PostgresPro

Векторный поиск в PostgreSQL: pgvector изнутри

Дарья Барсукова @brskv_dm

Дарья Барсукова

PostgresPro

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

Postgres Professional – разработчик самой популярной российской СУБД

PostgresPro

11 лет

на рынке с 2015

ТОП-5

в мире по вкладу
в PostgreSQL

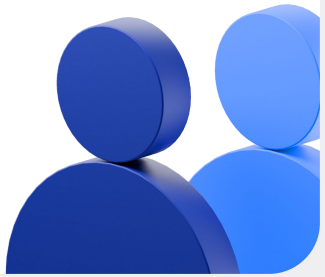
(100+ патчей ежегодно)

№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

PostgresPro

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

О чем расскажу

PostgresPro

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

01 Постановка задачи: иллюзия простоты

О чем расскажу

PostgresPro

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

PostgresPro

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

PostgresPro

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

04 IVFFlat: стриминг против прыжков и SIMD

О чем расскажу

PostgresPro

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

04 IVFFlat: стриминг против прыжков и SIMD

05 Практика: perf и загадка о потерянном recall

О чем расскажу

PostgresPro

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

04 IVFFlat: стриминг против прыжков и SIMD

05 Практика: perf и загадка о потерянном recall

06 Рекомендации

PostgresPro

Постановка задачи: иллюзия простоты

От букв к смыслам

PostgresPro

От букв к смыслам

PostgresPro

Классический подход:

- Полнотекстовый поиск

От букв к смыслам

PostgresPro

Классический подход:

- Полнотекстовый поиск
- Ищет совпадение БУКВ

От букв к смыслам

PostgresPro

Классический подход:

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

От букв к смыслам

PostgresPro

Классический подход:

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

Современная реальность:

- Векторный поиск + полнотекстовый поиск

От букв к смыслам

PostgresPro

Классический подход:

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

Современная реальность:

- Векторный поиск + полнотекстовый поиск
- Ищет совпадение СМЫСЛОВ

От букв к смыслам

PostgresPro

Классический подход:

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

Современная реальность:

- Векторный поиск + полнотекстовый поиск
- Ищет совпадение СМЫСЛОВ
- Плотные эмбеддинги

От букв к смыслам

PostgresPro

Классический подход:

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

Современная реальность:

- Векторный поиск + полнотекстовый поиск
- Ищет совпадение СМЫСЛОВ
- Плотные эмбеддинги

Цель: минимальная задержка (latency) на миллионах объектов

Пример векторного поиска

PostgresPro

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

Интерстеллар

Прибытие

Комедия

word2vec

Пример векторного поиска

PostgresPro

Эмбеддинг — 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 эмбеддинги имеют сотни и тысячи измерений.

word2vec

Пример векторного поиска

PostgresPro

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

	Sci-Fi	Драма	Экшн	Эпич.	Филос.	Юмор	Роман.	Напряж.
Интерстеллар	0.90	0.70	0.20	0.85	0.80	0.10	0.30	0.60
Прибытие	0.85	0.75	0.15	0.70	0.85	0.05	0.20	0.55
Комедия	0.05	0.20	0.10	0.10	0.05	0.95	0.80	0.15

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

word2vec

Пример векторного поиска

PostgresPro

Эмбеддинг — 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

PostgresPro

Эмбеддинг — 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

PostgresPro

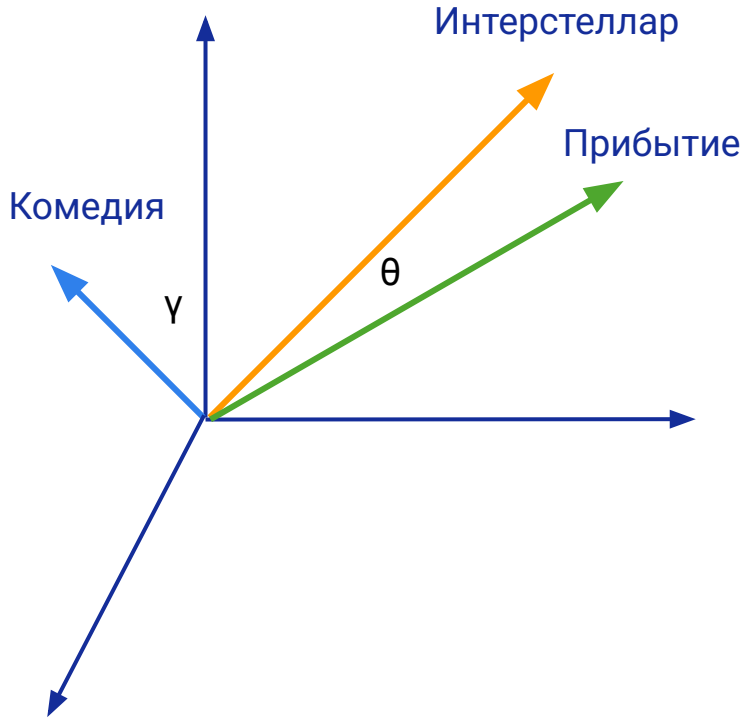
Эмбеддинг — массив чисел, который получается после преобразования текста языковой моделью.

Метрики для сравнения векторов:

- Cosine similarity

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

- Евклидова
- Манхэттенская
- Hamming



cos(Интерстеллар, Прибытие) ≈ 0.993
cos(Интерстеллар, Комедия) ≈ 0.332

word2vec

PostgresPro

Что с инфраструктурой?

Дилемма архитектора на старте

PostgresPro

Путь 1. Специализированные VectorDB

- Новые лицензии и стек

Дилемма архитектора на старте

PostgresPro

Путь 1. Специализированные VectorDB

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

Дилемма архитектора на старте

PostgresPro

Путь 1. Специализированные VectorDB

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

Дилемма архитектора на старте

PostgresPro

Путь 1. Специализированные VectorDB

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

Дилемма архитектора на старте

PostgresPro

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

PostgresPro

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

PostgresPro

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

PostgresPro

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

PostgresPro

Задача решена?

PostgresPro

Не совсем...

Столкновение с физикой железа

PostgresPro

15 мин

от установки расширения
до работающего RAG-прототипа

Столкновение с физикой железа

PostgresPro

15 мин

от установки расширения
до работающего RAG-прототипа

10x

во столько раз выросла
база данных в тесте

Столкновение с физикой железа

PostgresPro

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

PostgresPro

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

PostgresPro

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

PostgresPro

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

PostgresPro

15 мин

от установки расширения
до работающего RAG-прототипа

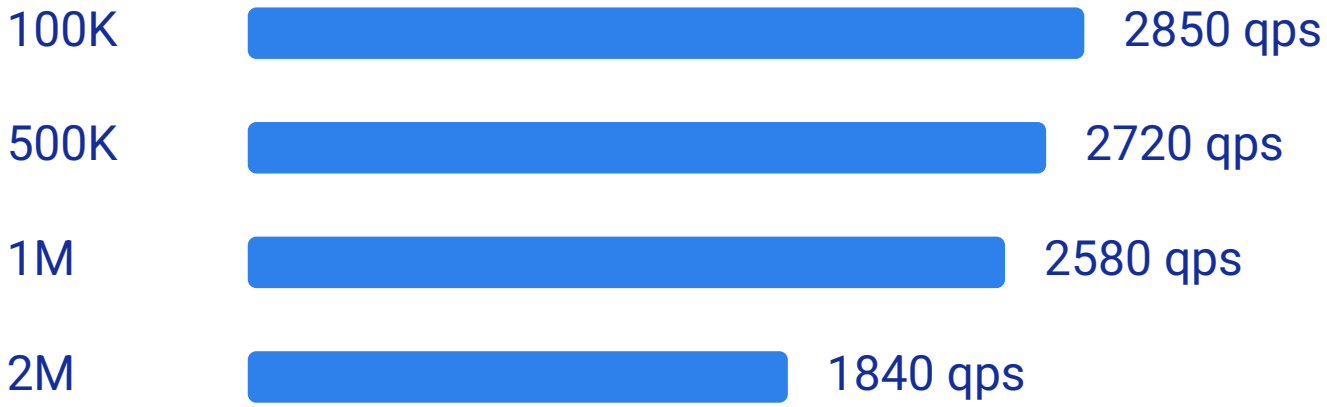
10x

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

PostgresPro

15 мин

от установки расширения
до работающего RAG-прототипа

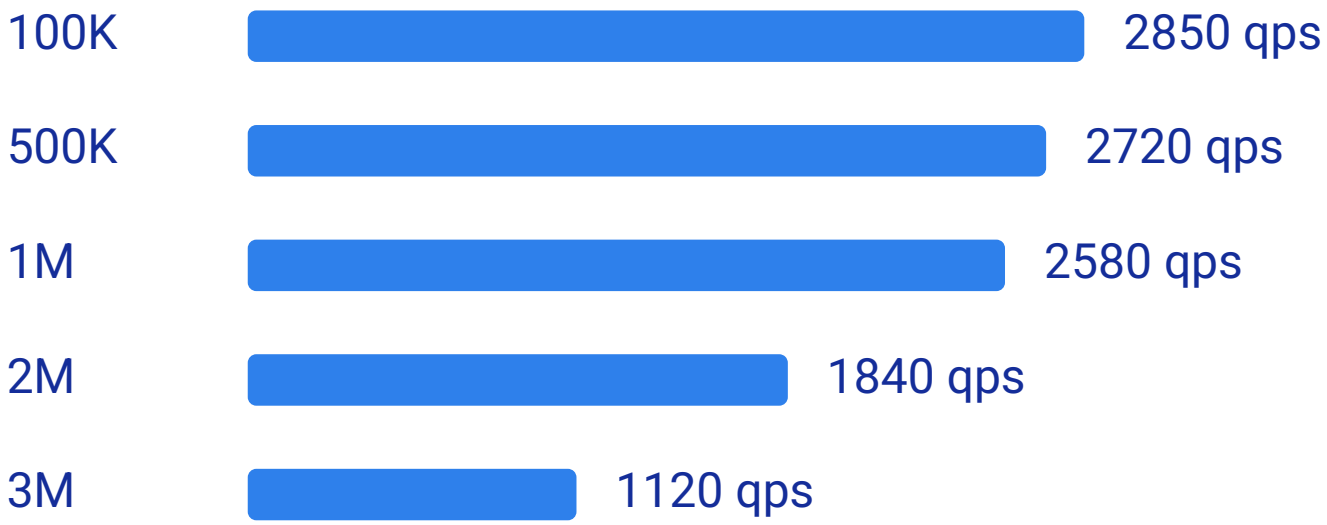
10x

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

PostgresPro

15 мин

от установки расширения
до работающего RAG-прототипа

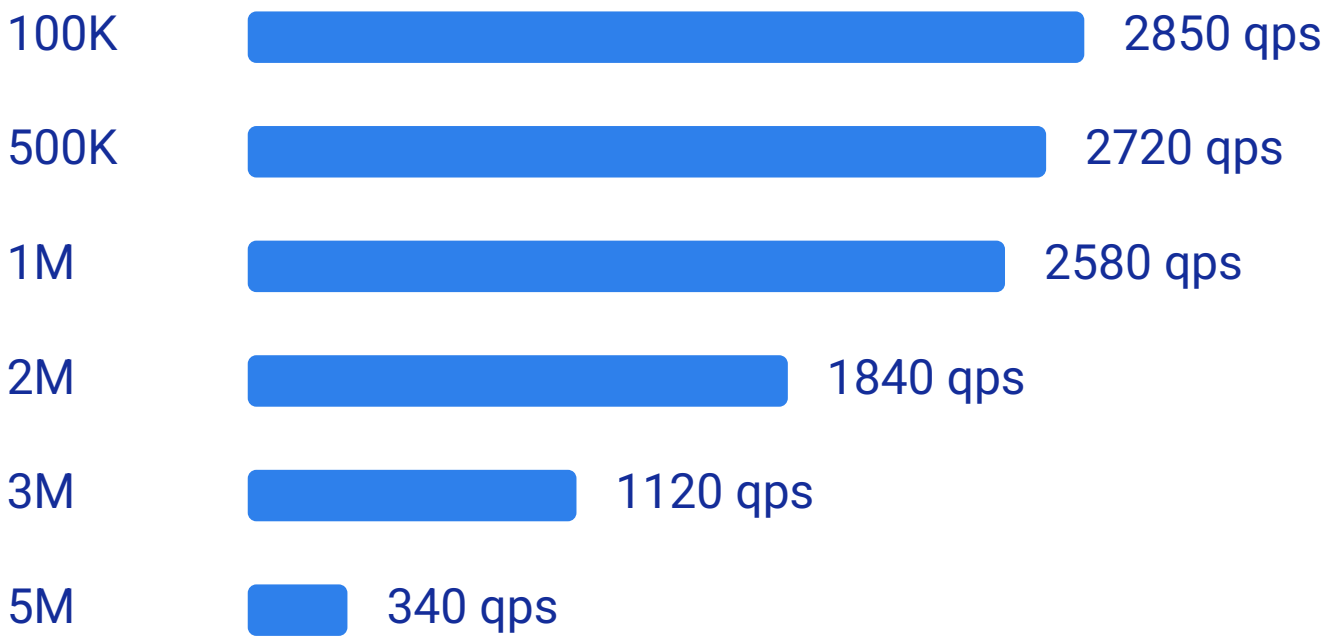
10x

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

PostgresPro

15 мин

от установки расширения
до работающего RAG-прототипа

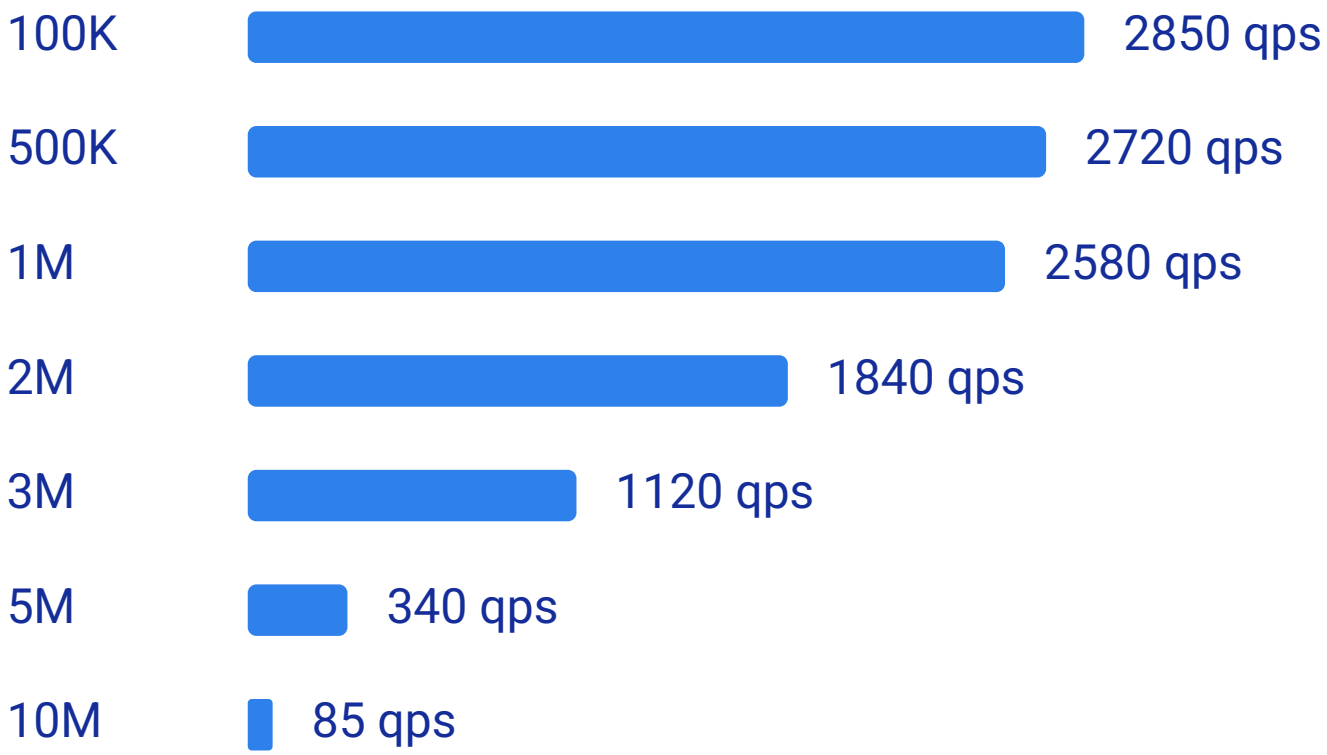
10x

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)

PostgresPro

Что именно должно быть быстрым?

Что ломается первым?

PostgresPro

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?

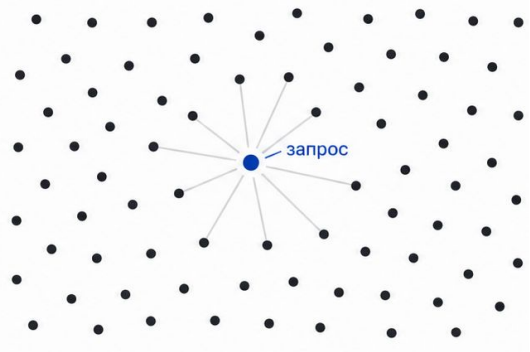
PostgresPro

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·D). На миллионах объектов это слишком дорого.

PostgresPro

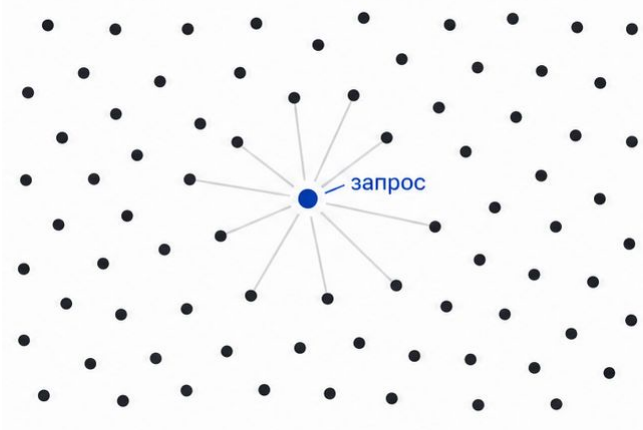
Так может искать иначе?

ANN или KNN

PostgresPro

KNN (k ближайших соседей):

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

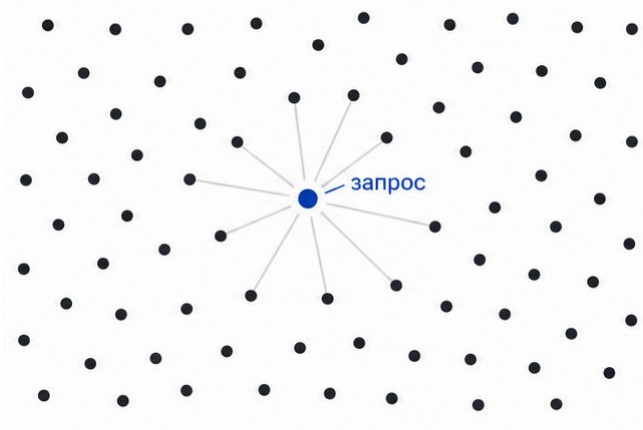


ANN или KNN

PostgresPro

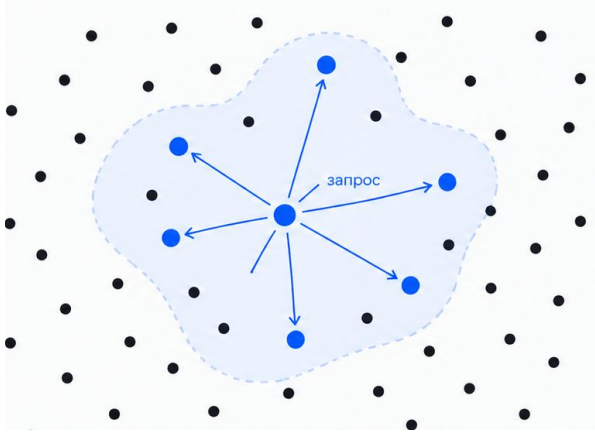
KNN (k ближайших соседей):

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



ANN (примерно ближайшие соседи):

- приблизительный поиск
- смотрим только ближайшие к запросу
- recall <= 100%
- быстро



Что ломается первым?

