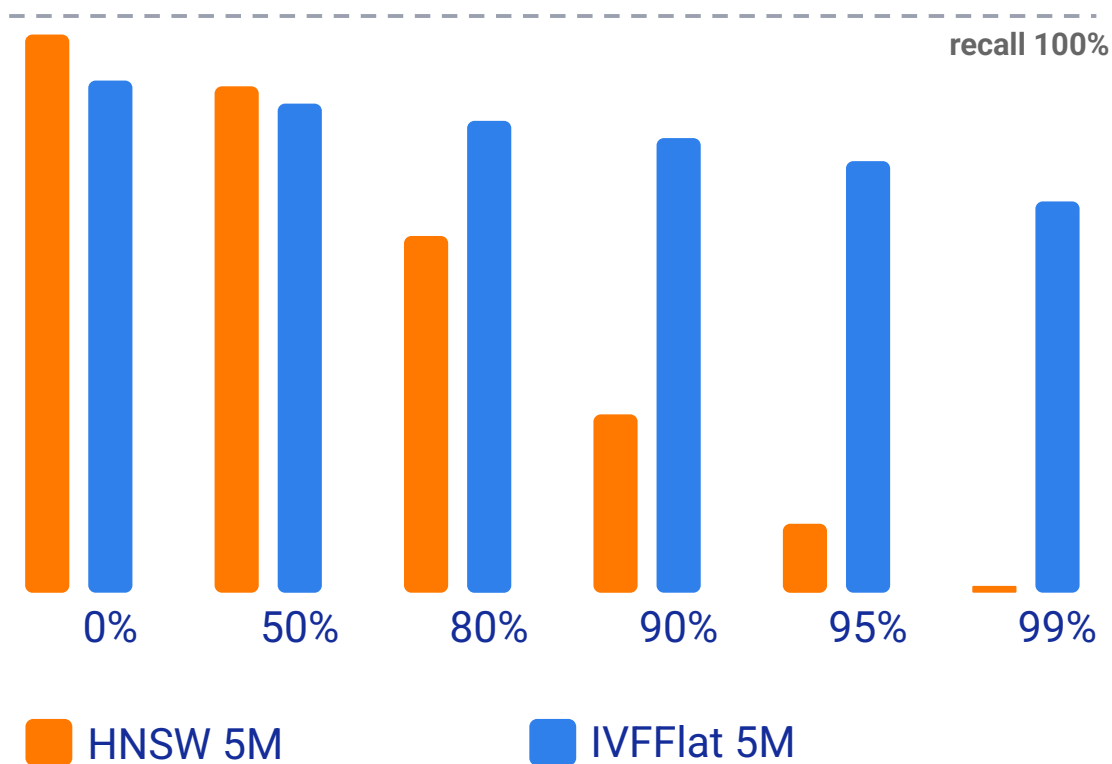
Recall



WARNING

Загадка о потерянном recall

RECALL при росте фильтрации (5M векторов)



% отфильтрованных строк (WHERE)

При фильтрации 99% на 5M векторов:
Recall = 1.2%

HNSW находит в среднем около 1 релевантного результата из каждых 83 — остальное мусор.

Причина:

- Граф строится по всем данным
 - `ef_search` набирает кандидатов вслепую без фильтра
 - Фильтрация идет на уже найденных кандидатах
 - Стоп на числе кандидатов, а не на пройденном фильтре
- `ef_search` ×10 (128→1280) поднимает Recall до ~85%, но QPS падает в 8× (340→42)

