МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет Механико-математический

Кафедра Программирования

Направление подготовки Математика и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Петрунёва Таисия Евгеньевна

(Фамилия, Имя, Отчество автора)

Тема работы: Система векторного поиска с возможностью комбинирования различных методов квантизации

«К защите допущена»

Вр.и.о. зав. кафедрой,

к.ф.-м.н.

Емельянов П. Г./_____

(фамилия , И., О.) (подпись, МП)

«……»……20…г.

Научный руководитель

к.ф.-м.н.,

доцент кафедры систем информатики ФИТ

Наумчев А.В. _/_____

(фамилия , И., О.) (подпись, МП)

«……»……20…г.

Дата защиты: «……»……20…г.

Новосибирск, 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последние годы стремительно увеличивается объём данных, из-за чего становится необходимым эффективно хранить, обрабатывать и искать информацию. Современные системы работают с информацией, полученной из текста, изображения, видео и аудио. Традиционные методы поиска, основанные на точном сравнении, уже не способны обеспечить высокую скорость и точность поиска, в силу высокой размерности и неструктурированности данных. Поэтому существует необходимость в создании новых подходов к организации поиска.

Одним из наиболее перспективных направлений является векторный поиск, основанный на представлении объектов в виде многомерных векторов, расположенных в пространстве признаков. Поиск в данном методе осуществляется за счёт нахождения ближайших векторов с помощью различных метрик сходства. Векторный поиск получил широкое распространение в областях, связанных с анализом сложных данных.

С увеличением количества векторов в базе данных возрастают требования к вычислительным ресурсам и объёму используемой памяти. Для решения данной проблемы используют квантизации, позволяющие заменить исходные вектора на более компактное представление с меньшей размерностью. Однако использование квантизаций негативно влияет на точность поиска, из-за чего результат может оказаться неправильным. В связи с этим актуальной становится задача поиска оптимального компромисса между скоростью работы системы, объёмом используемой памяти и качеством результатов поиска.

Актуальность исследования определяется необходимостью разработки гибких и универсальных систем векторного поиска, способных эффективно функционировать при различных требованиях к производительности и точности. Кроме того, отдельный интерес представляет использование нескольких методов квантизации одновременно, так как их эффективность напрямую зависит от вида данных и от особенностей аппаратуры.

В рамках данной работы разрабатывается система векторного поиска с возможностью комбинирования различных методов квантизации и индексации. Предлагаемый подход предоставит возможность адаптировать систему под конкретную задачу, обеспечивая баланс между производительностью и точностью поиска. В работе рассматриваются современные методы квантизации векторных данных, а также предлагается архитектура системы, способная объединять различные подходы.

Таким образом, выполнение данной работы позволит внести вклад в развитие технологий поиска на больших данных, а также создать практическое решение, способное применяться в различных сферах, требующих быстрого и точного поиска информации.

1. ПОСТАНОВКА ЗАДАЧИ

Задача данной работы заключается в разработке системы векторного поиска с возможностью комбинирования различных методов квантизации и индексации.

2. Обзор предметной области

В данной главе вводятся основные определения, которые понадобятся в дальнейшей работе, рассматриваются разные методы квантизаций, а также существующие на данный момент методы.

2.1. Основные определения

2.1.1. Что такое векторная база данных

Векторная база данных – это специализированная система хранения, индексирования и поиска данных, работающая с векторными представлениями (эмбеддингами). Эти векторы – числовые массивы фиксированной длины, полученные из текста, изображений, аудио или других данных с помощью нейросетей.

Определение 2.1. *Вектором называется упорядоченный набор числовых значений, представляющий точку в многомерном пространстве признаков.*

Определение 2.2. *Векторное вложение – процесс преобразования исходных данных в векторы.*

Определение 2.3. *Пространство вложений – многомерное пространство, в котором располагаются все векторы. Геометрическое расположение векторов отражает семантические отношения между данными.*

Определение 2.4. *Эмбеддинг (embedding) – представление данных в виде высокоразмерных векторов. Данный способ преобразует объекты в координаты многомерного пространства признаков.*

Определение 2.5. *Аппроксимация – это метод, заменяющий изначальный объект на его приближённое представление, облегчая тем самым хранение, обработку или вычисления. Результат аппроксимации не совпадает с исходным объектом, но при этом обеспечивает достаточную точность.*

2.1.2. Как работает векторная база данных

Перед сохранением информации в векторную базу данных, данные преобразуются с помощью модели машинного обучения в вектора.