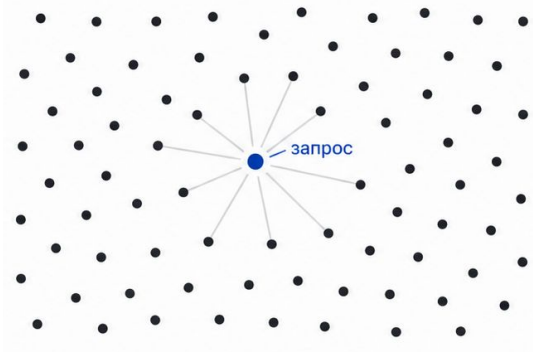
PostgresPro

Запрос

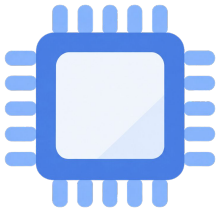
```
SELECT
    id,
    title,
    platform
FROM movies
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```

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

Перебор



pgvector

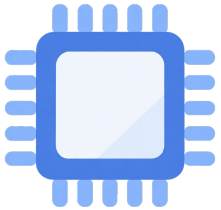


Точный поиск требует сравнить запрос с каждым вектором: O(N·D). На миллионах объектов это слишком дорого.

pgvector

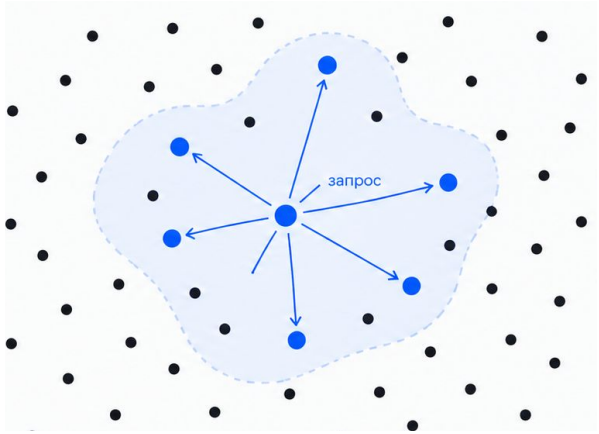
PostgresPro

pgvector



\+ ? =

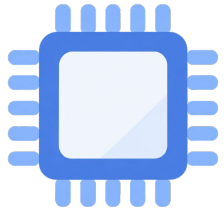
Неполный перебор



pgvector

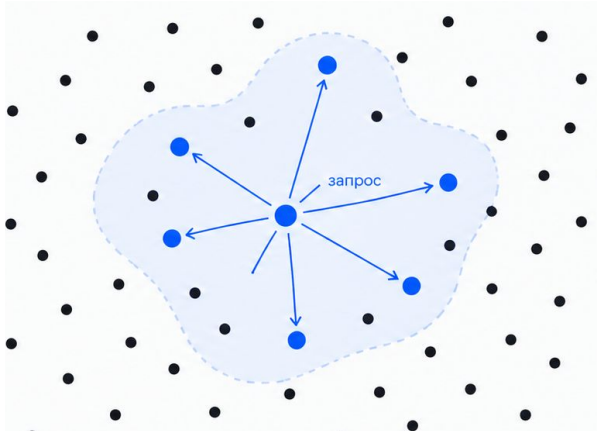
PostgresPro

pgvector



+ индекс =

Неполный перебор



PostgresPro

Что за индексы?

Индексы

PostgresPro

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.

PostgresPro

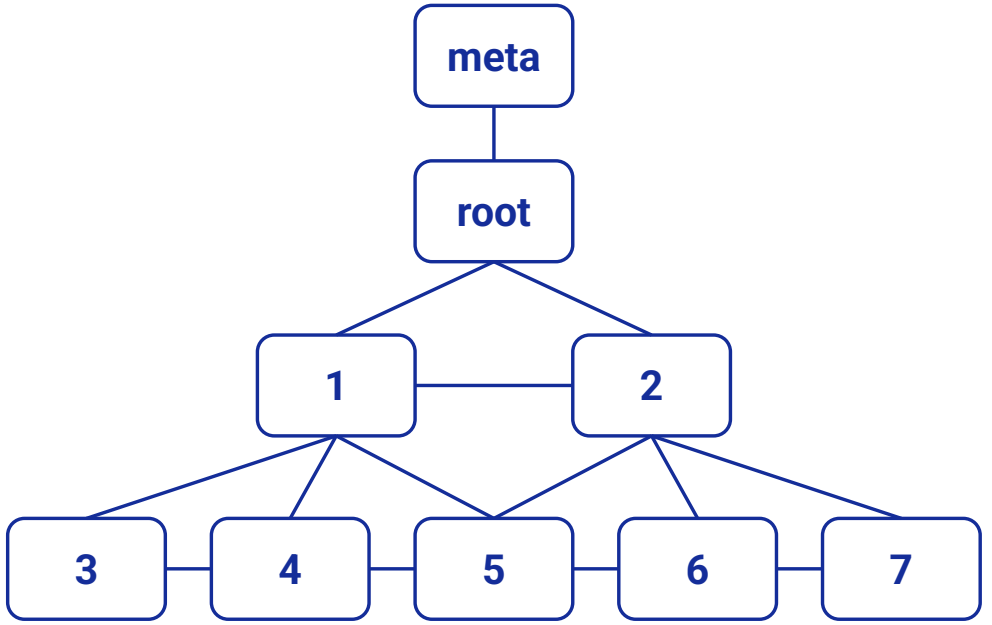
Какой вопрос они решают?

PostgreSQL, индексы…

PostgresPro

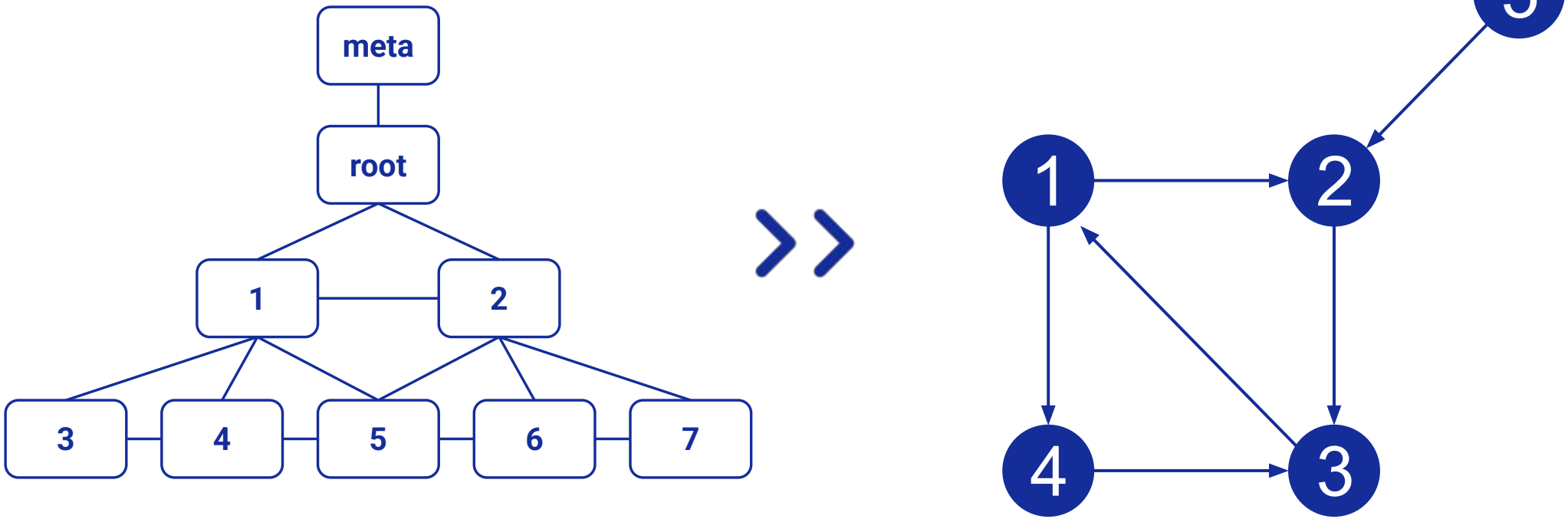
PostgreSQL, индексы…

PostgresPro



PostgreSQL, индексы…

PostgresPro



Почему B-Tree не болит, а векторы страдают?

PostgresPro

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

Почему B-Tree не болит, а векторы страдают?

PostgresPro

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?

PostgresPro

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?

PostgresPro

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?

PostgresPro

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?

PostgresPro

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?

PostgresPro

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)

200 +

ключей B-Tree
на одной странице 8 КБ

Почему B-Tree не болит, а векторы страдают?

PostgresPro

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)

200 +
ключей B-Tree
на одной странице 8 КБ

1
вектор на одной
странице 8 КБ

Почему B-Tree не болит, а векторы страдают?

PostgresPro

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)

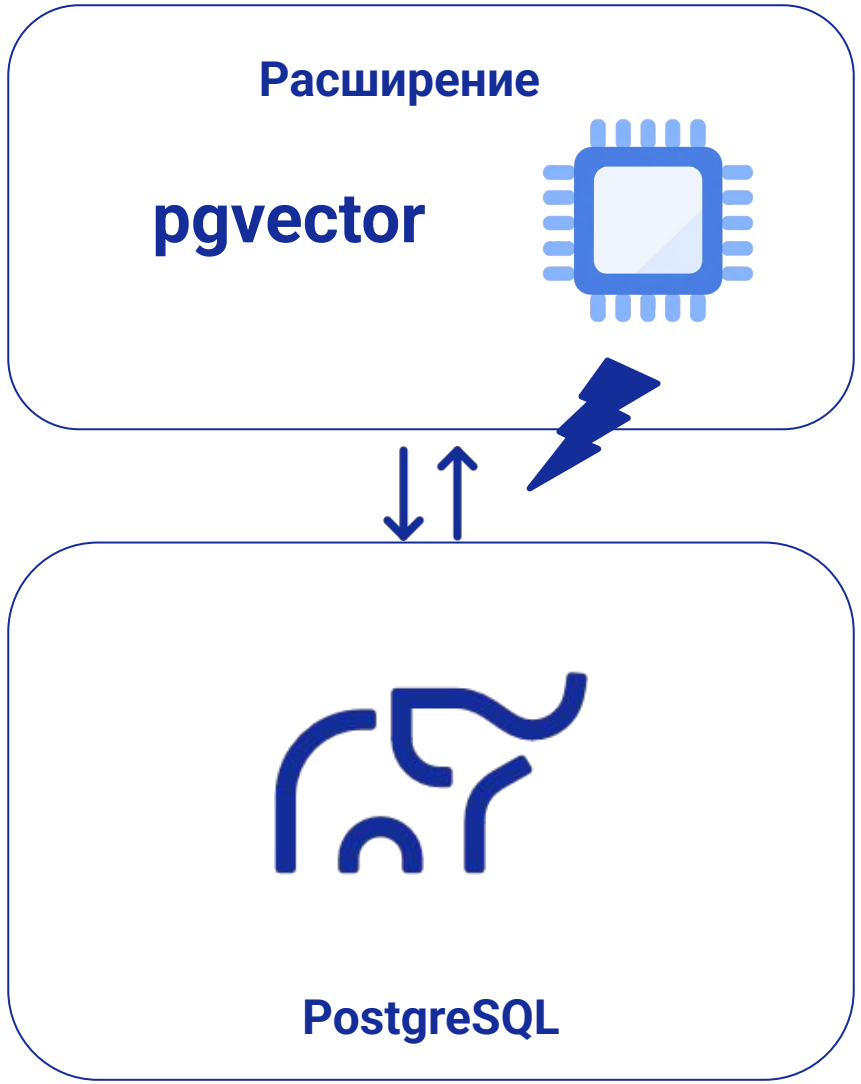
200 +
ключей B-Tree
на одной странице 8 КБ

1
вектор на одной
странице 8 КБ

96
кэш-линий CPU нужно
прочитать на 1 вектор

Интеграция

PostgresPro



PostgresPro

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

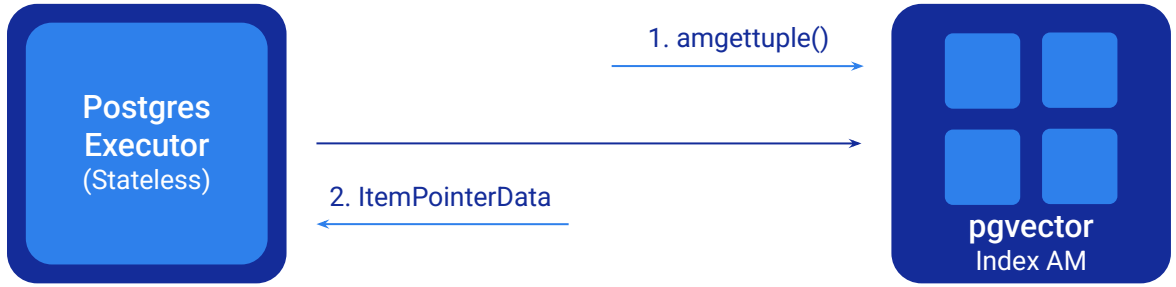
2. Исполнение: Модель **Volcano**

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: executor «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 зацикливания при обходе связей

Как Postgres «дружит» с векторами

Внутренний контекст сканирования

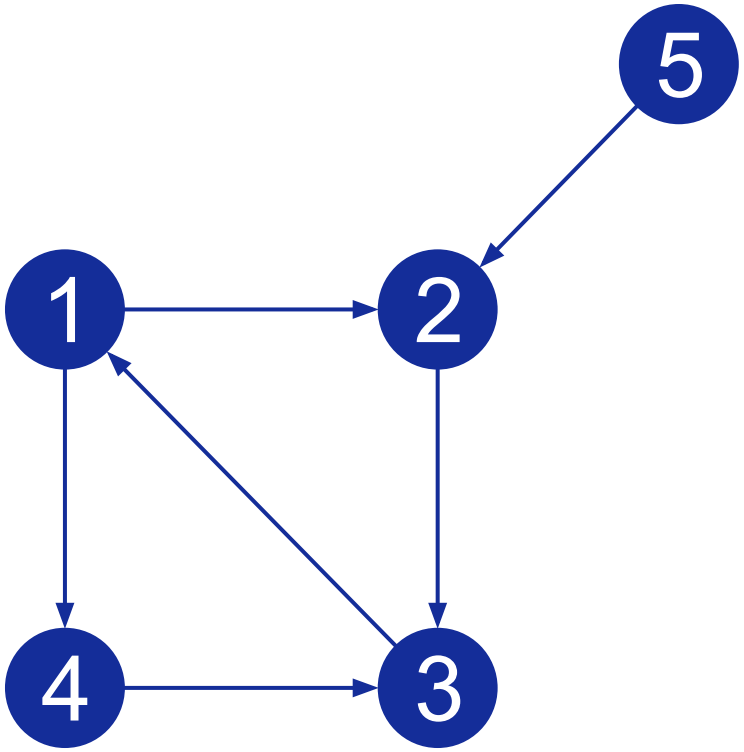
Каждый раз pgvector тащит за собой в памяти:

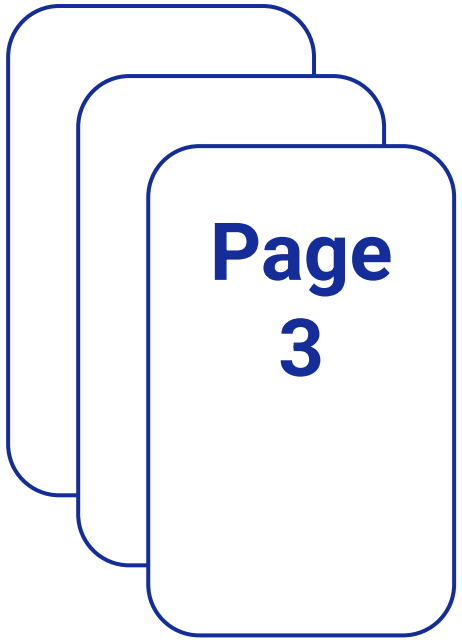
1. Pairing Heap − 2 приоритетных очереди
    - **C** − кандидаты для дальнейшего исследования (min-heap по расстоянию)
    - **W** − уже найденные ближайшие (max-heap по расстоянию)

2. Visited Hash Table − защита от зацикливания при обходе связей

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

Хранение





Хранение: вектор в мире 8-килобайтных страниц

PostgresPro

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

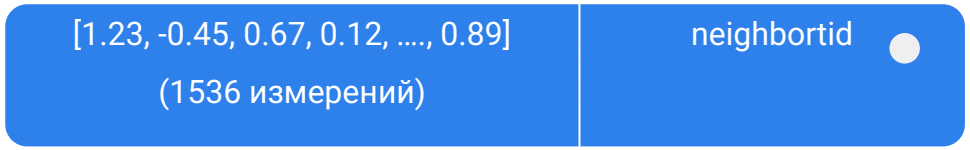
PostgresPro

Page 0 (Метастраница)

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