Рекомендации

Подводя итог

1

SQL → AM

```
Select ...  
ORDER BY  
vector <-> '[...]'  
LIMIT 100
```



Подводя итог

1

SQL → AM

```
Select ...  
ORDER BY  
vector <-> '[...]'  
LIMIT 100
```



2

Страница на диске



Подводя итог

1

SQL → AM

```
Select ...  
ORDER BY  
vector <-> '[...]'  
LIMIT 100
```



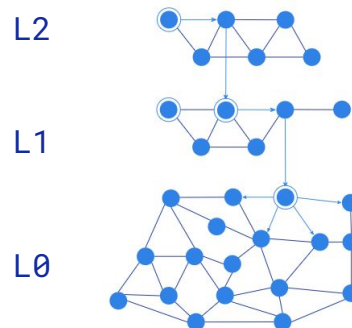
2

Страница на диске

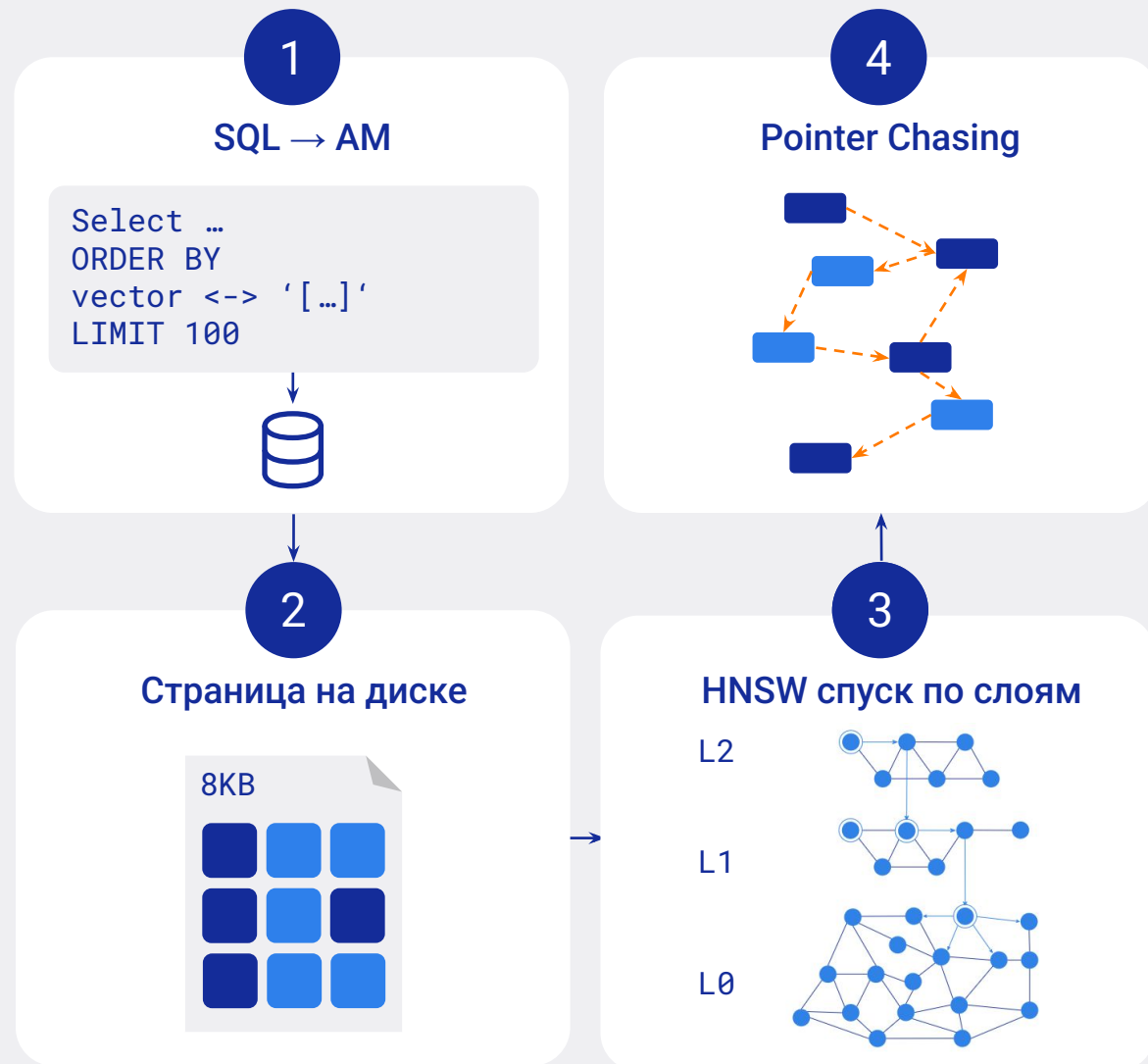


3

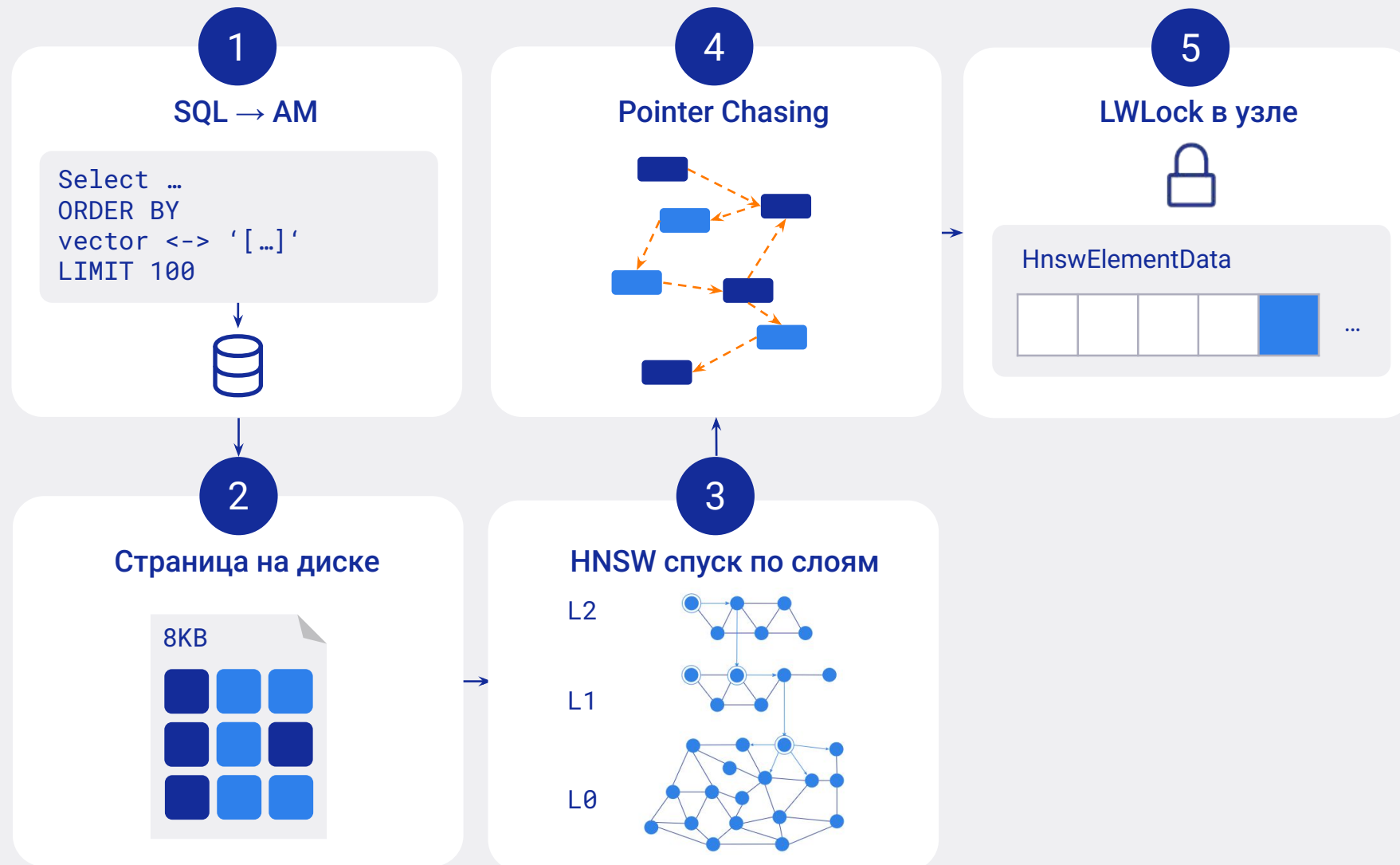
HNSW спуск по слоям