Этапы работы векторной базы данных:

1. Добавление векторов. Сохраняется такая информация как: вектор, уникальный ID, метаданные.
2. Построение индекса. Чтобы избежать сравнение запроса с каждым вектором, строится индекс, позволяющий быстро находить наиболее близкие вектора к вектору-запросу.
3. ANN. Если необходимо выполнить поиск по запросу, то запрос преобразуется в вектор и выполняется поиск ближайших соседей (ANN). В конце можно сделать переранжирование, после чего система вернёт результат.

2.1.3. Индексирование

В векторных базах данных индексы совпадают с индексными структурами ANN.

Определение 2.6. *Индексирование – это процесс создания специальной структуры данных (индекса), которая позволяет искать нужную информацию быстрее, чем если бы система просматривала все данные подряд.*

Как индексирование работает:

1. берём вектор;
2. добавляем его в структуру индекса;
3. индекс оптимизируется (векторы распределяются, связываются, сортируются или кодируются);
4. при поиске база данных использует индекс.

Определение 2.7. *Индекс – это структура данных, созданная на основе векторов.*

В векторных базах данных индекс могут быть разных типов:

- графовые индексы (рассматриваются в разделе 2.6);
- кластерные индексы и инвертированные структуры (раздел 2.5);
- методы квантизации (раздел 2.4);
- деревья и методы локально-чувствительного хеширования (LSH) [8];
- гибридные индексы;
- индексы в распределённых системах (здесь возникают понятия локального и глобального индекса).

Определение 2.8. *ANN-алгоритм (Approximate Nearest Neighbor, алгоритм приближённого поиска ближайших соседей) – это алгоритм поиска, предназначенный для нахождения объектов, наиболее близких к заданному вектору запроса, с допустимой погрешностью относительно точного результата.*

В отличие от точного поиска ближайших соседей, ANN-алгоритмы не гарантируют нахождение абсолютно ближайших объектов, однако позволяют значительно уменьшить вычислительные затраты и ускорить поиск на больших высокоразмерных данных.

Работа ANN-алгоритмов обычно включает следующие этапы:

1. построение структуры индексирования (графа, дерева, кластеров или квантизированных представлений);
2. ограничение области поиска путём отбора наиболее перспективных кандидатов;
3. вычисление расстояний только для выбранных объектов;
4. возврат множества ближайших объектов.

2.1.4. Кластеризация

Определение 2.9. *Кластеризация – это процесс группировки объектов в группы(кластеры) так:*

1. *Векторы внутри одной группы максимально похожи друг на друга по определённым признакам*
2. *Векторы из разных групп максимально отличаются*

Кластеризация разбивает набор данных на множества:

$$X = \bigcup_{i=1}^{K} C_i$$

Определение 2.10. *Центроид – это представитель группы векторов в векторной базе данных.*

Более формально:

Центроид – это среднее значение кластера векторов, присвоенных ему. Если векторы $x_1, \ldots, x_n$ *принадлежат одному кластеру* C_i*, тогда центроид равен:*

$$c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

где $|C_i|$ *– количество элементов в кластере.*

Определение 2.11. *Внутрикластерная ошибка – сумма расстояний всех точек от центроида.*

Определение 2.12. *Внутрикластерная компактность – как сильно сжаты точки внутри кластера.*

Определение 2.13. *Межкластерная разделённость – насколько хорошо кластеры отделены друг от друга.*

Определение 2.14. *Ячейка Вороного (или область Вороного) – это область в пространстве, состоящая из всех точек, которые ближе к одному заданному центру, чем ко всем другим центрам.*

Пусть есть набор центроидов $C = c_1, c_2, \ldots, K$*. Тогда ячейка Вороного*

для центроида c_i – это множество точек x, таких что:

$$\|x - c_i\| \leq \|x - c_j\|, \quad \forall j \neq i$$

То есть это регион, которому ближе всего центроид c_i.