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

Вектор (ID: 42)



Page 1 (Элементный кортеж)

Хранение: вектор в мире 8-килобайтных страниц

PostgresPro

Page 0 (Метастраница)

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

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

Page 1 (Элементный кортеж)

Вектор (ID: 42)

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

Page 2 (Кортежи соседей)

Кортеж соседей (ID: 999):

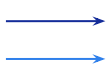
9 | 18 | 33 | 56 | 82 | …

Кортеж соседей (ID: 1001):

7 | 15 | 23 | 67 | 88 | …

Кортеж соседей (ID: 1002):

3 | 8 | 42 | 77 | 91 | …



Ссылка на первый кортеж соседей для данного вектора

Ссылка на векторы (элементные кортежи) по их ID (TID)

ID 42 → вектор [1.23, -0.45, 0.67, 0.12, …, 0.89] (1536 измерений)
ID 7 → вектор [-0.11, 0.98, 0.33, -0.21, …, 0.77] (1536 измерений)
ID 15 → вектор [0.44, -0.22, -0.91, 0.05, …, -0.36] (1536 измерений)

Хранение: вектор в мире 8-килобайтных страниц

PostgresPro

«Интерстеллар» - запрос

Хранение: вектор в мире 8-килобайтных страниц

PostgresPro

«Интерстеллар» - запрос

1

Страница A (Элемент)

Текущий узел: **«Дюна»**

→ Указатель neighbortid -> Page B

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

↓ Disc Read #1

«Где соседи?»

Хранение: вектор в мире 8-килобайтных страниц

PostgresPro

«Интерстеллар» - запрос

1

Страница A (Элемент)

Текущий узел: **«Дюна»**

→ Указатель neighbortid -> Page B

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

↓ Disc Read #1

«Где соседи?»

2

Страница B (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

PostgresPro

«Интерстеллар» - запрос

1

Страница A (Элемент)

Текущий узел: **«Дюна»**

→ Указатель neighbortid -> Page B

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

↓ Disc Read #1

«Где соседи?»

2

Страница B (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 C (Элемент)

Текущий узел: **«Прибытие»**

→ указатель neighbortid -> Page F

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

↓ Disc Read #3

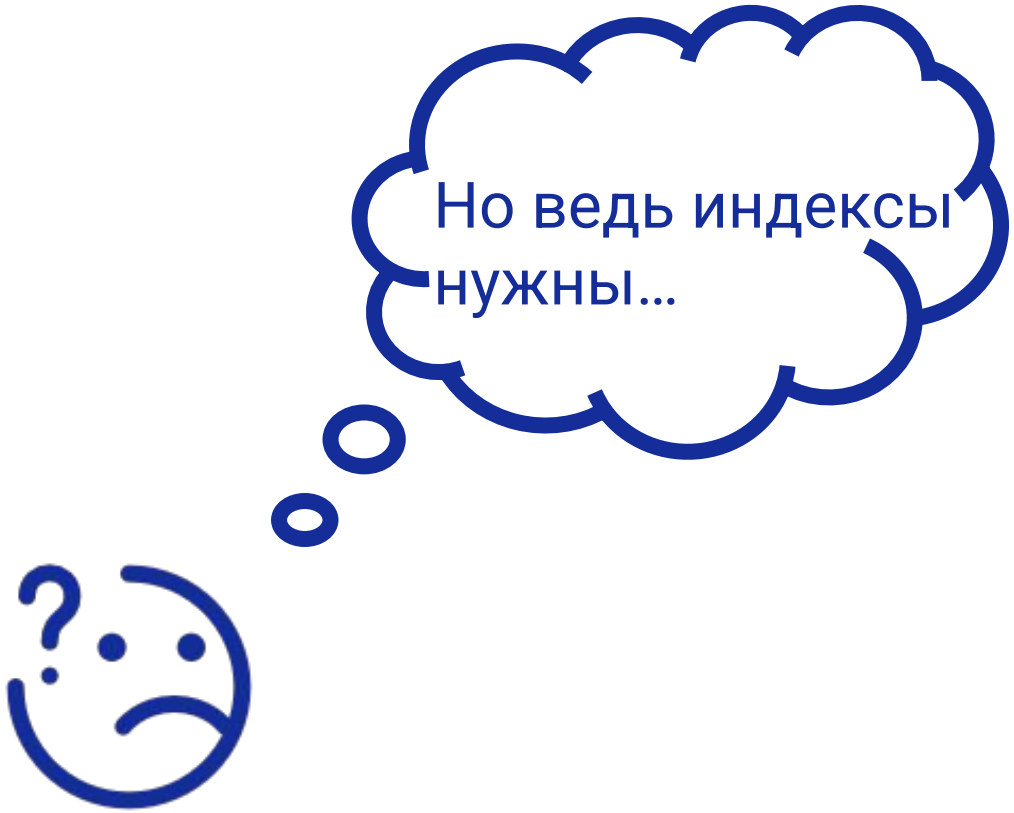
Повторить цикл от Page C - **«Прибытие»**

1 шаг графа = 3 I/O

100 шагов поиска = 300 I/O

Хранение: вектор в мире 8-килобайтных страниц

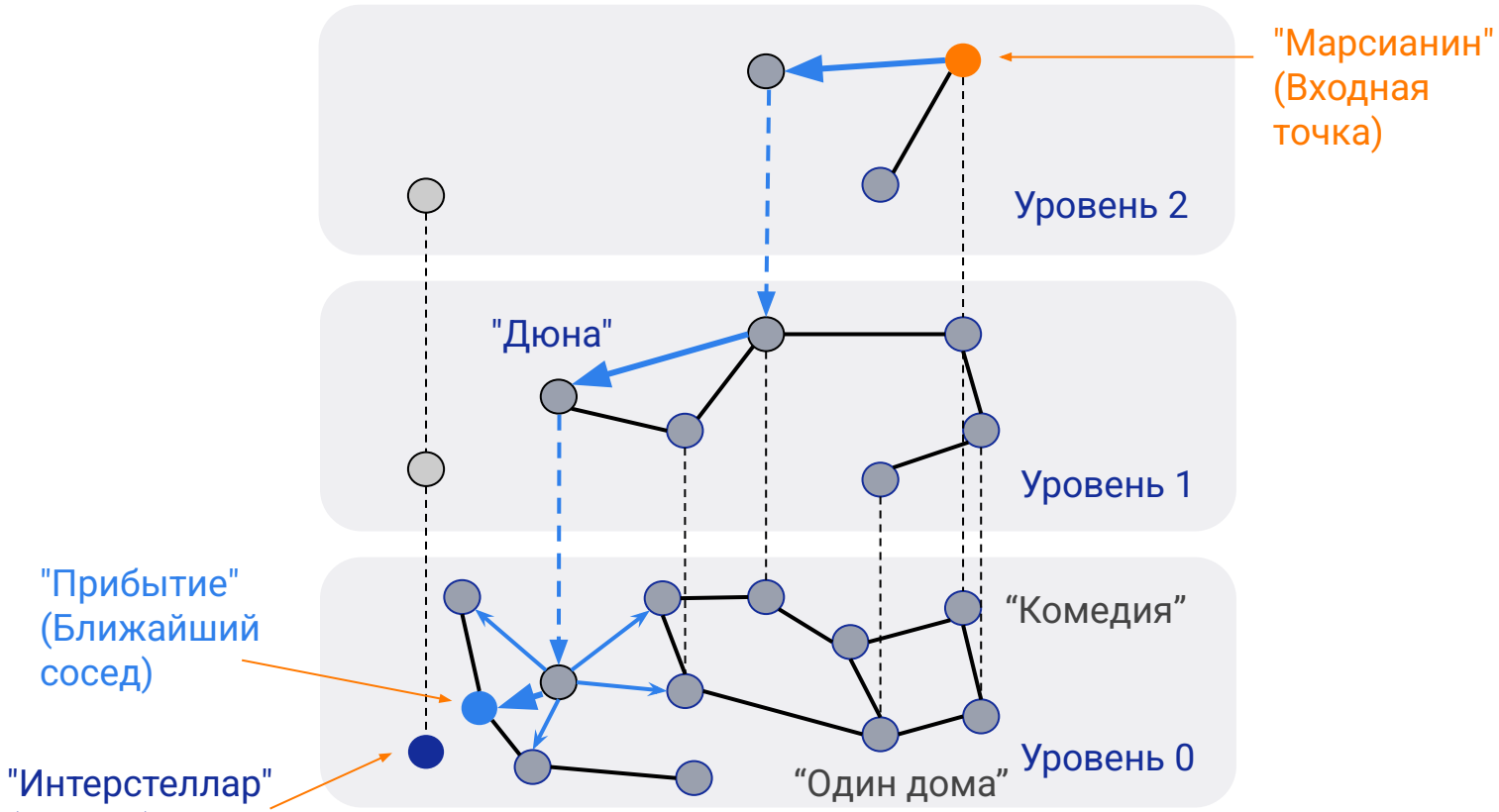
PostgresPro



PostgresPro

HNSW: анатомия графа и погоня за указателями

HNSW (иерархический маленький мир)

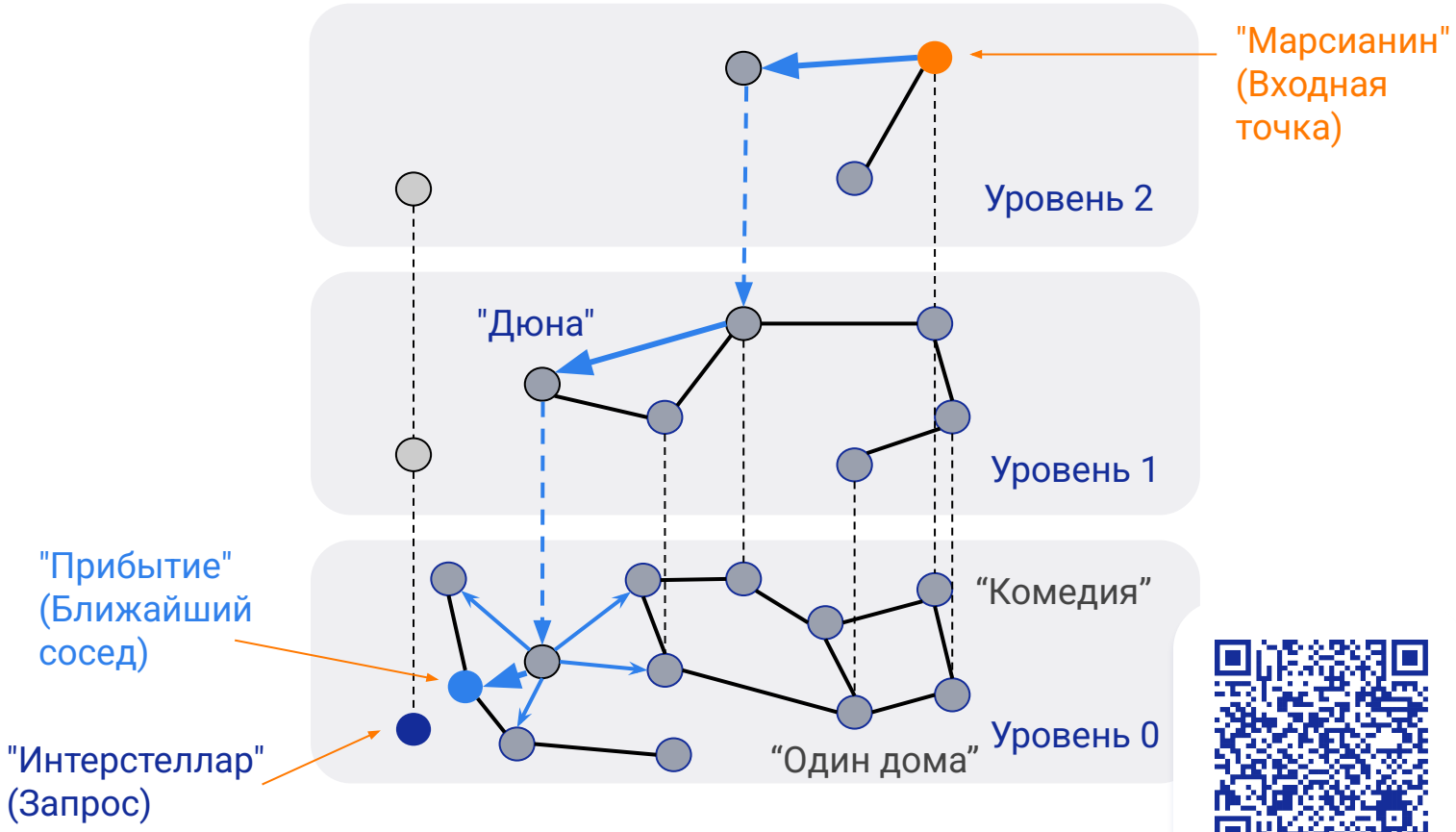


HNSW (иерархический маленький мир)

PostgresPro

Параметры индекса:

- M
- ef_construction



HNSW (иерархический маленький мир)

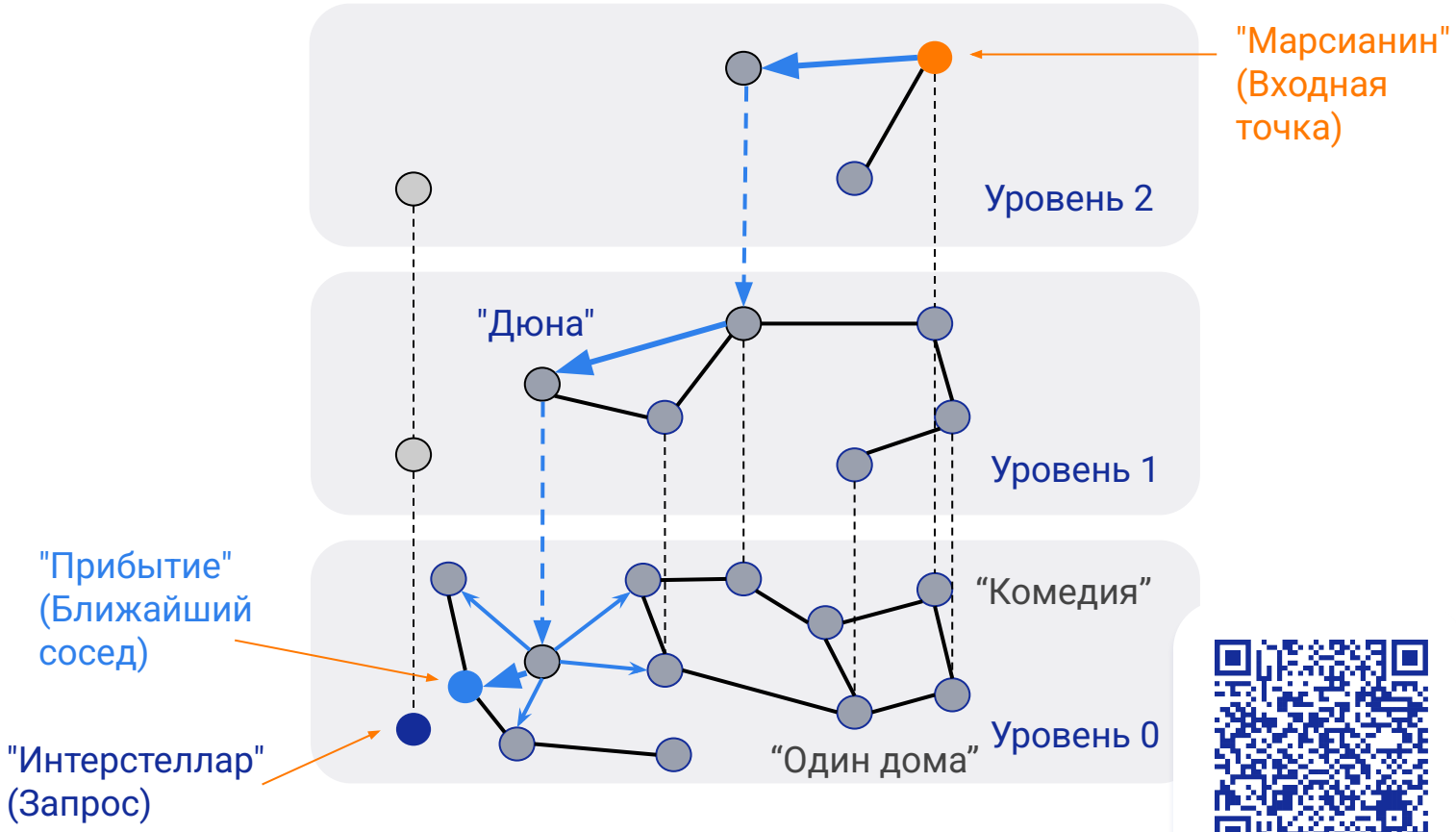
PostgresPro

Параметры индекса:

- M
- ef_construction

Параметры поиска:

- ef_search



Структура HnswElementData

PostgresPro

Распределение памяти индекса (5M векторов)

Структура HnswElementData

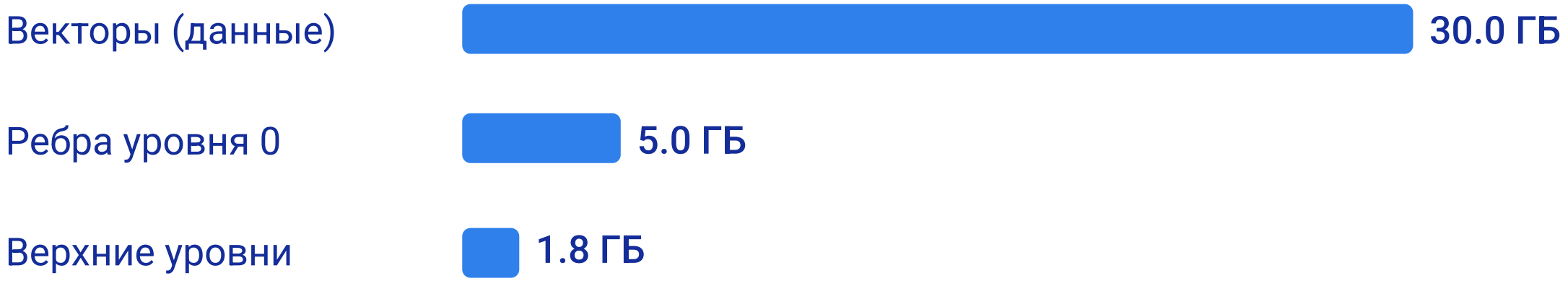
PostgresPro

Распределение памяти индекса (5M векторов)

Структура HnswElementData

PostgresPro

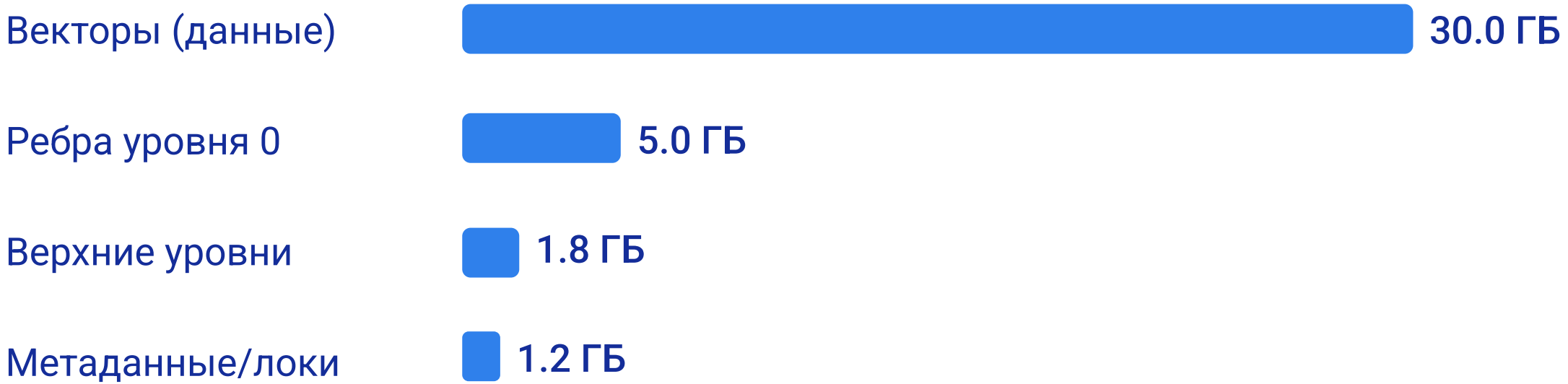
Распределение памяти индекса (5M векторов)



Структура HnswElementData

PostgresPro

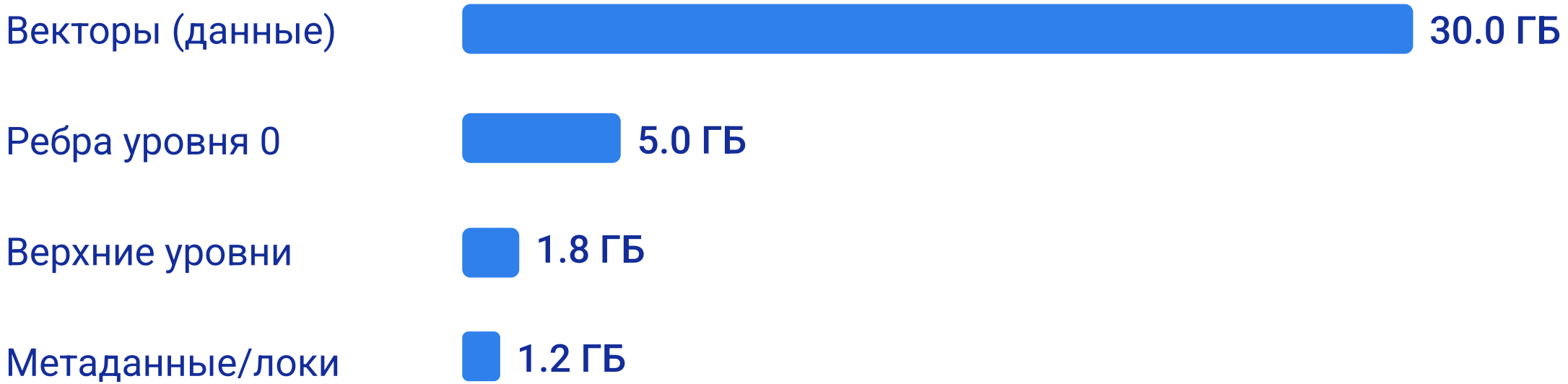
Распределение памяти индекса (5M векторов)



Структура HnswElementData

PostgresPro

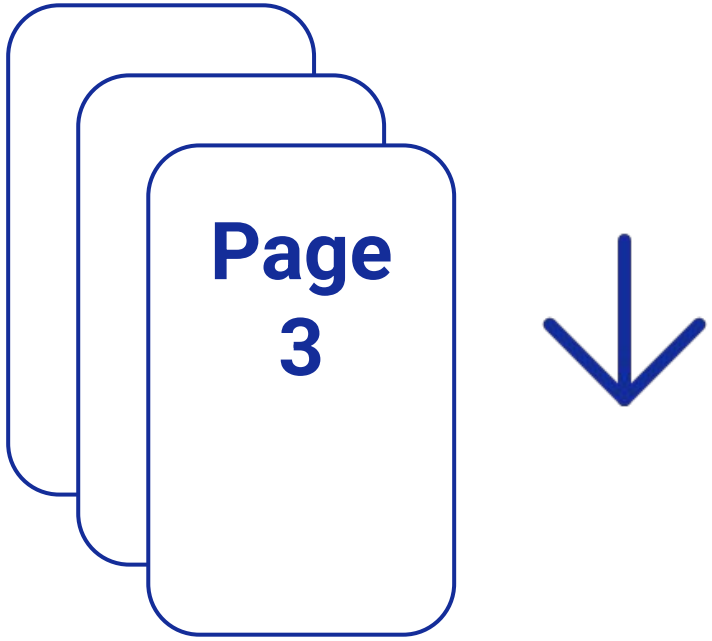