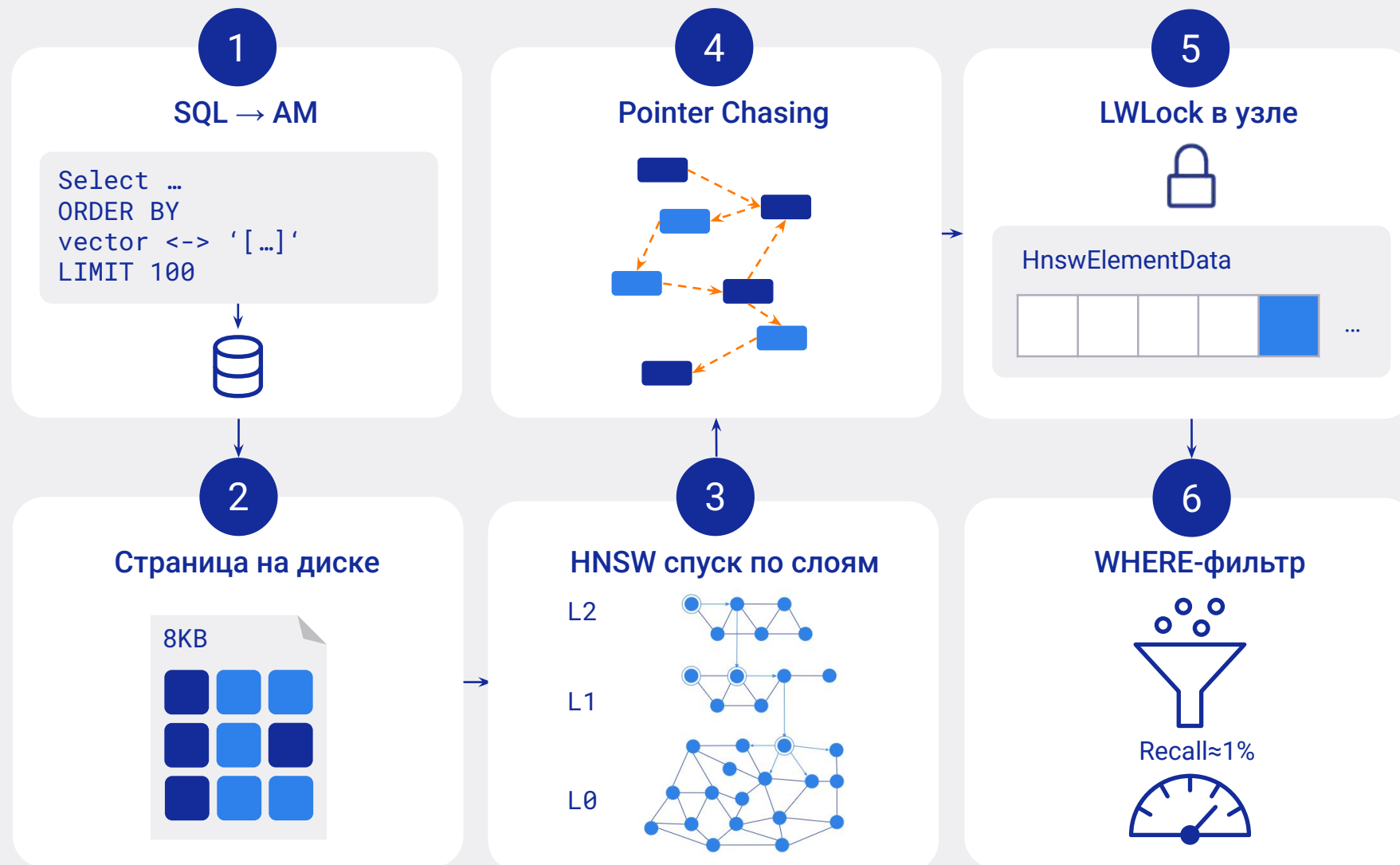
Подводя итог



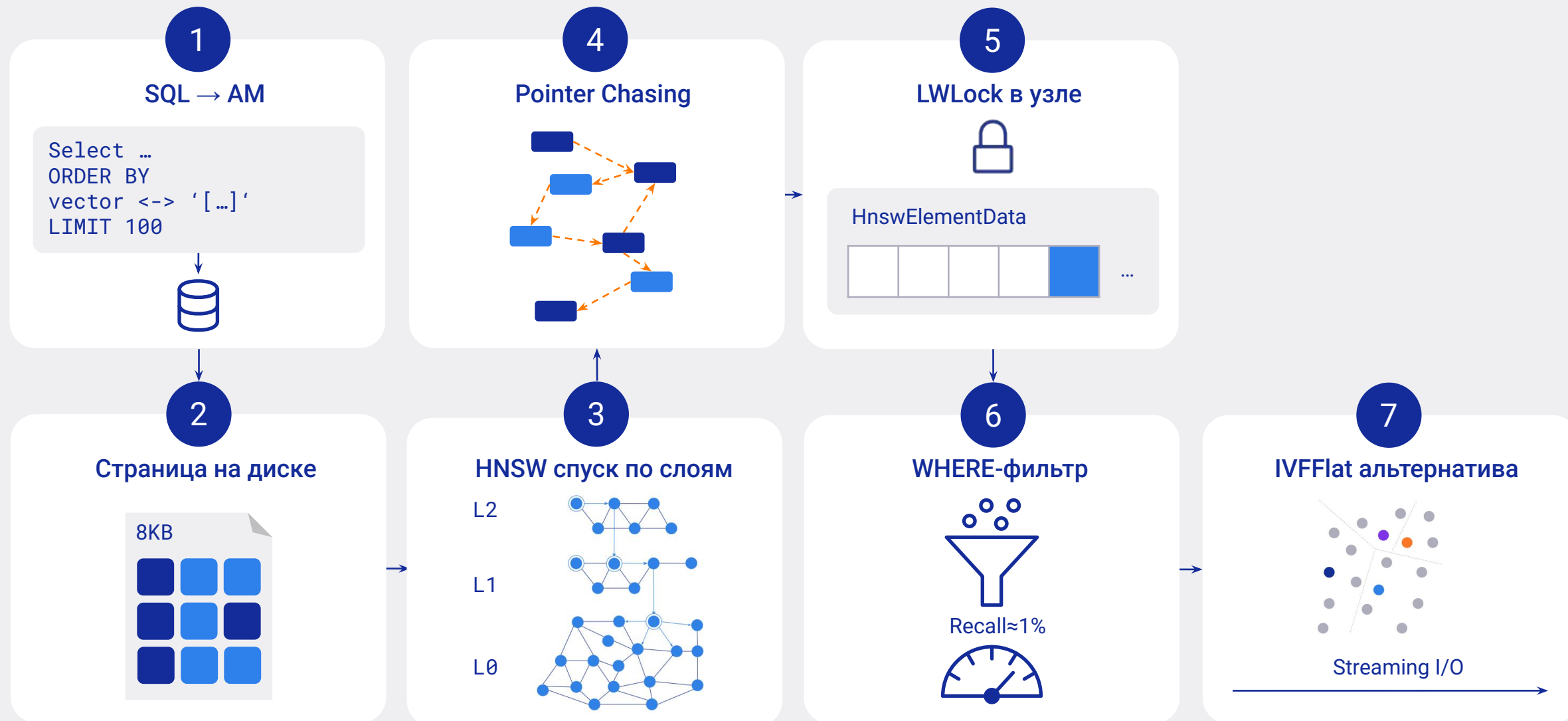
Подводя итог



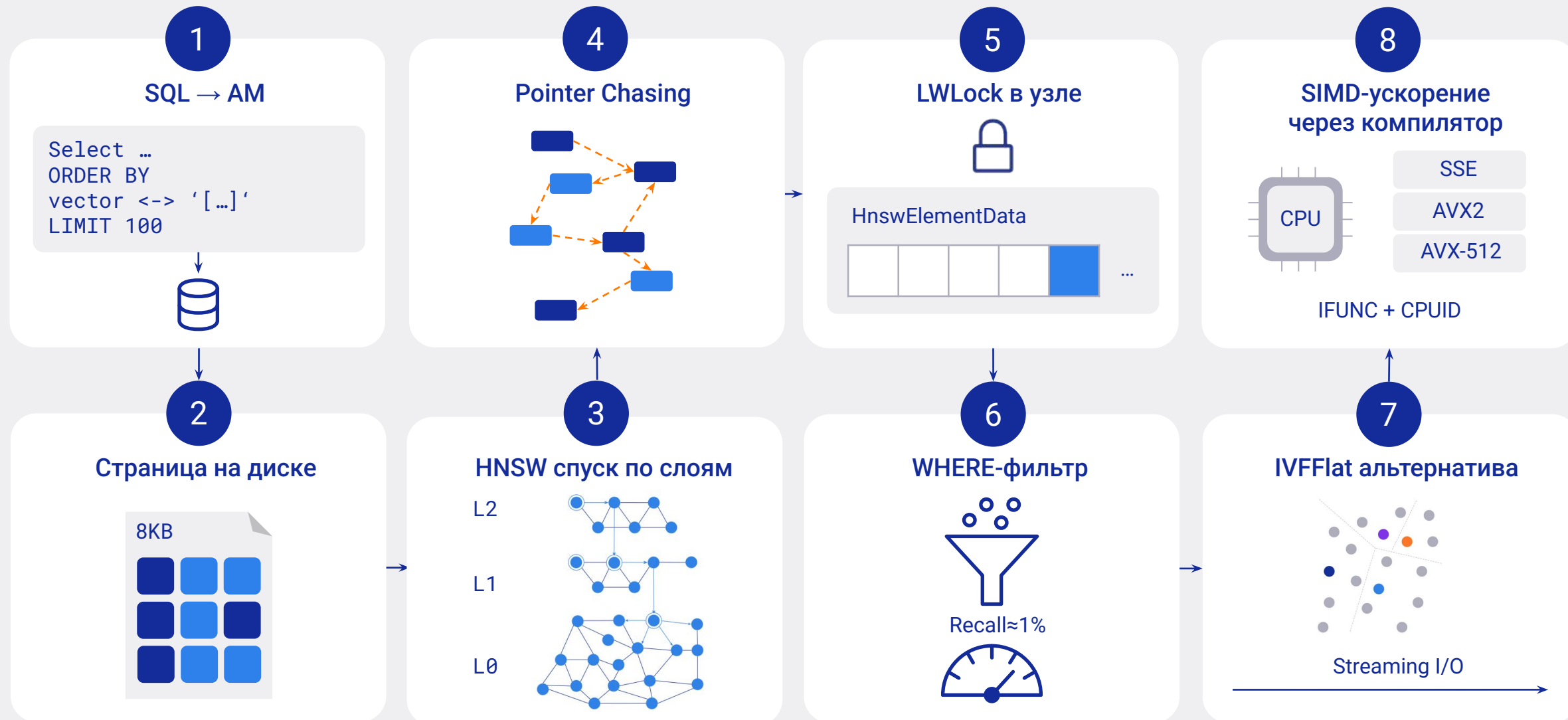
Подводя итог



Подводя итог



Подводя итог



Инженерный выбор: Где ваш порог?

Какой индекс выбрать?

IVFFlat

- **lists**
 - **lists** малый → крупные списки, быстрый build, дорогой перебор внутри списка
 - **lists** большой ($\approx \text{rows}/1000$) → точные списки, но требует роста probes для recall
- **probes (runtime)**
 - повышать, если recall недостаточен
 - recall и фильтрация

Расчет:

$\text{lists} = \text{rows}/1000$ ($\leq 1\text{M}$) или $\sqrt{\text{rows}}$ ($> 1\text{M}$)

$\text{probes} = \sqrt{\text{lists}}$ (runtime)

Инженерный выбор: Где ваш порог?

Какой индекс выбрать?