Определение 2.15. *Кластеризация методом k-средних (k-means) – это метод кластеризации, предназначенный для разделения множества объектов размера n на k кластеров таким образом, чтобы объекты внутри одного кластера были максимально похожи друг на друга, а объекты из разных кластеров максимально различались. Это приводит к разбиению пространства на ячейки Вороного.*

Пусть имеется множество векторов $X = \{x_1, x_2, \dots, x_n\}$, которое необходимо разбить на k кластеров $C_1, C_2, \dots, C_k$.

Цель метода заключается в минимизации суммарного квадрата расстояний между объектами и центроидами соответствующих кластеров:

$$J = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - c_i\|^2$$

где c_i – центроид кластера C_i.

Как работает алгоритм:

1. Задаётся число кластеров k.
2. Инициализируются k центроидов. Например, выбрать случайные k точек.
3. Каждой точке назначается ближайший центроид.
4. Для каждого кластера пересчитывается новый центроид как среднее значение его объектов.

5. Шаги 3 – 4 повторяются до достижения сходимости (точки больше не меняют кластер, или центроиды перестают изменяться) или заданного числа итераций.

2.1.5. Квантизация

Определение 2.16. *Квантизация – это процесс сжатия векторов путём замены их небольшим количеством репрезентативных кодовых слов из кодовой книги.*

Определение 2.17. *Подпространство – часть исходного пространства признаков, полученная разбиением вектора на несколько независимых сегментов.*

Определение 2.18. *Кодовая книга – набор центроидов для каждого подпространства.*

Определение 2.19. *Кодовое слово (code) – номер центроида в кодовой книге для конкретного подпространства.*

Определение 2.20. *Пусть дана кодовая книга* $C = \{c_1, c_2, \dots, c_K\}$. *Квантователь – это функция, которая заменяет исходный вектор на его приближение, используя конечный набор центроидов (кодовых слов):*

$$q : \mathbb{R}^d \to C$$

$$q(x) = argmin_{i \in \{1,..,K\}} \|x - c_i\|$$

Квантователь делает две вещи:

1. Находит ближайшее кодовое слово (по расстоянию)
2. Возвращает индекс этого слова

Определение 2.21. *Декодер – это алгоритм, восстанавливающий изначальные данные либо их приближённое представление.*

Определение 2.22. *Ошибка квантизации – это разница между изначальным вектором и его приближенной версией.*

Определение 2.23. *Таблица быстрого доступа (lookup table) – это заранее вычисленная таблица расстояний между вектором-запросом и всеми кодовыми словами (центроидами) кодовых книг.*

Определение 2.24. *Короткий код (short code) – это компактное представление вектора, состоящее из индексов выбранных центроидов.*

2.2. Остаточная индексация

Пусть есть вектор x. Пространство кластеризуется с помощью алгоритма $k-means$, получается набор центроидов $C_i = \{c_1, c_2, \dots, c_k\}$. Для каждого вектора находится ближайший центроид $c(x)$. Тогда остаток (residual) равен $r = x - c(x)$.

2.3. Метрики расстояний между векторами

Расстояния между векторами отражают схожесть каждого вектора с другими векторами векторной базы данных. Схожесть можно определять различными способами:

1. Евклидово расстояние (L2-норма) для n-мерного вектора

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

2. Манхэттенское расстояние (L1-норма) для n-мерного вектора

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

3. Косинусная дистанция

   На основе косинусного сходства:

$$\cos(x, y) = \frac{x \cdot y}{\|x\|\|y\|}$$

   Расстояние:

$$d_{cos}(x, y) = 1 - \cos(x, y)$$

   Используется данный способ вычисления, когда важны углы, а не длины. Широко распространена в рекомендательных системах и в работе с текстами.

4. Расстояние Хэмминга [8]

$$d(x, y) = \sum_{i=1}^{n} \delta_{x_i y_i}$$

   где δ – это дельта Кронекера.