Распределение памяти индекса (5M векторов)



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

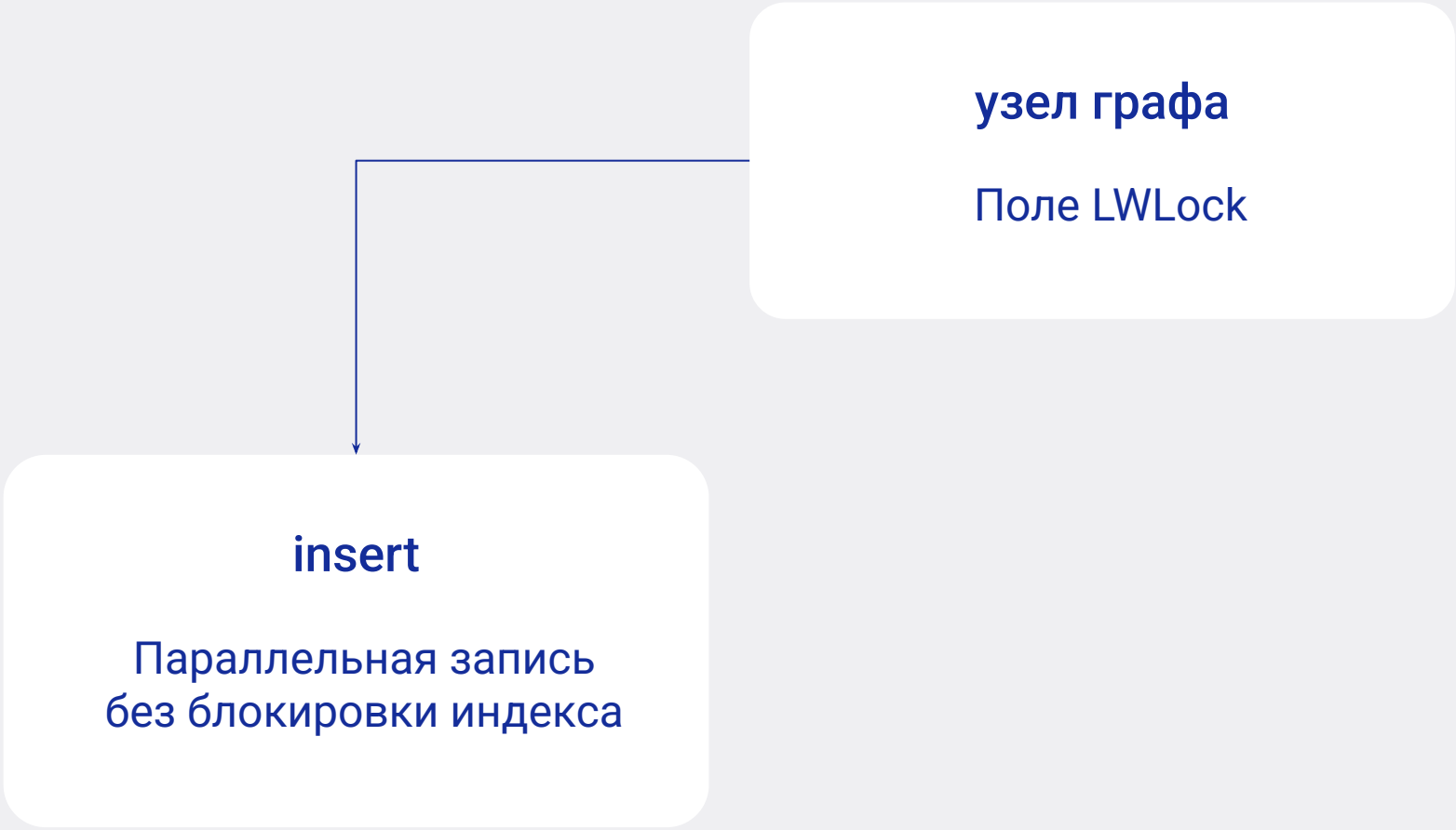
Структура HnswElementData

PostgresPro



Структура HnswElementData

PostgresPro



Структура HnswElementData

PostgresPro

узел графа

Поле LWLock

insert

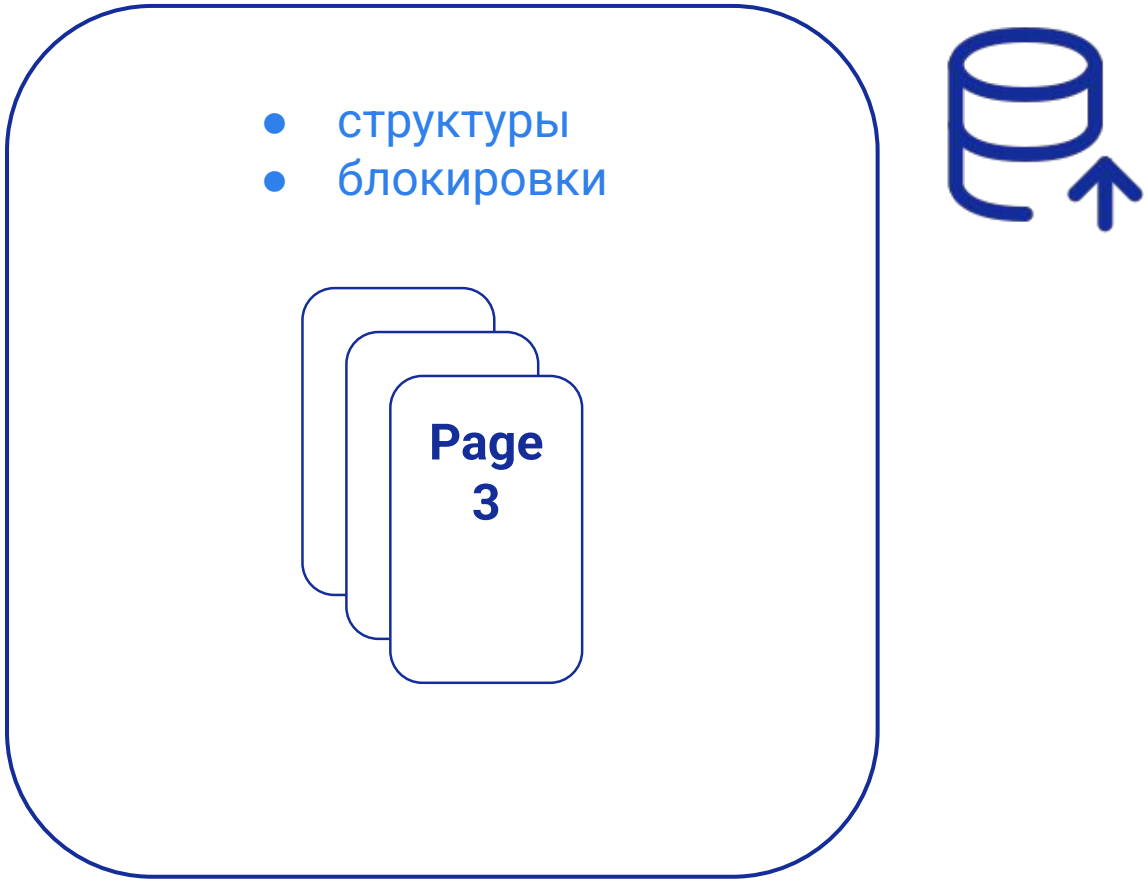
Параллельная запись
без блокировки индекса

search

Атомарные проверки
на каждом узле графа

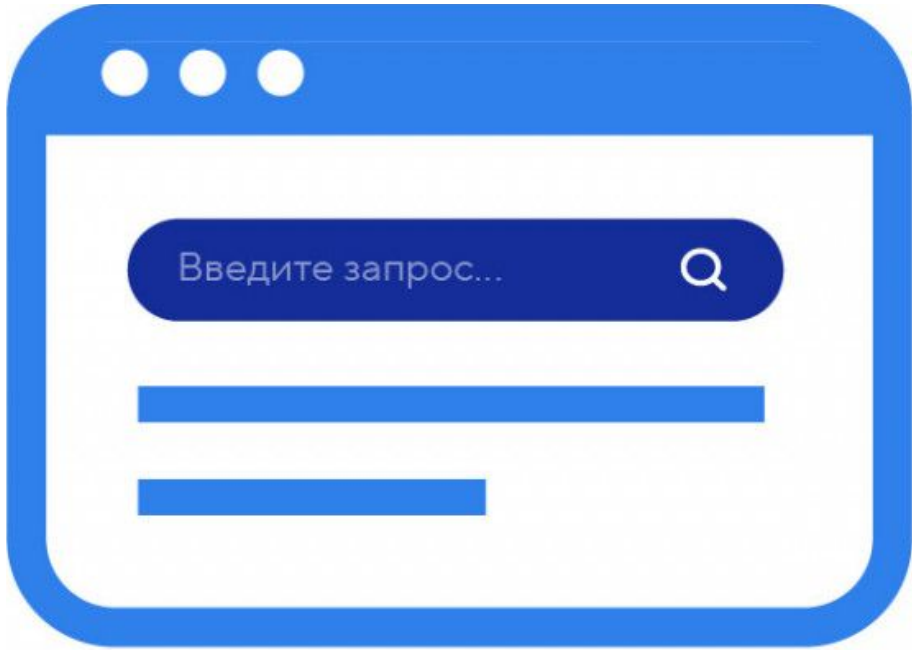
Структура HnswElementData

PostgresPro



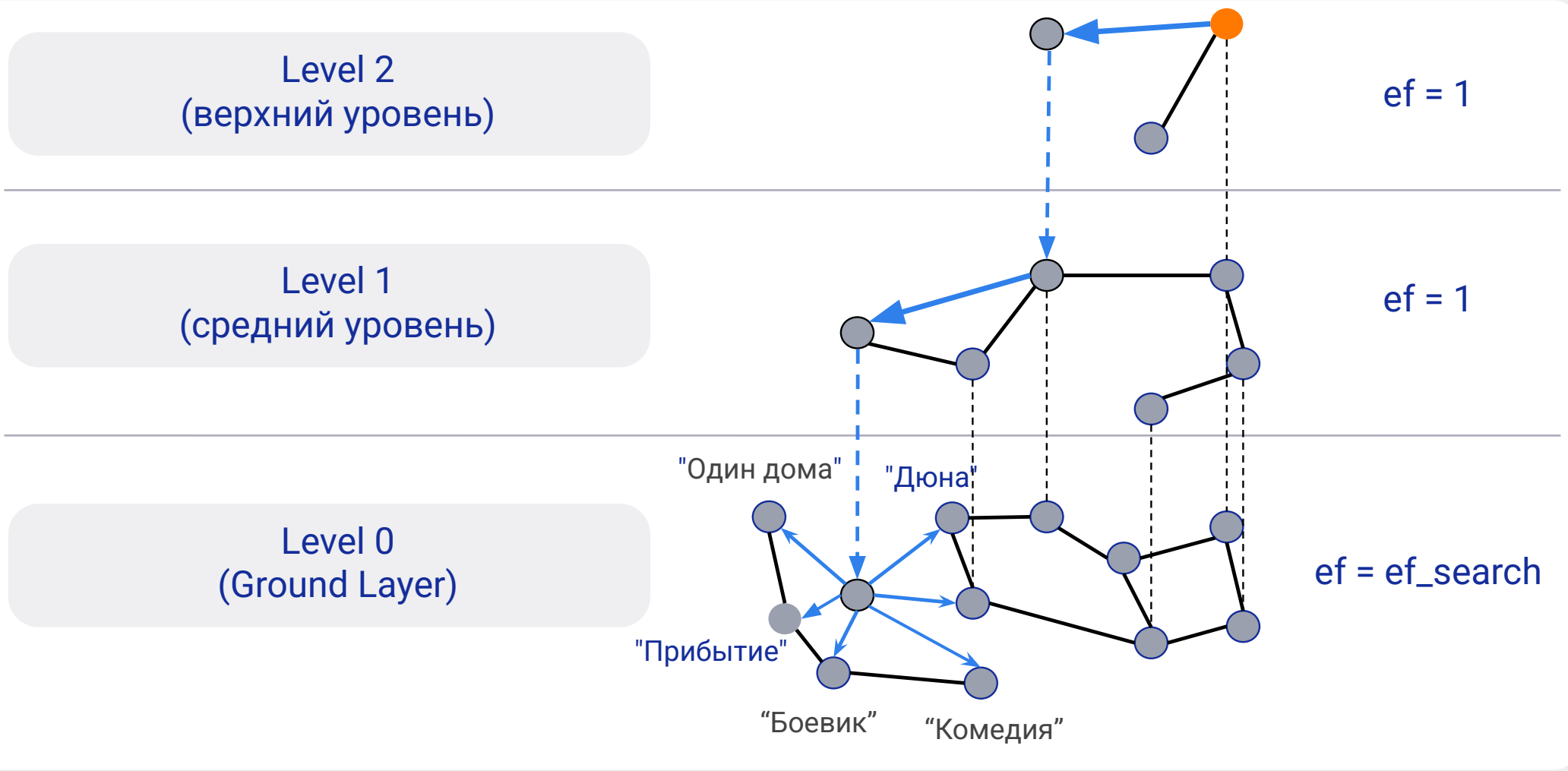
Ради чего все это?

PostgresPro



HnswSearchLayer:
Цикл, который убивает кэш

PostgresPro



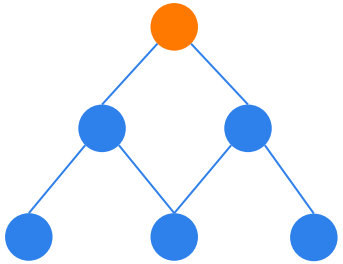
HnswSearchLayer: Цикл, который убивает кэш

PostgresPro

1

Remove first
priority queue (heap)

Heap
(priority queue)



HnswSearchLayer: Цикл, который убивает кэш

PostgresPro

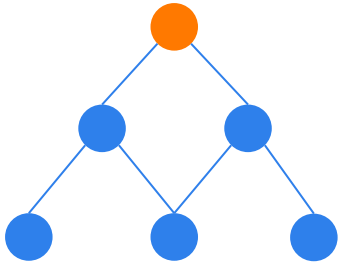
1

Remove first
priority queue (heap)

2

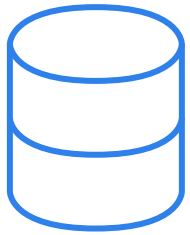
Load Element
Buffer Manager (I/O)

Heap
(priority queue)



→

Buffer Manager



→

HnswSearchLayer: Цикл, который убивает кэш

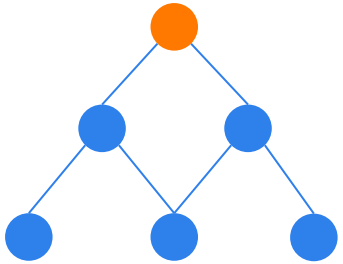
PostgresPro

1 **Remove first**
priority queue (heap)

2 **Load Element**
Buffer Manager (I/O)

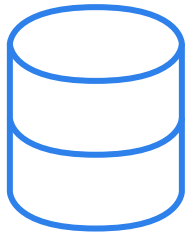
3 **dist_func**
расчет расстояния

Heap
(priority queue)



→

Buffer Manager



→

dist_func

→

HnswSearchLayer: Цикл, который убивает кэш

PostgresPro

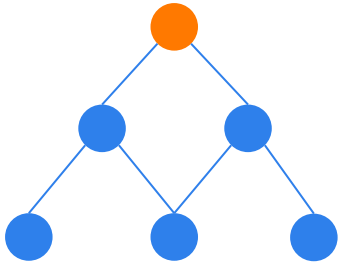
1 **Remove first**
priority queue (heap)

2 **Load Element**
Buffer Manager (I/O)

3 **dist_func**
расчет расстояния

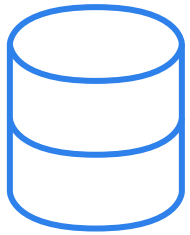
4 **Loop neighbors**
повторяем 1-3

Heap
(priority queue)



→

Buffer Manager

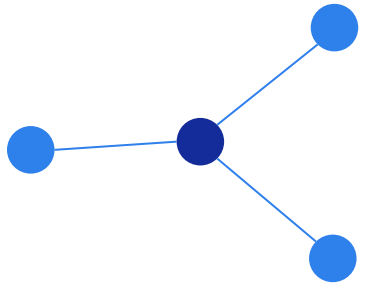


→

dist_func

→

Neighbors
(adjacency list)



HnswSearchLayer: Цикл, который убивает кэш

PostgresPro

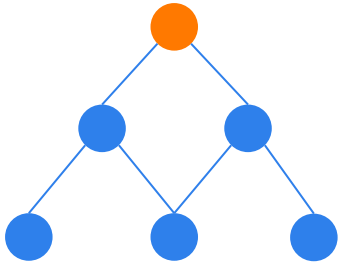
1 **Remove first**
priority queue (heap)

2 **Load Element**
Buffer Manager (I/O)

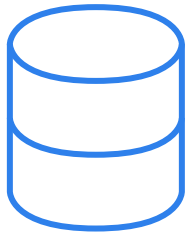
3 **dist_func**
расчет расстояния

4 **Loop neighbors**
повторяем 1-3

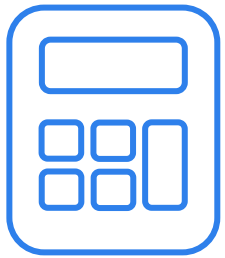
Heap
(priority queue)



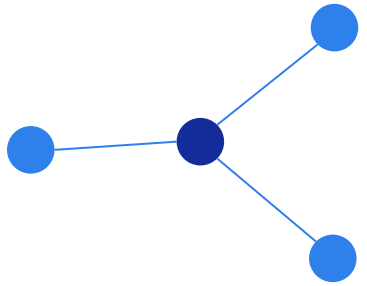
Buffer Manager



dist_func



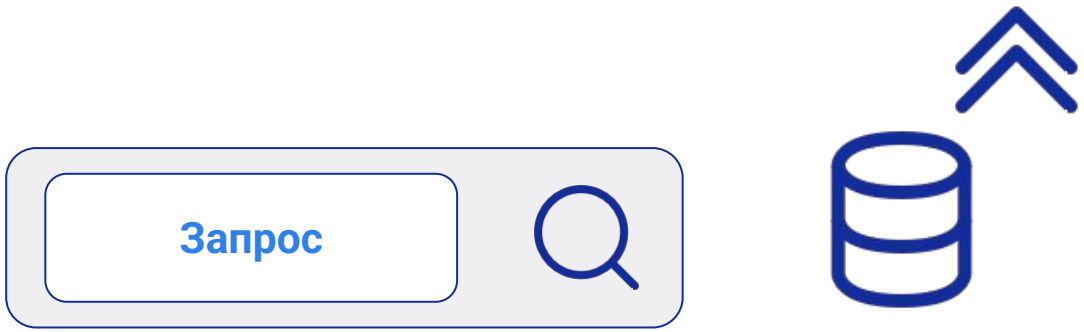
Neighbors
(adjacency list)



Pop closest

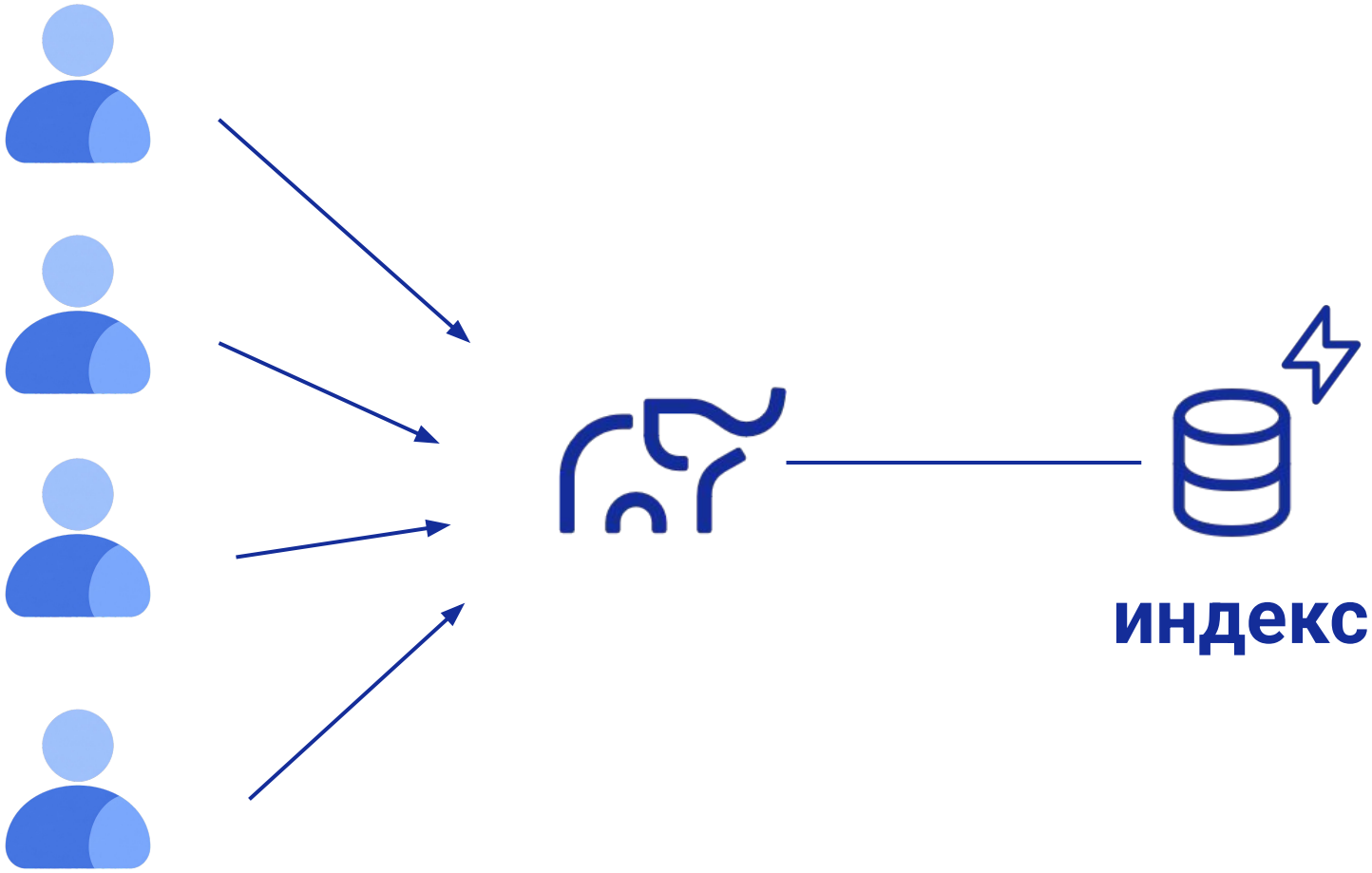
Это был только 1 запрос...

PostgresPro



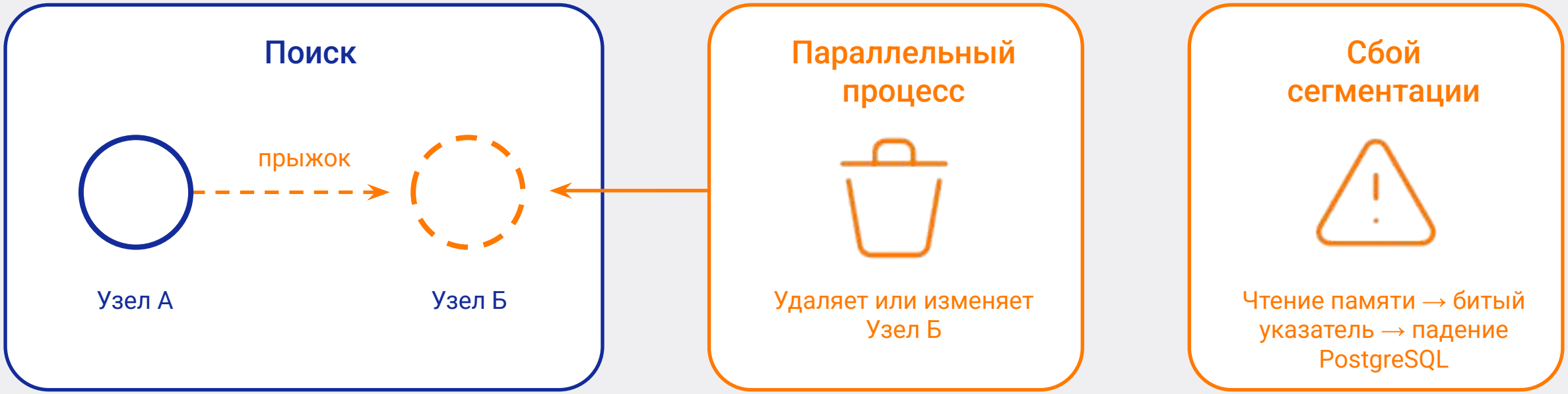
А если запросов много?

PostgresPro



MVCC и Lock: как выжить в аду

PostgresPro



MVCC и Lock: как выжить в аду

PostgresPro

1

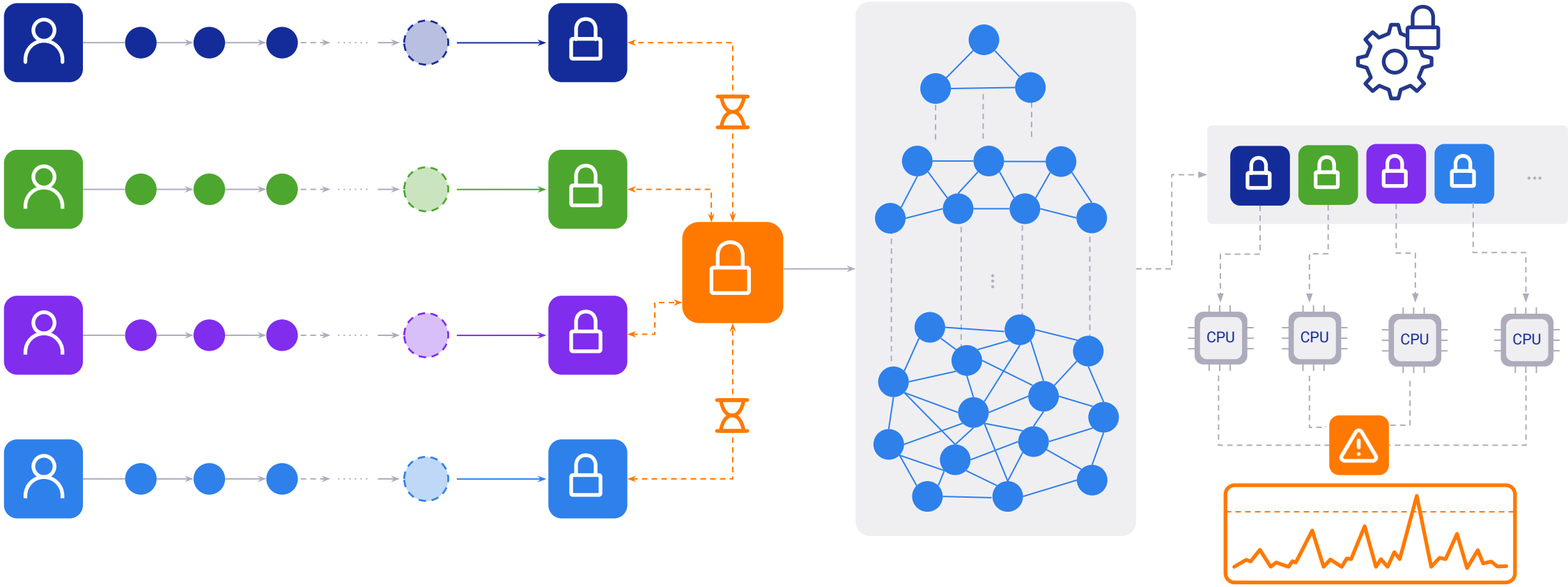
MVCC снимки

MVCC — разрешены
не-MVCC — запрещены

pgvector не держит
пины буферов

MVCC и Lock: как выжить в аду

PostgresPro



MVCC и Lock: как выжить в аду

PostgresPro

1

MVCC снимки

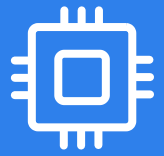
MVCC — разрешены
не-MVCC — запрещены

pgvector не держит
пины буферов

2

ShareLock

Lock страниц индекса
в режиме SHARE



Каждый шаг поиска
= системный вызов

MVCC и Lock: как выжить в аду

PostgresPro

	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

MVCC и Lock: как выжить в аду

PostgresPro

	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
```