IVFFlat

- **lists**
 - lists малый → крупные списки, быстрый build, дорогой перебор внутри списка
 - lists большой ($\approx \text{rows}/1000$) → точные списки, но требует роста probes для recall
- **probes (runtime)**
 - повышать, если recall недостаточен
 - recall и фильтрация

Расчет:

$\text{lists} = \text{rows}/1000 (\leq 1\text{M})$ или $\sqrt{\text{rows}} (> 1\text{M})$
 $\text{probes} = \sqrt{\text{lists}}$ (runtime)

HNSW

- **M**
 - $m=8-12$ → быстрее строится, компактнее, но чуть ниже recall на сложных данных
 - $m>16$ → почти не даёт прироста recall, но заметно тяжелее по памяти
- **ef_construction**
 - recall насыщается уже на 64, >256 — 2× время build за $<2\%$ recall
- **ef_search (runtime)**
 - повышайте постепенно (40 → 100 → 200), если recall недостаточен

Полный перебор оправдан в двух случаях: корпус совсем маленький или корпус постоянно переписывается

Инженерный выбор: Где ваш порог?

Что с параметрами?

- **halfvec** — float16, 2x меньше памяти без заметной потери recall
- **bit** — бинарные векторы, максимальная компактность
- **sparsevec** — разреженные эмбединги (TF-IDF, SPLADE и т.п.)

Инженерный выбор: Где ваш порог?

Что с параметрами?

- **halfvec** — float16, 2x меньше памяти без заметной потери recall
- **bit** — бинарные векторы, максимальная компактность
- **sparsevec** — разреженные эмбединги (TF-IDF, SPLADE и т.п.)

Тип	Лимиты индексирования	Поддерживаемые метрики
halfvec	до 4 000 измерений	<->, <#>, <=>, <+> (все стандартные)
bit	до 64 000 измерений	только <~> (Hamming) и <%> (Jaccard)
sparsevec	до 1 000 ненулевых элементов	<->, <#>, <=> — но считается только по ненулевым позициям

Найдётся ли релевантный документ — решают chunking, модель эмбедингов и полнота корпуса, а не выбор между HNSW и IVFFlat

Что можно сделать с фильтрацией?

- Объявить дополнительный набор кандидатов, вместо жесткого отбора (off – по умолчанию)
 - `SET hnsw.iterative_scan = strict_order;` -- точный порядок результатов
 - `SET hnsw/ivfflat.iterative_scan = relaxed_order;` -- допускает разупорядоченность результатов ради лучшего recall

Recall докупается реранкером, а не размером индекса. Быстрый ANN + реранкер поверх top-k дешевле, чем гонка за recall на этапе поиска

Что можно сделать с фильтрацией?

- Объявить дополнительный набор кандидатов, вместо жесткого отбора (off – по умолчанию)
 - `SET hnsw.iterative_scan = strict_order;` -- точный порядок результатов
 - `SET hnsw/ivfflat.iterative_scan = relaxed_order;` -- допускает разупорядоченность результатов ради лучшего recall
- Мало уникальных значений фильтра
 - `CREATE INDEX ON items USING hnsw (embedding vector_cosine_ops) WHERE (release = 2026);`

Recall докупается реранкером, а не размером индекса. Быстрый ANN + реранкер поверх top-k дешевле, чем гонка за recall на этапе поиска

Что можно сделать с фильтрацией?

- Объявить дополнительный набор кандидатов, вместо жесткого отбора (off – по умолчанию)
 - `SET hnsw.iterative_scan = strict_order;` -- точный порядок результатов
 - `SET hnsw/ivfflat.iterative_scan = relaxed_order;` -- допускает разупорядоченность результатов ради лучшего recall
- Мало уникальных значений фильтра
 - `CREATE INDEX ON items USING hnsw (embedding vector_cosine_ops) WHERE (release = 2026);`
- Много уникальных значений, есть шансы повысить recall
 - `CREATE TABLE items (embedding vector(3), release int) PARTITION BY LIST(release);`

Recall докупается реранкером, а не размером индекса. Быстрый ANN + реранкер поверх top-k дешевле, чем гонка за recall на этапе поиска

**Спасибо! Ответим
на ваши вопросы**

Дарья Барсукова @brskv_dm