MVCC и Lock: как выжить в аду

PostgresPro

1

MVCC снимки

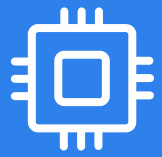
MVCC — разрешены
не-MVCC — запрещены

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

MVCC и Lock: как выжить в аду

PostgresPro

1

MVCC снимки

MVCC — разрешены
не-MVCC — запрещены

Поиск всегда видит целостные данные, даже при параллельных изменениях

2

ShareLock

Lock страниц индекса
в режиме SHARE

Защита от конкурентных изменений страниц

3

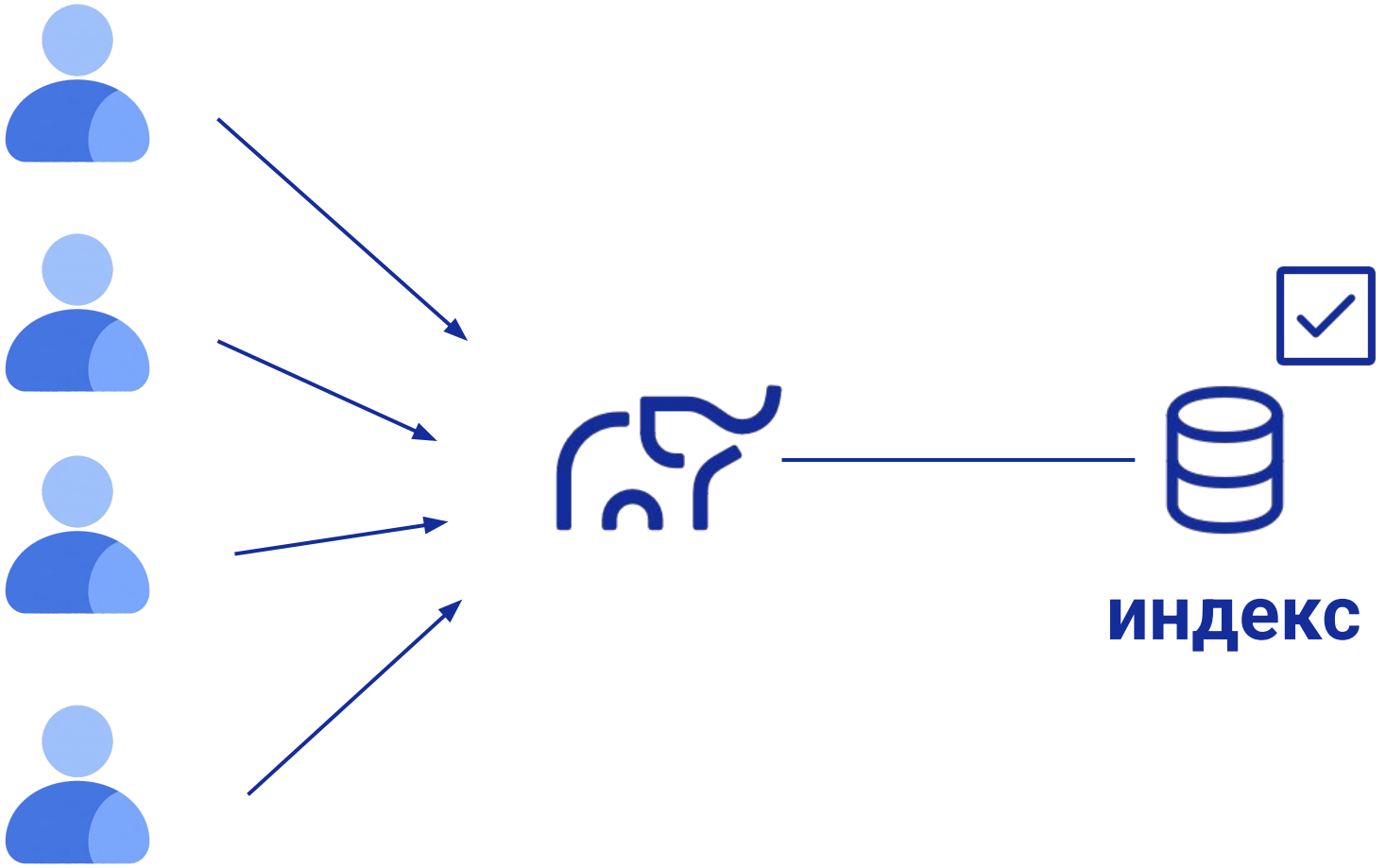
Сброс контекста

На каждом
возобновлении

Защита от утечек и переполнения памяти

А если запросов много?

PostgresPro



PostgresPro

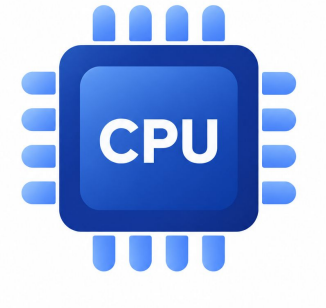
И все же, что с графами?

Три ~~кита~~ слона

PostgresPro

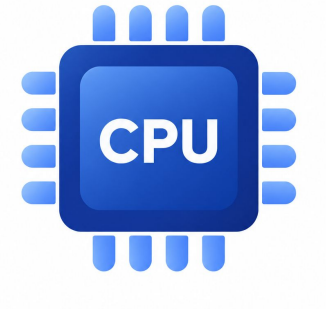
Три ~~кита~~ слона

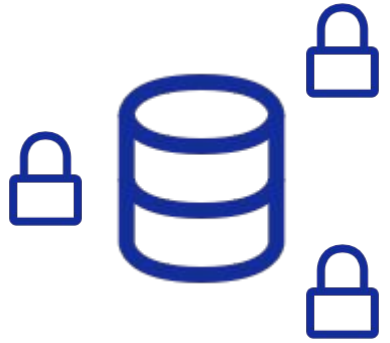
PostgresPro



Три ~~кита~~ слона

PostgresPro



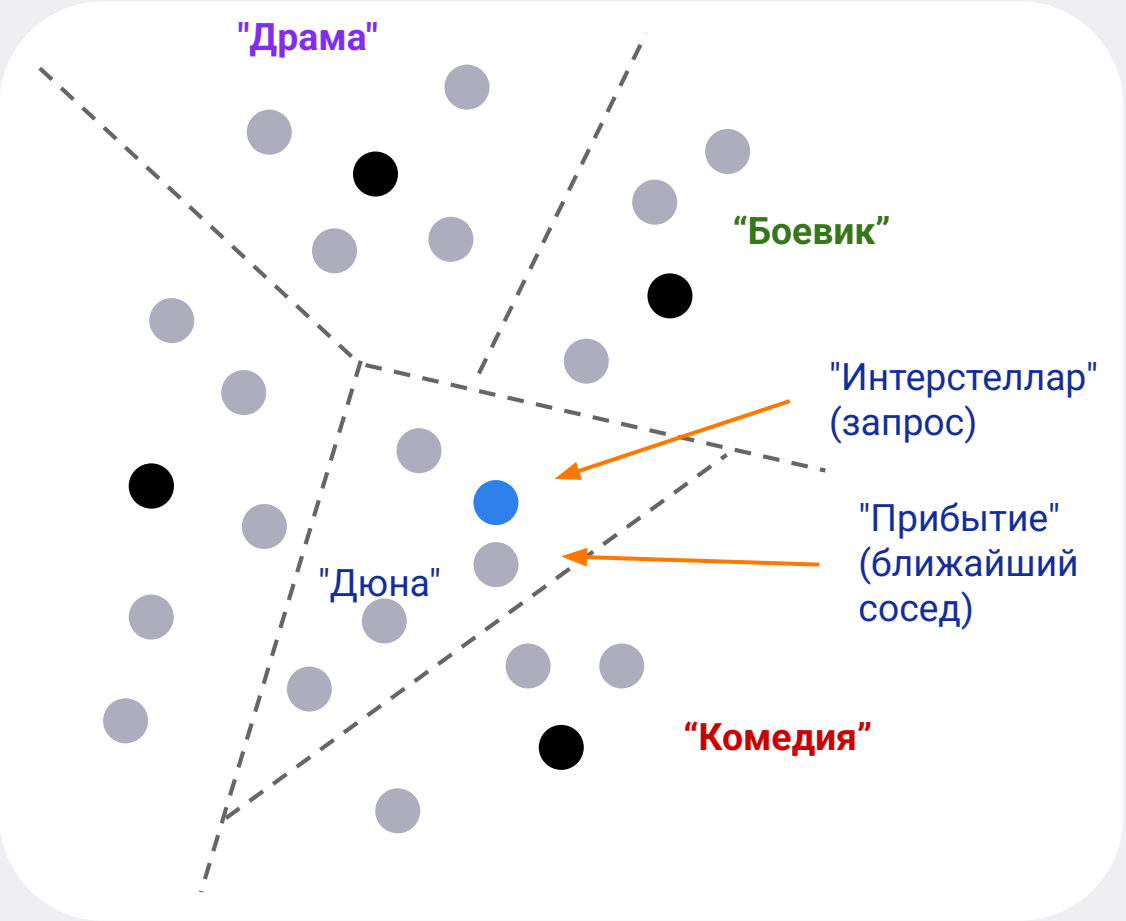


PostgresPro

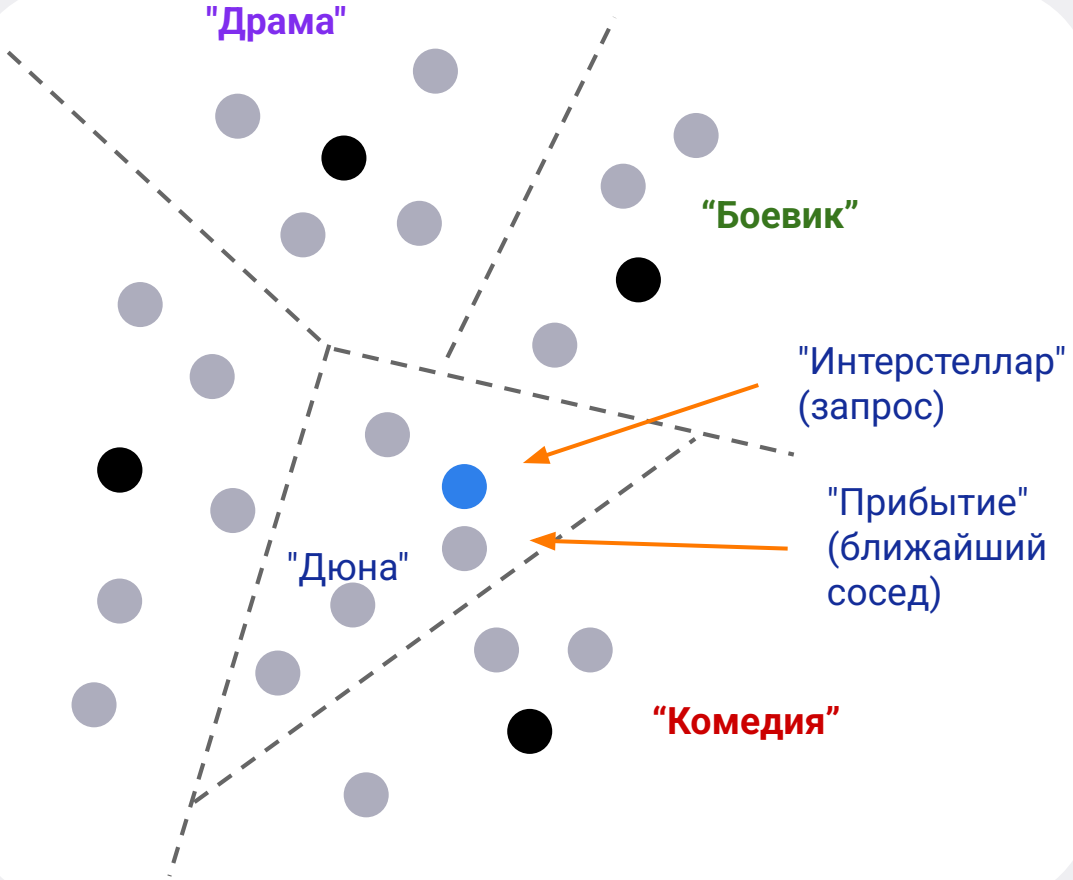
IVFFlat: стриминг против прыжков

IVFFlat
(Инвертированный файл с плоским сжатием)

PostgresPro



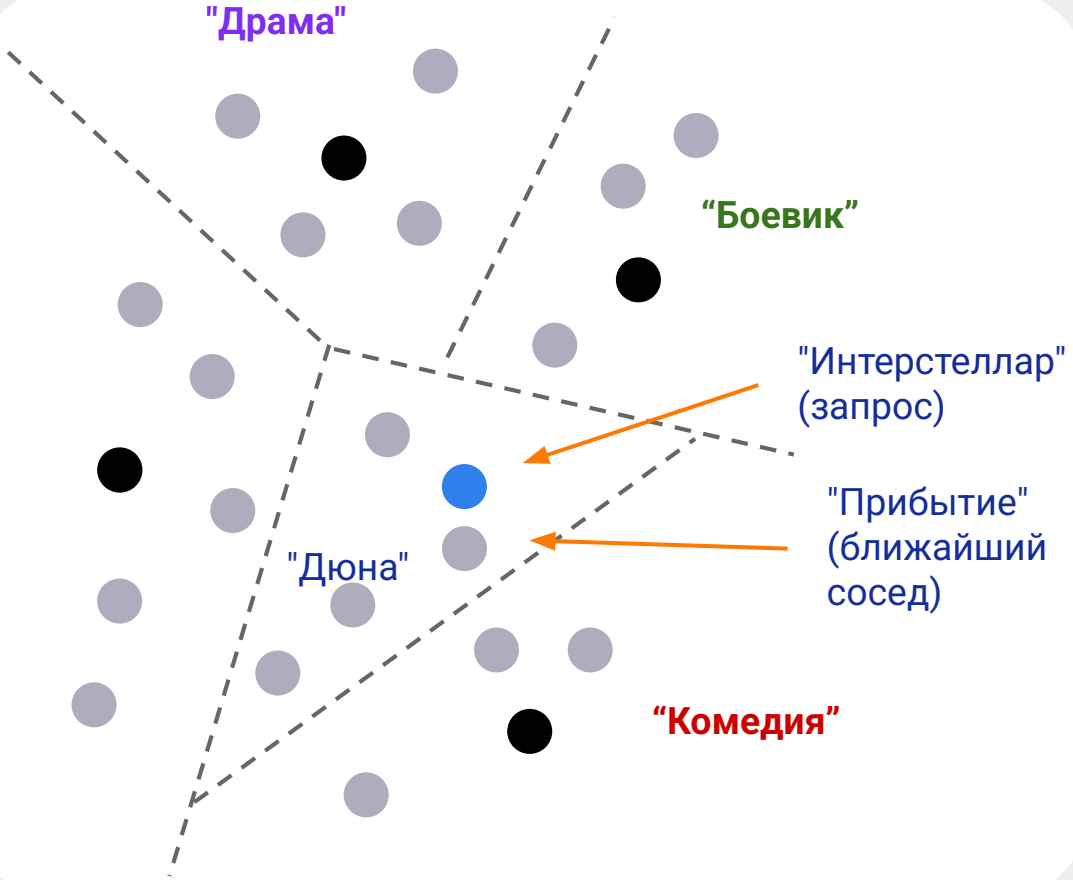
IVFFlat (Инвертированный файл с плоским сжатием)



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 ↑ probes → ↑ recall, ↑ время

Структура IVFFlat

PostgresPro

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)

Структура IVFFlat

PostgresPro

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)

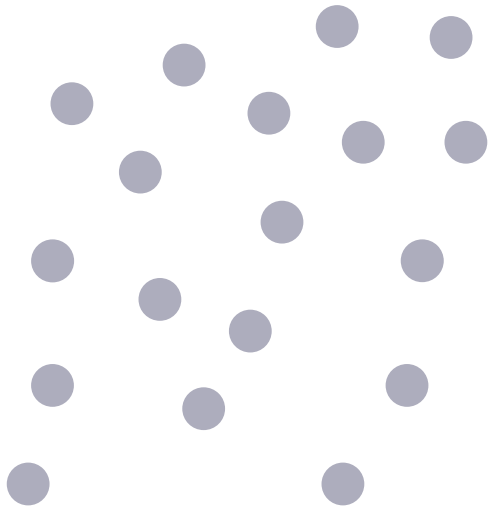
[0.46, 0.21, -0.97, 1.12, …., 0.98]

(1536 измерений)

IVFFlat: когда простота — это SeqScan

PostgresPro

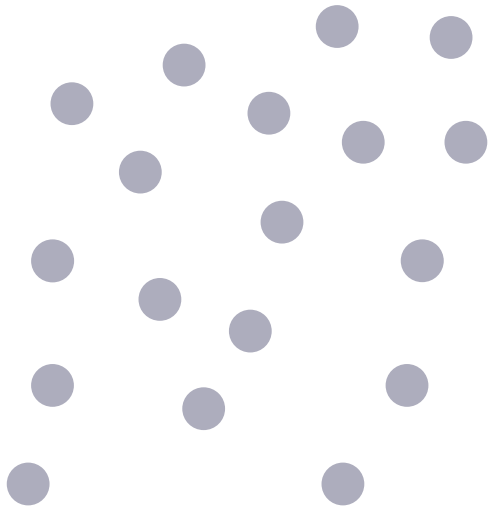
Исходные
вектора



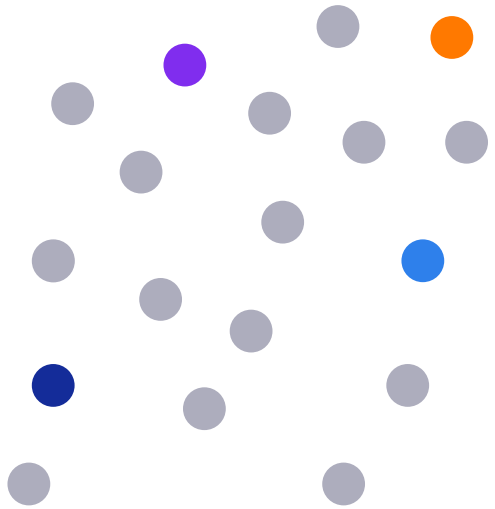
IVFFlat: когда простота — это SeqScan

PostgresPro

Исходные вектора



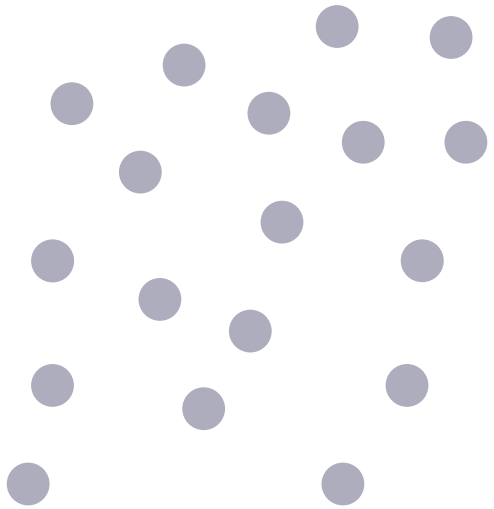
Выбор k центроидов



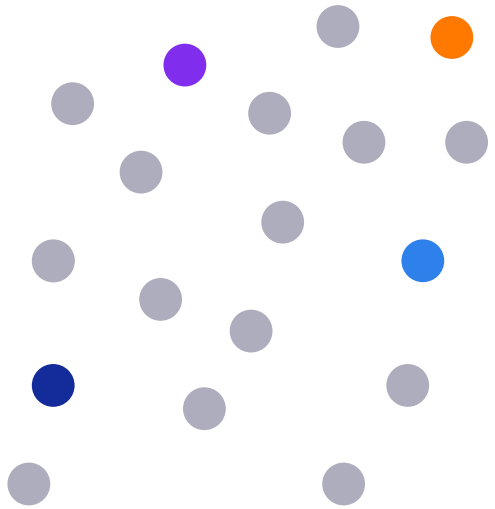
IVFFlat: когда простота — это SeqScan

PostgresPro

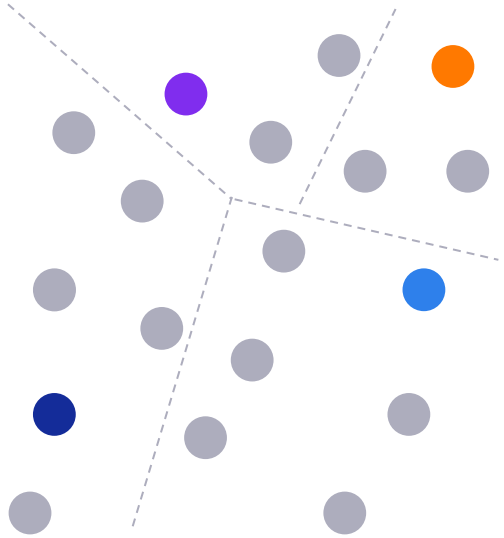
Исходные вектора



Выбор k центроидов



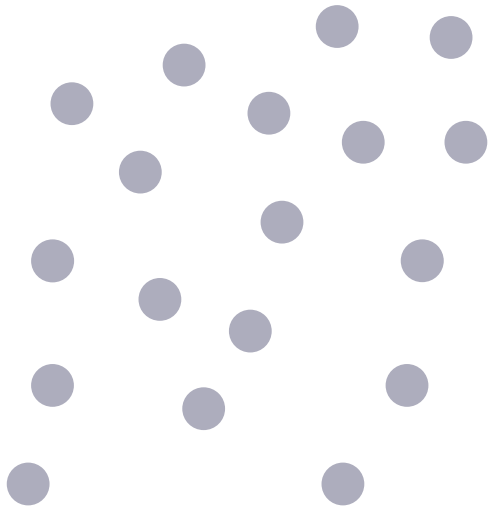
Назначаем векторы к центроидам



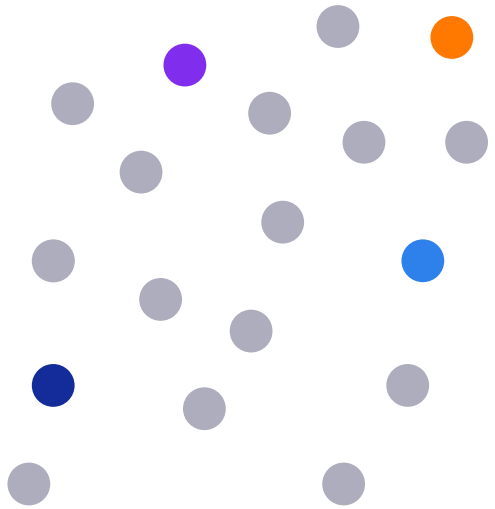
IVFFlat: когда простота — это SeqScan

PostgresPro

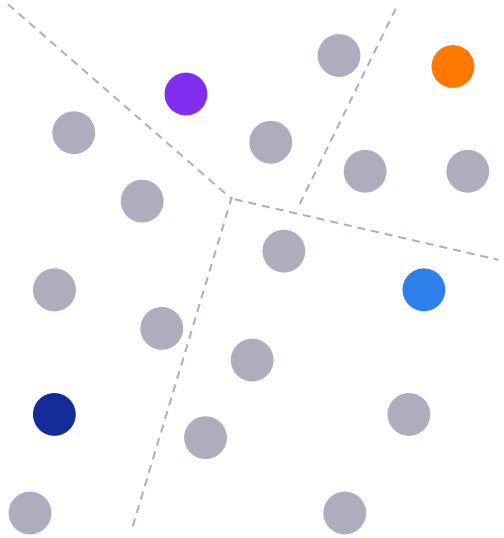
Исходные вектора



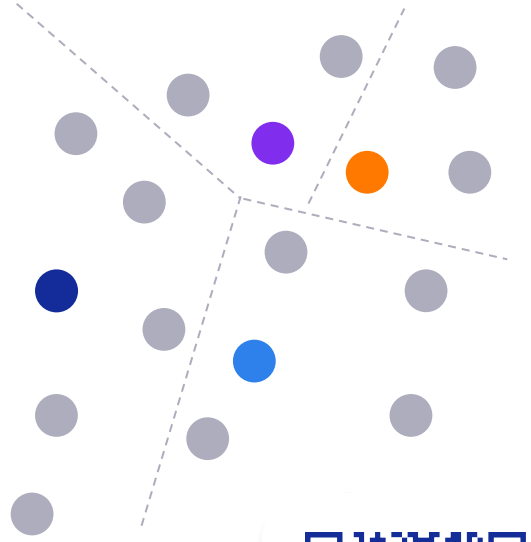
Выбор k центроидов



Назначаем векторы к центроидам

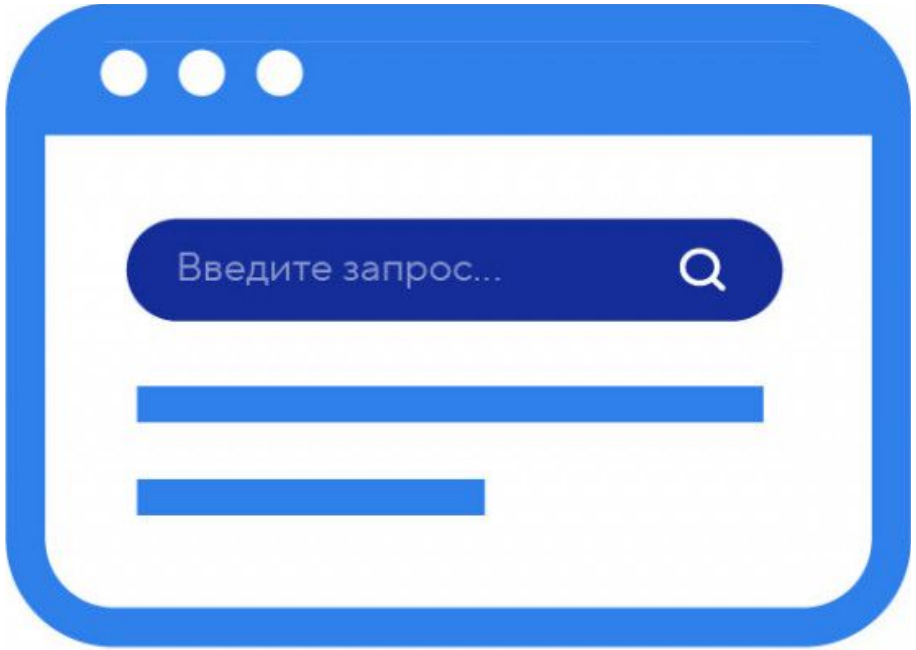


Перерасчет центров



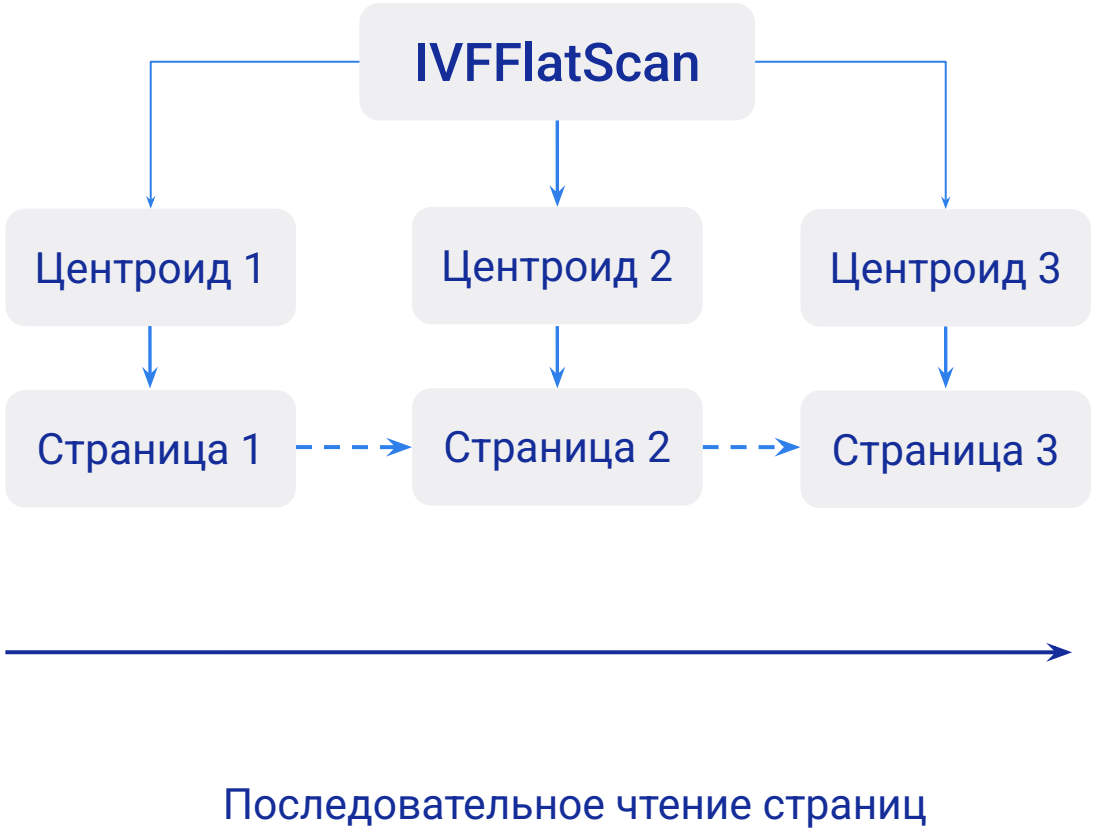
А поиск?

PostgresPro



IVFFlat: когда простота — это SeqScan

PostgresPro

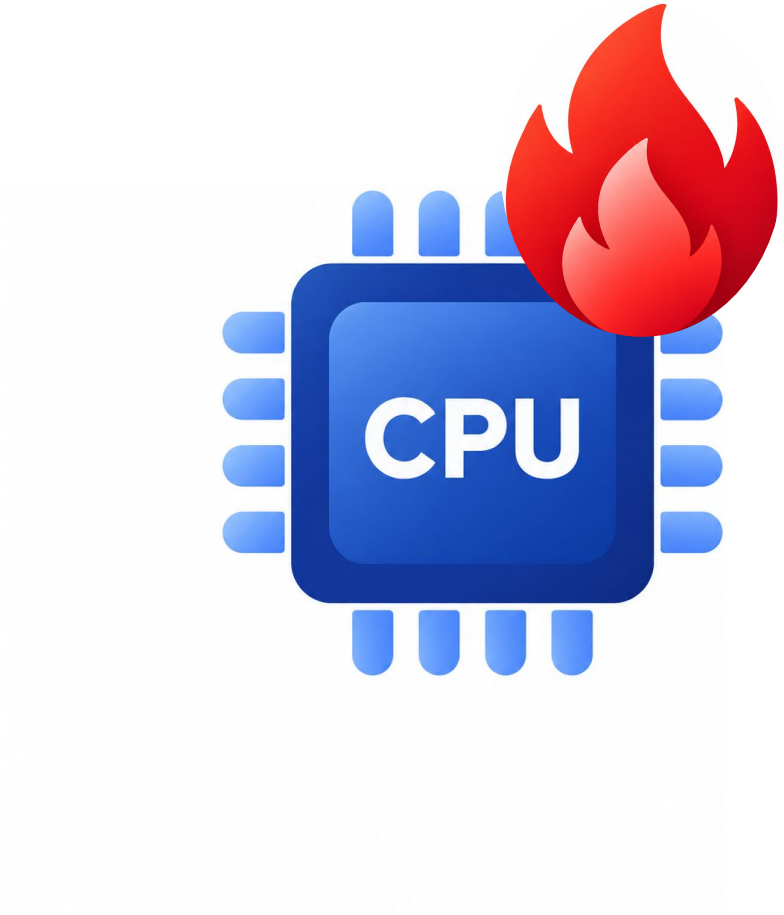


IVFFlat: стриминг

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

Главная проблема IVFFlat

PostgresPro



SIMD в vector_dist.c

PostgresPro

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(...)
```

SIMD в vector_dist.c

PostgresPro

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

PostgresPro

Миф AVX

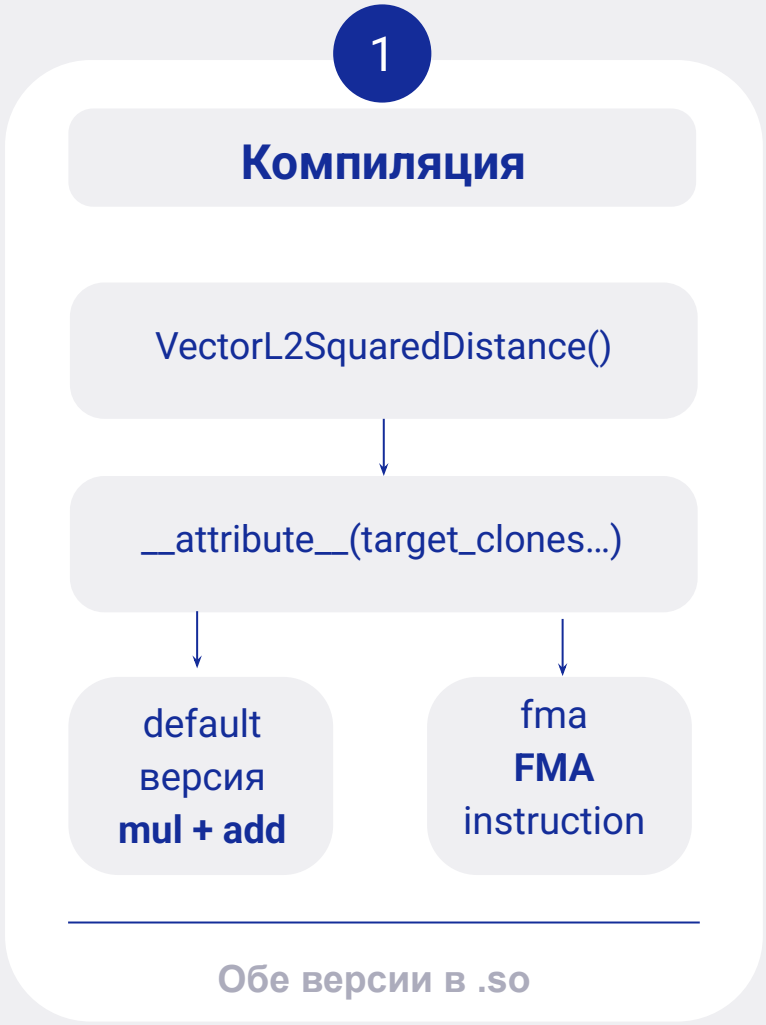
```
__m256 a = _mm256_load_ps(...)

__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;
}
```

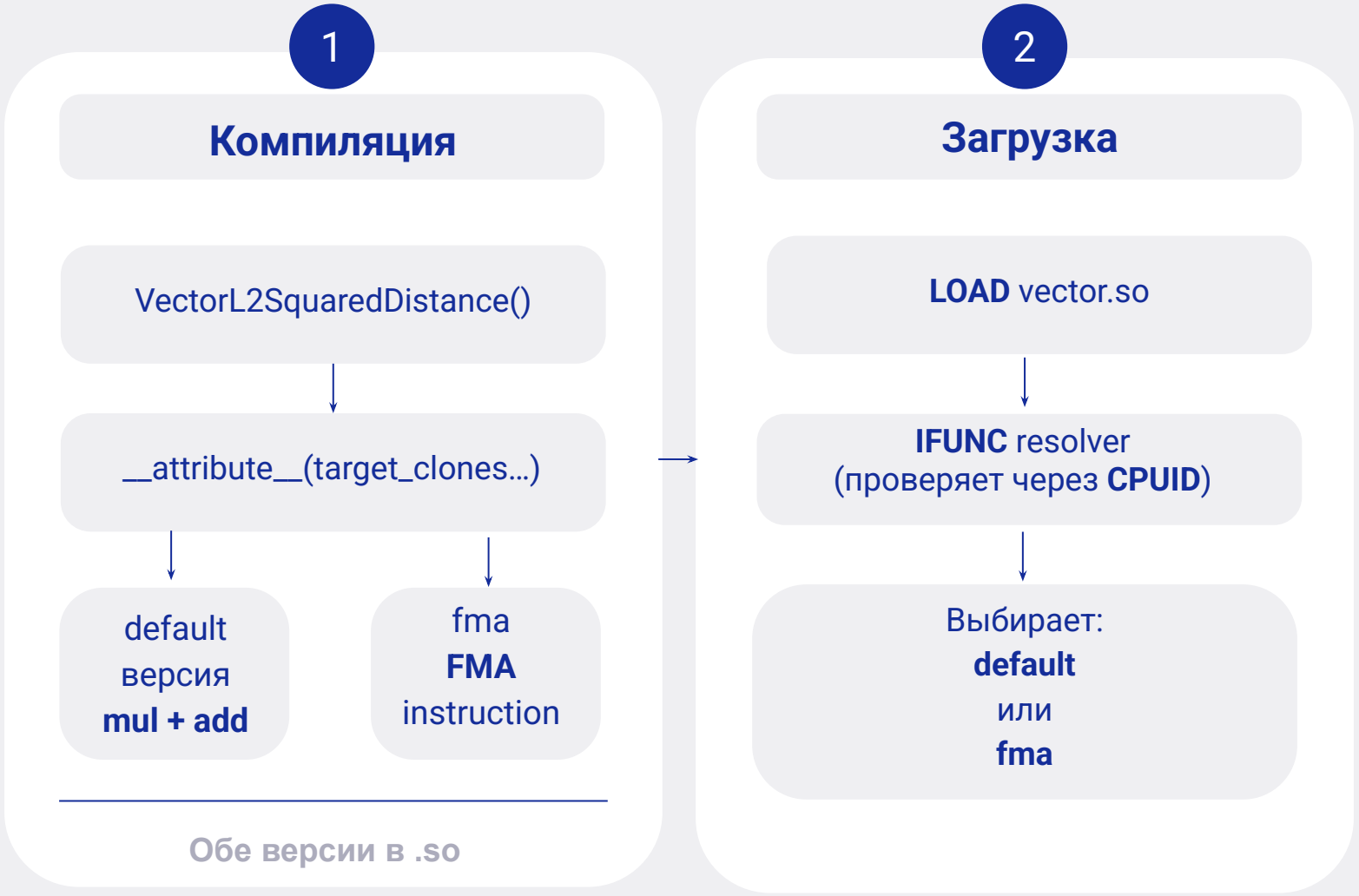
SIMD в vector_dist.c

PostgresPro



SIMD в vector_dist.c

PostgresPro



SIMD в vector_dist.c

PostgresPro

1

Компиляция

VectorL2SquaredDistance()

↓

__attribute__(target_clones...)

↓

default
версия
mul + add

fma
FMA
instruction

Обе версии в .so

2

Загрузка

LOAD vector.so

↓

IFUNC resolver
(проверяет через **CPUID**)

↓

Выбирает:
default
или
fma

3

Выполнение

VectorL2SquaredDistance(a,b)

Без **AVX**: по одному float за такт

a1 a2 a3 a4

С **AVX**: 8 float одной инструкцией

a1 a2 a3 a4 a5 ...

FMA instruction

Значит IVFFlat наш выбор?

PostgresPro

Значит IVFFlat наш выбор?

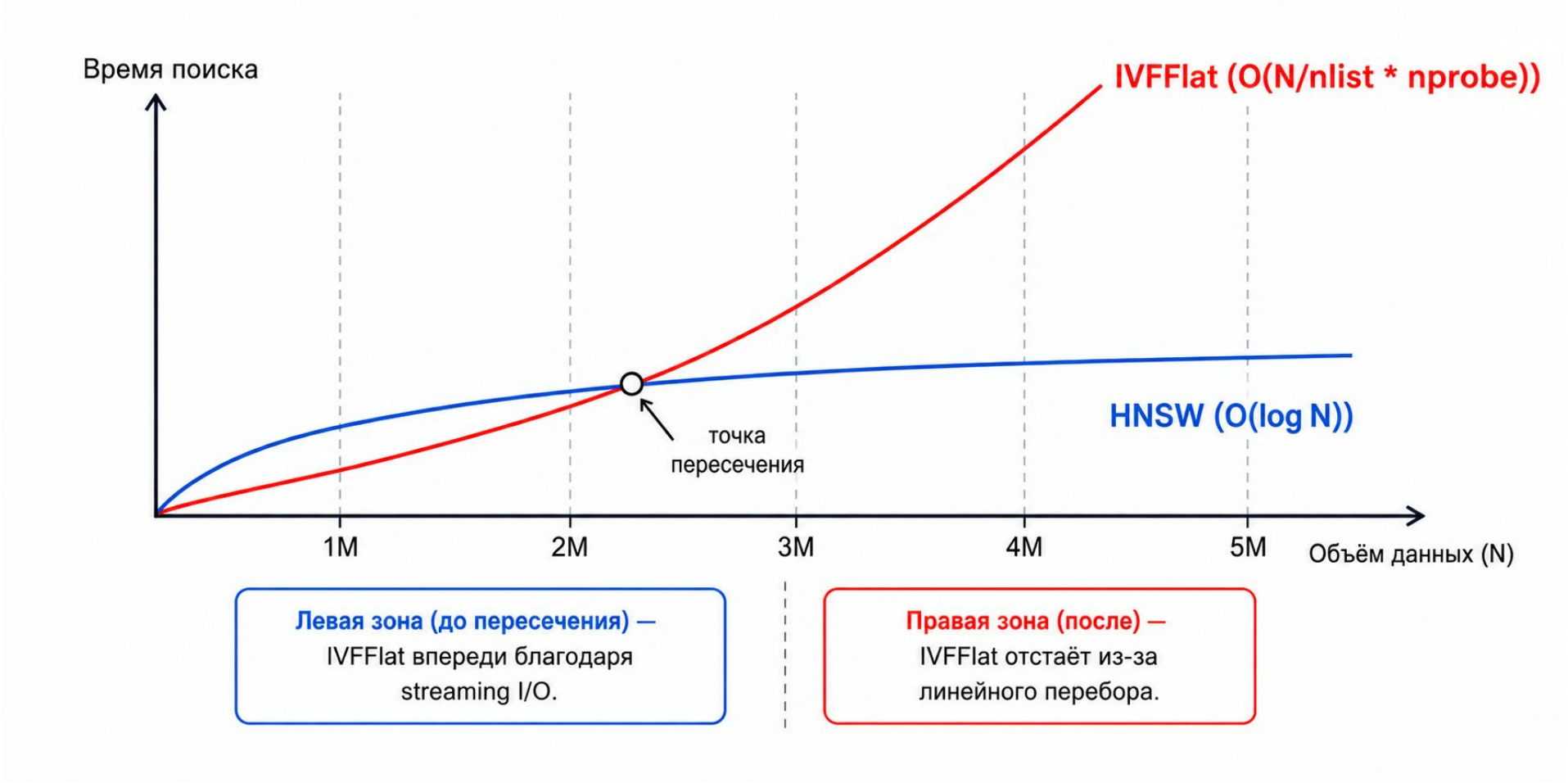
PostgresPro

PostgresPro

IVFFlat: сложность побеждает

Слабое место IVFFlat

PostgresPro

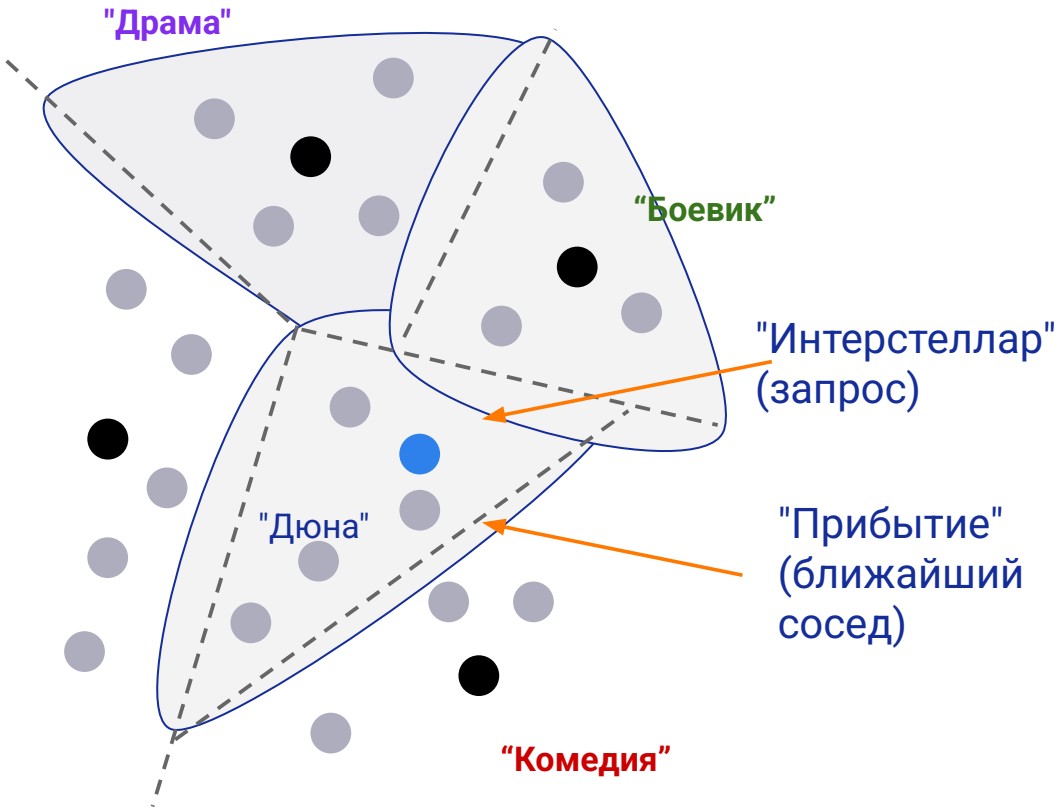


Слабое место IVFFlat

PostgresPro

Шаг 1: probes = 3

Выбираем 3 ближайших кластера по метрике

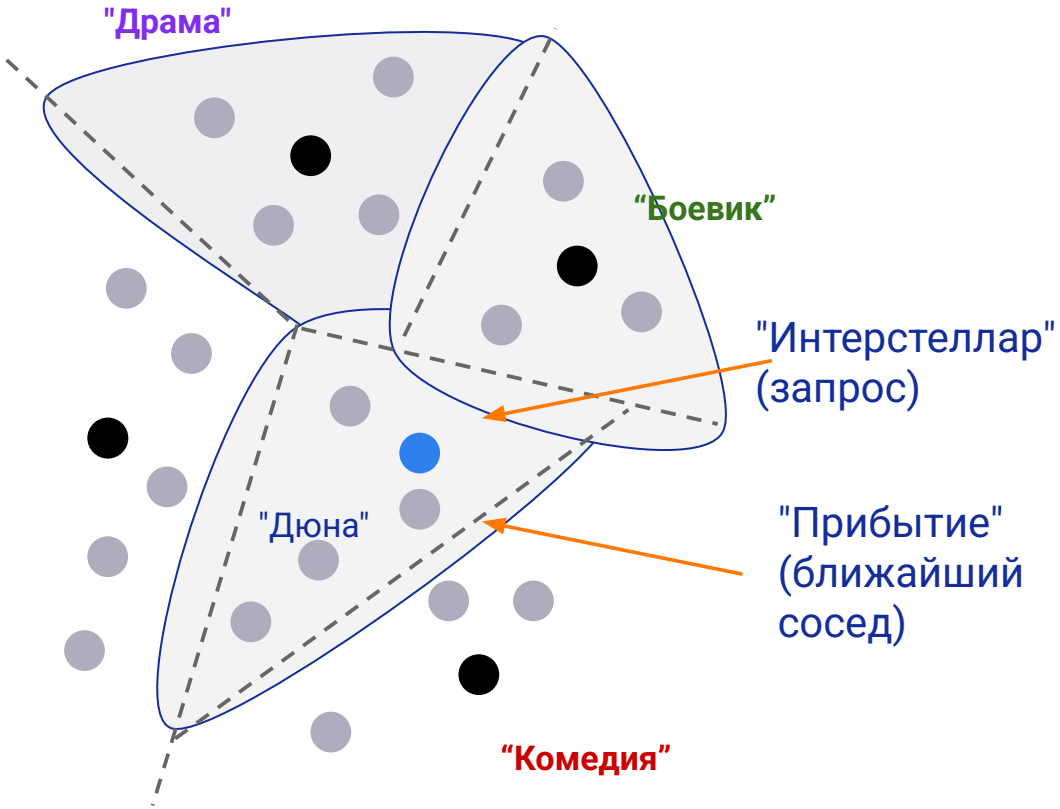


Слабое место IVFFlat

PostgresPro

Шаг 1: probes = 3

Выбираем 3 ближайших кластера по метрике



Шаг 2: внутри кластеров

Полный перебор (brute-force)

Кластер A — 200к векторов

Кластер B — 150к векторов

Кластер C — 150к векторов

Всего: 500K brute-force

PostgresPro

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
- Деградирует на больших данных ~ O(N/lists * probes)

HNSW

- Сложнее для кеша
- Больше случайных переходов

IVFFLAT

- Прост в реализации
- Хорош при малых данных
- Деградирует на больших данных ~ O(N/lists * probes)

HNSW

- Сложнее для кеша
- Больше случайных переходов
- Хорошо масштабируется

IVFFLAT

- Прост в реализации
- Хорош при малых данных
- Деградирует на больших данных ~ O(N/lists * probes)
- Упирается в память и I/O

HNSW

- Сложнее для кеша
- Больше случайных переходов
- Хорошо масштабируется

IVFFLAT

- Прост в реализации
- Хорош при малых данных
- Деградирует на больших данных ~ O(N/lists * probes)
- Упирается в память и I/O

HNSW

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

Теперь к практике

PostgresPro

PostgresPro

Профилирование: код против железа

Доказательство: Perf и 43% Cache Misses

PostgresPro

visited_hash

ef_search = 100
(максимум кандидатов в очереди)

```
visited_hash
    tids      // set of (blkno, offno)
    pointers
    offsets
    key -> visited
 ≈ 16 байт на запись
```

Всего: 100 x 16 = 1600 байт
≈ 1.6 Кб

Доказательство: Perf и 43% Cache Misses

PostgresPro

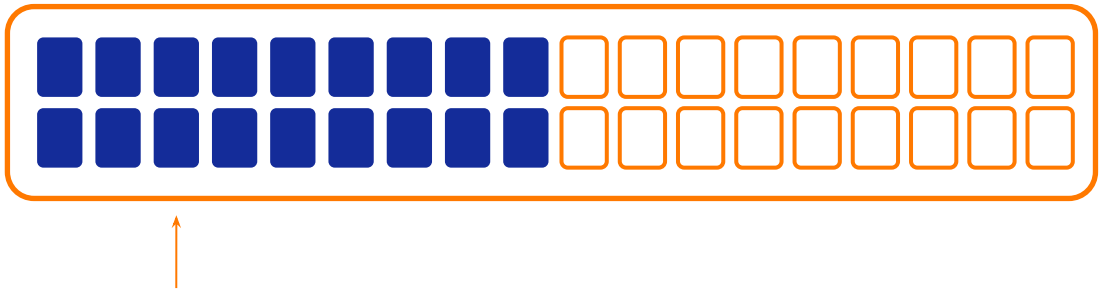
visited_hash

ef_search = 100
(максимум кандидатов в очереди)

```
visited_hash
    tids      // set of (blkno, offno)
    pointers
    offsets
    key -> visited
 ≈ 16 байт на запись
```

Всего: 100 x 16 = 1600 байт
≈ 1.6 Кб

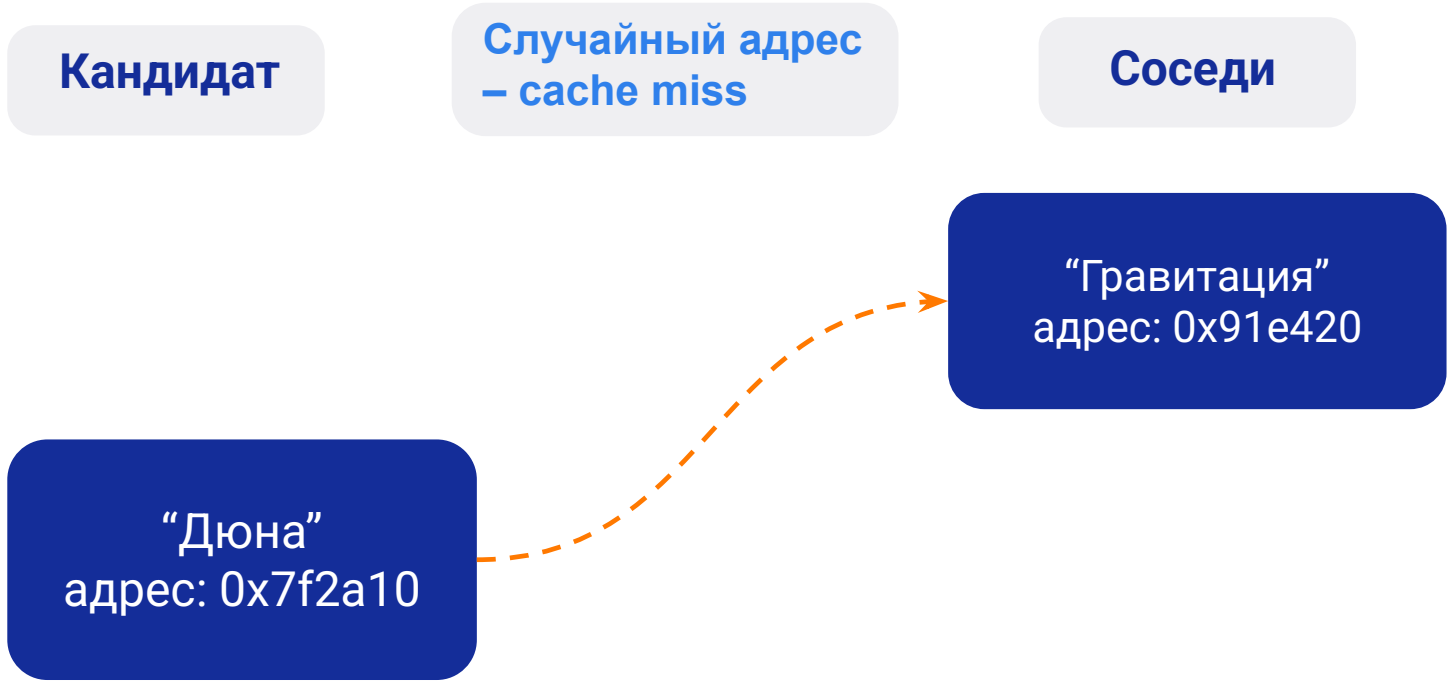
L1 cache: 32 Кб



Visited hash (1.5 Кб)

Доказательство: Perf и 43% Cache Misses

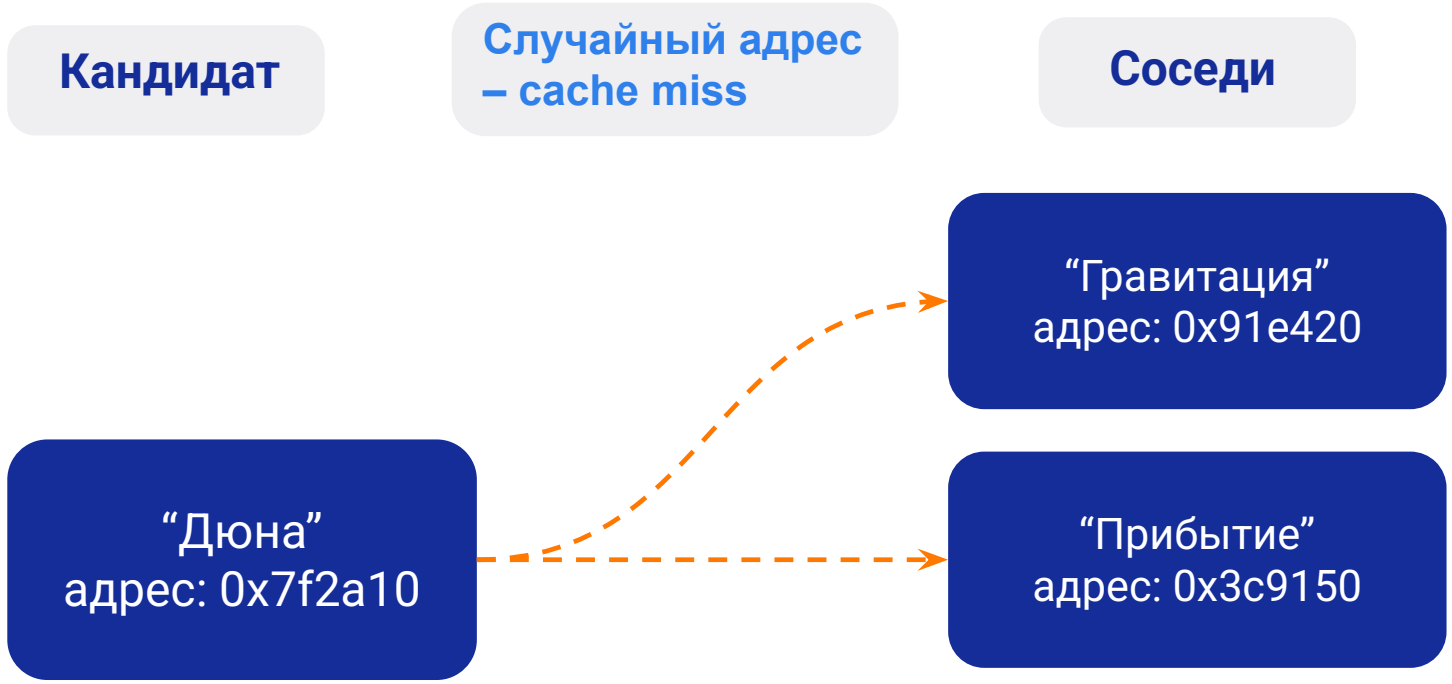
PostgresPro




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

Доказательство: Perf и 43% Cache Misses

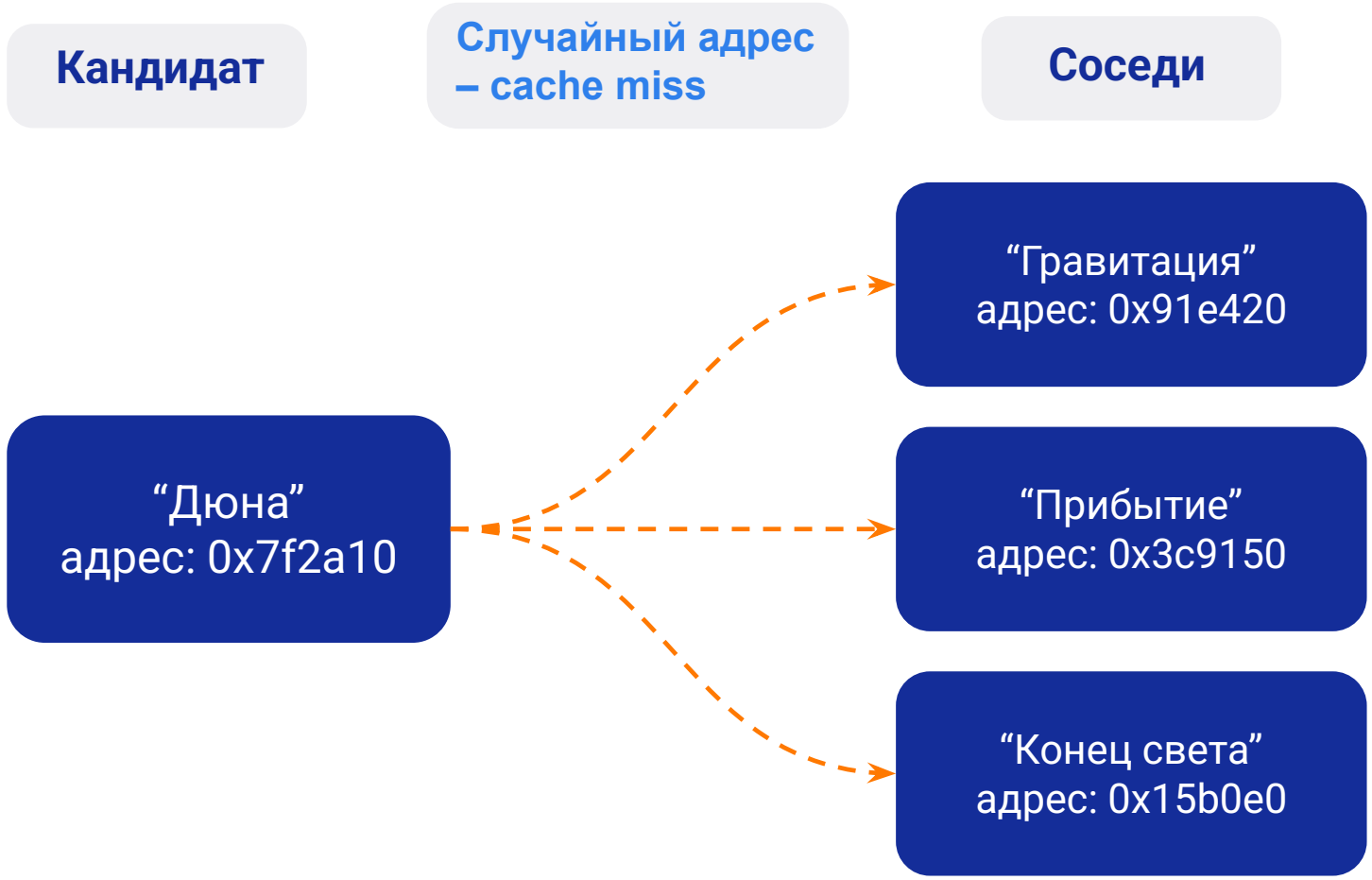
PostgresPro




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

Доказательство: Perf и 43% Cache Misses

PostgresPro




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

Доказательство: Perf и 43% Cache Misses

PostgresPro

43.4%

Cache Misses (L1-dcache-load-misses)

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

Доказательство: Perf и 43% Cache Misses

PostgresPro

43.4%
Cache Misses (L1-dcache-load-misses)

0.74
IPC (Instructions Per Cycle)

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

Доказательство: Perf и 43% Cache Misses

PostgresPro

43.4%
Cache Misses (L1-dcache-load-misses)

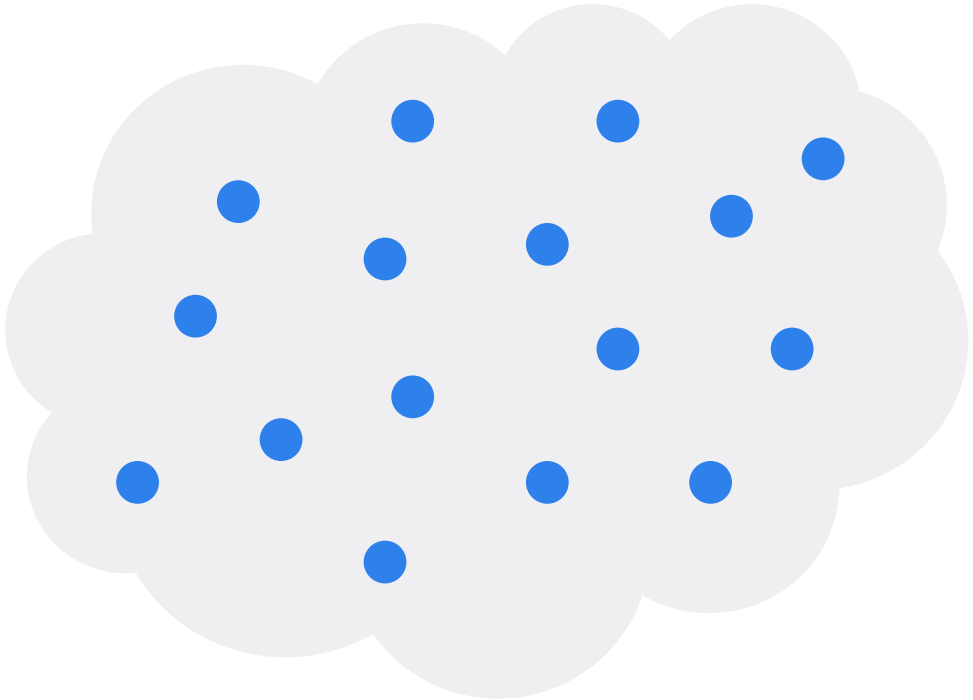
0.74
IPC (Instructions Per Cycle)

~80%
времени CPU ждёт данные из RAM

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

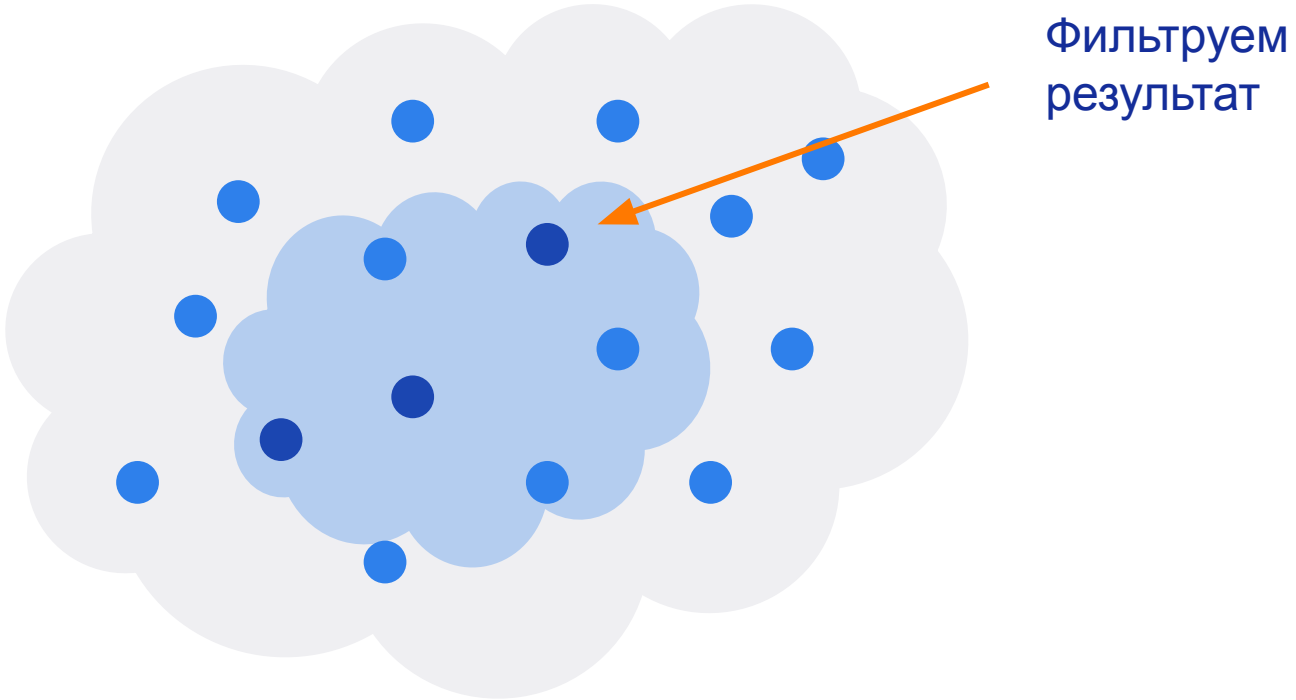
Стандартная ситуация

PostgresPro



Стандартная ситуация

PostgresPro



PostgresPro

Загадка о потерянном recall

Загадка о потерянном recall

PostgresPro

SQL - запрос

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

Загадка о потерянном recall

PostgresPro

SQL - запрос

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

Загадка о потерянном recall

PostgresPro

SQL - запрос

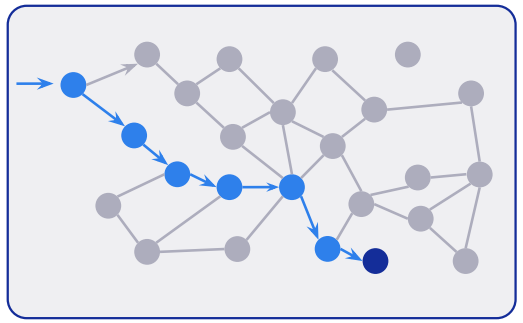
```
SELECT
    id,
    title,
    platform
FROM movies
WHERE release=2026
ORDER BY embedding
<=> :query_embedding
LIMIT 100;
```

Загадка о потерянном recall

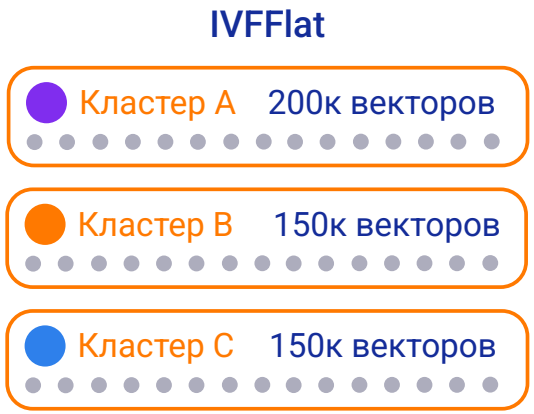
PostgresPro

SQL - запрос

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



HNSW



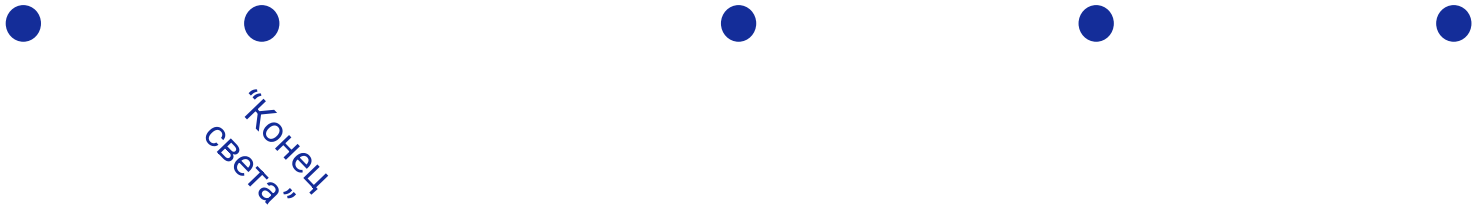
Загадка о потерянном recall

PostgresPro

Шаг 1. Топ-100 ближайших соседей

WHERE release=2026

Шаг 2. Применяем фильтрацию

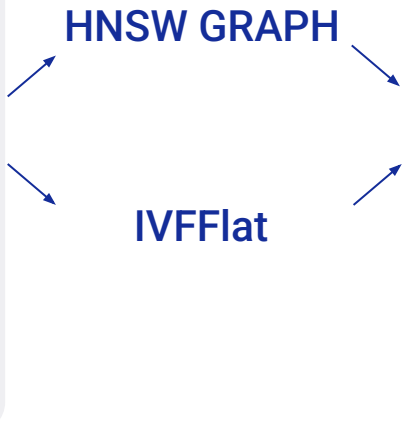


Загадка о потерянном recall

PostgresPro

SQL - запрос

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



Загадка о потерянном recall

PostgresPro

SQL - запрос

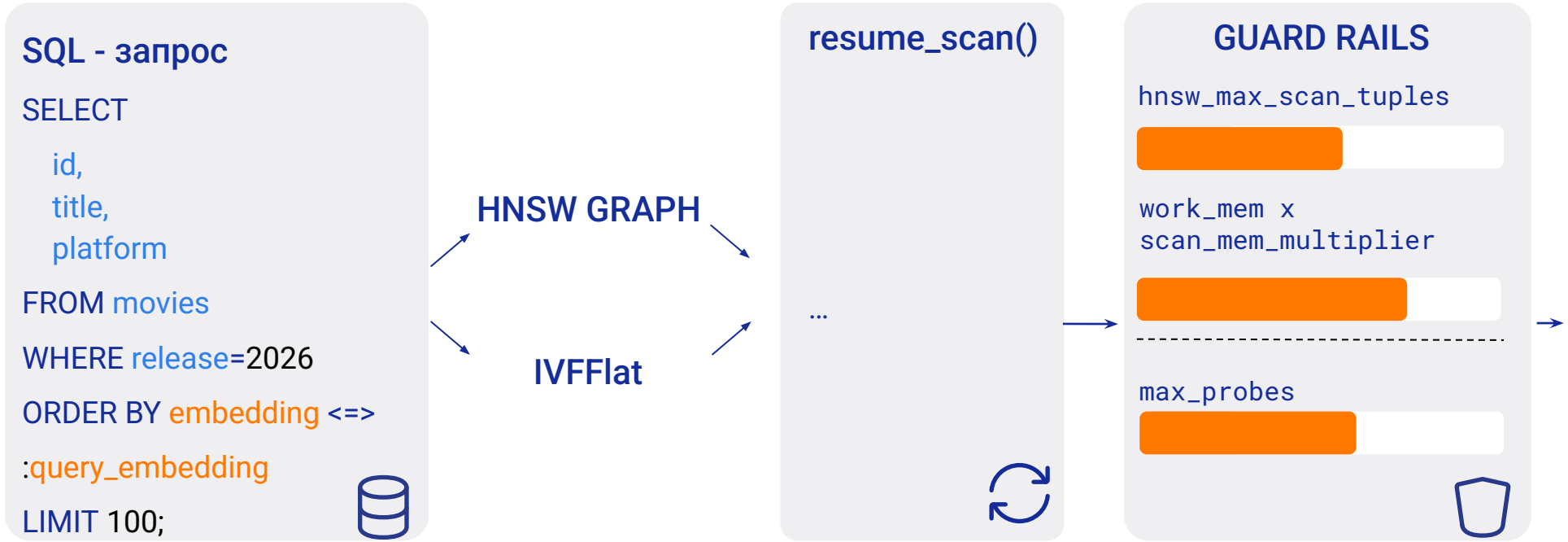
```
SELECT
    id,
    title,
    platform
FROM movies
WHERE release=2026
ORDER BY embedding <=>
:query_embedding
LIMIT 100;
```

resume_scan()

...

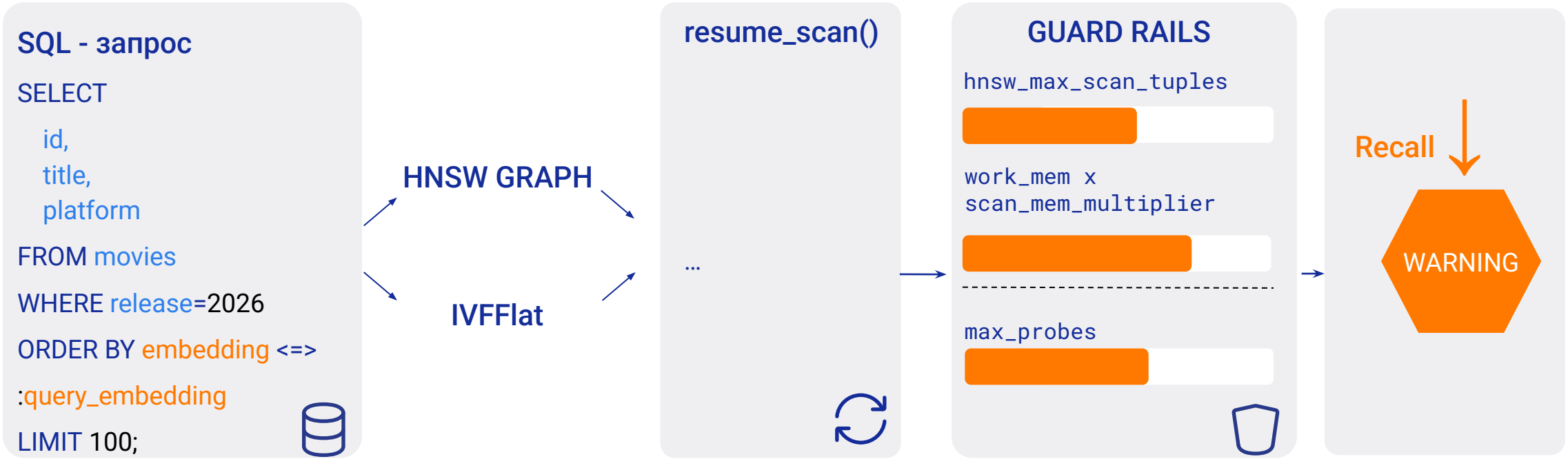
Загадка о потерянном recall

PostgresPro



Загадка о потерянном recall

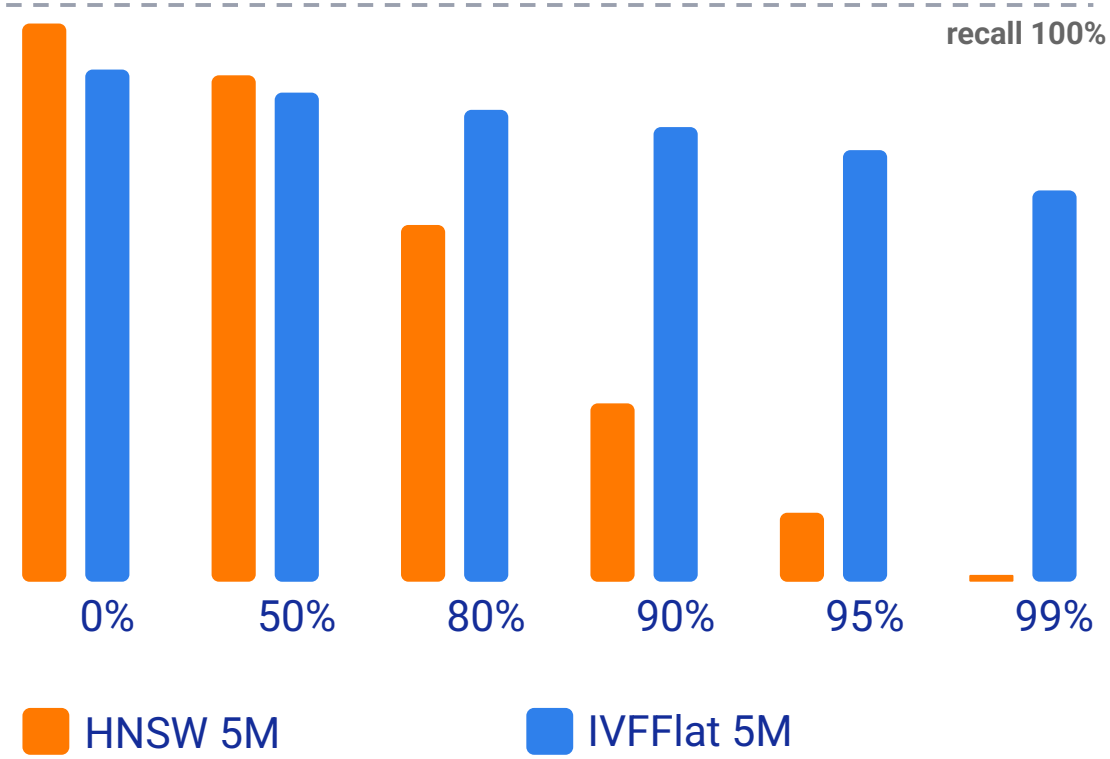
PostgresPro



Загадка о потерянном recall

PostgresPro

RECALL при росте фильтрации (5M векторов)



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)

PostgresPro

Рекомендации

Подводя итог

PostgresPro

1

SQL → AM

```
Select …
ORDER BY
vector <-> ‘[…]‘
LIMIT 100
```

Подводя итог

PostgresPro

1

SQL → AM

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

2

Страница на диске

Подводя итог

PostgresPro

1

SQL → AM

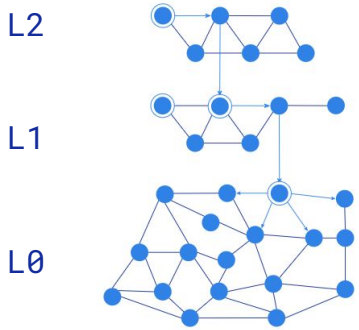
```
Select …
ORDER BY
vector <-> '[…]'
LIMIT 100
```

2

Страница на диске

3

HNSW спуск по слоям



Подводя итог

PostgresPro

1

SQL → AM

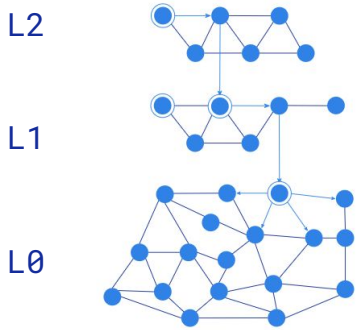
```
Select …
ORDER BY
vector <-> '[…]'
LIMIT 100
```

2

Страница на диске

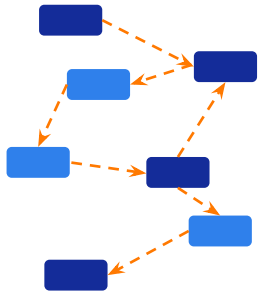
3

HNSW спуск по слоям



4

Pointer Chasing



Подводя итог

PostgresPro

1

SQL → AM

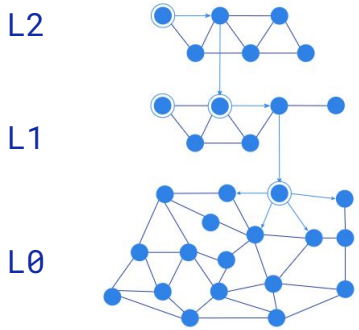
```
Select …
ORDER BY
vector <-> '[…]'
LIMIT 100
```

2

Страница на диске

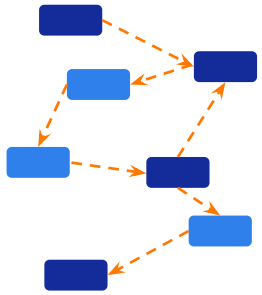
3

HNSW спуск по слоям



4

Pointer Chasing



5

LWLock в узле

Подводя итог

PostgresPro

1

SQL → AM

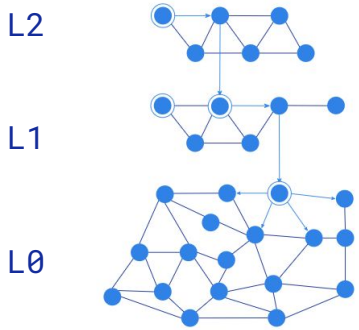
```
Select …
ORDER BY
vector <-> '[…]'
LIMIT 100
```

2

Страница на диске

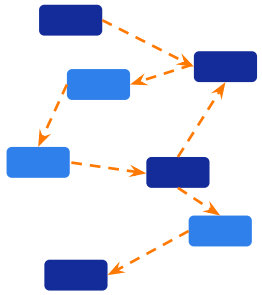
3

HNSW спуск по слоям



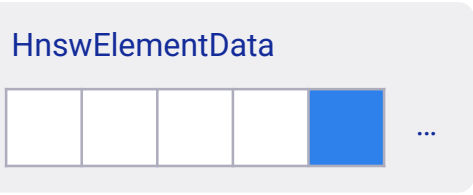
4

Pointer Chasing



5

LWLock в узле



6

WHERE-фильтр

Подводя итог

PostgresPro

1

SQL → AM

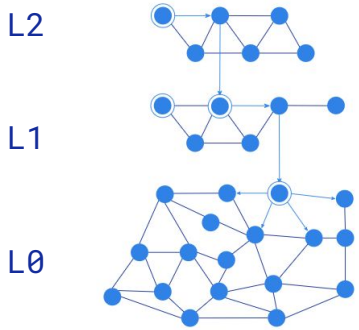
```
Select …
ORDER BY
vector <-> '[…]'
LIMIT 100
```

2

Страница на диске

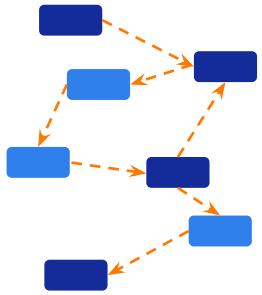
3

HNSW спуск по слоям



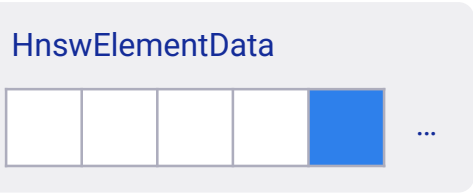
4

Pointer Chasing



5

LWLock в узле



6

WHERE-фильтр

7

IVFFlat альтернатива

Подводя итог

PostgresPro

1

SQL → AM

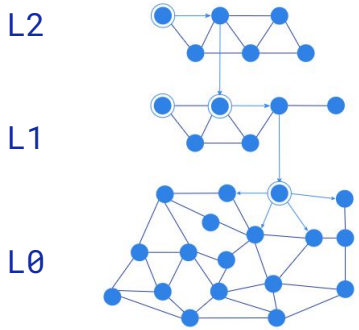
```
Select …
ORDER BY
vector <-> '[…]'
LIMIT 100
```

2

Страница на диске

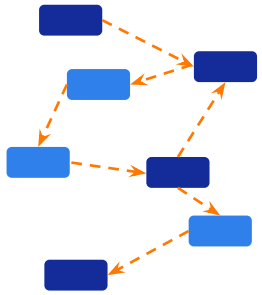
3

HNSW спуск по слоям



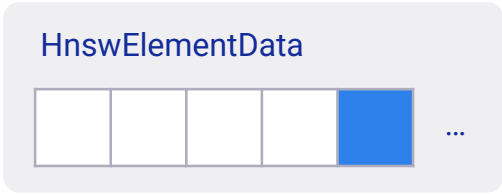
4

Pointer Chasing



5

LWLock в узле

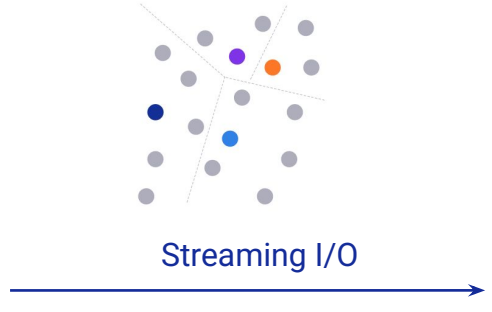


6

WHERE-фильтр

7

IVFFlat альтернатива



8

SIMD-ускорение через компилятор

IFUNC + CPUID

Инженерный выбор: Где ваш порог?

PostgresPro

Какой индекс выбрать?

IVFFlat

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

Расчет:

lists = rows/1000 (≤1M) или √rows (>1M)
probes = √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

Инженерный выбор: Где ваш порог?

PostgresPro

Что с параметрами?

- **halfvec** — float16, 2x меньше памяти без заметной потери recall
- **bit** — бинарные векторы, максимальная компактность
- **sparsevec** — разряженные 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

Инженерный выбор: Где ваш порог?

PostgresPro

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 уникальный 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);

PostgresPro

Спасибо! Ответим на ваши вопросы

Дарья Барсукова @brskv_dm