   Дельта Кронекера или символ Кронекера – индекатор равенства элементов:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

5. ADC

   Асимметричное вычисление расстояния [15] [5] относится к эффективному алгоритму поиска в таблице, который вычисляет приблизи-

тельное значение IP.

ADC работает следующим образом:

- Сначала для каждой ячейки вычисляется остаточное значение $z' = r(z)$ вектора запроса ($z' = z$ если индекс не используется).
- Из этого остаточного значения z' вычисляется набор таблиц быстрого доступа $\{\mathbb{D}^j\}_{j=0}^{m}$, где m – количество подквантователей продуктового квантователя. ***J***-ая таблица быстрого доступа содержит расстояние между j-ым подвектором z' и всеми центроидами j-го подквантователя:

$$\mathbb{D}^j = (\|(z')^j - \mathbb{C}^j[0]\|^2, \dots, \|(z')^j - \mathbb{C}^j[k-1]\|^2)$$

- Затем все кандидаты сканируется, и таблицы быстрого доступа используются, чтобы вычислить расстояние между вектором запроса z и каждым коротким кодом c следующим образом:

$$adc(z, c) = \sum_{j=0}^{m-1} \mathbb{D}^j[c[j]]$$

Таким образом, ADC вычисляет расстояние между вектором запроса z и каждым кодом c, суммируя расстояния между подвекторами z' и центроидами, связанными с кодом c в m подпространствах продуктового квантователя.

6. SDC

В симметричном вычислении расстояния [13] оба вектора (вектор x запроса и вектор y) заменяются на их приближённые версии через кодовые слова, после чего вычисляется расстояние между соответ-

ствующими кодами. То есть

$$\bar{d}(x, y) \approx d(q(x), q(y))$$

где $q(x), q(y)$ – квантизированные представления для векторов x и y.

2.4. Методы квантизации

В данном разделе будут описаны разные существующие методы квантизаций. Они позволяют уменьшать размер векторов, что в свою очередь сокращает объем неоходимый для хранения памяти, и ускорять поиск ближайших соседей (ANN), при этом стараясь сохранять точность.

Методы квантизации различаются способом представления исходного вектора и методом построения кодовых книг.

По принципу работы их можно разделить на несколько групп:

- векторная квантизация (VQ, раздел 2.4.1);
- скалярная квантизация (SQ, раздел 2.4.2);
- методы разбиения пространства на подспространства (PQ – раздел 2.4.3, OPQ – раздел 2.4.4);
- остаточные методы (RVQ, раздел 2.4.6);
- аддитивные методы (AQ, раздел 2.4.5);
- специализированные методы для быстрого вычисления расстояний (RaBitQ – раздел 2.4.7, LVQ – раздел 2.4.8).

Далее рассматриваются основные представители этих подходов.

2.4.1. Vector Quantization (VQ)

В классической векторной квантизации [16] [5] кодовое слово представляет собой элемент кодовой книги и обычно соответсвует центроиду кластера.

1. Если кодовая книга одна

   Пусть есть кодовая книга с $\mathbb{C} = \{c_1, c_2, \dots, c_K\}$, которая содержит K кодовых слов. Для каждого вектора x выбираем $q(x) = c_{i^*}$, где q – это квантователь, $i^* = arg\min_i \|x - c_i\|$. Храним в итоге только индекс i, а не сам вектор x.

2. Если кодовых книг несколько

   VQ квантизирует элемены в наборе данных с помощью M кодовых книг $C^1, C^2, \dots, C^M$. Каждая кодовая книга C^m содержит K кодовых слов и каждое кодовое слово – это d-мерный вектор, то есть $C^m = \{c^m[1], c^m[2], \dots, c^m[K]\}, c^m[k] \in \mathbb{R}^d$ для $1 \le m \le M$ и $1 \le k \le K$. Обозначим i_x^m как индекс кодового слова в кодовой книге C^m, которому соответствует элемент x, затем x будет аппроксимировано выражением:

$$\tilde{x} = \sum_{m=1}^{M} c^m[i_x^m]$$

   Следовательно, внутреннее произведение между запросом q и элементом x, то есть $q \in x$, аппроксимировано следующим:

$$q^\top \tilde{x} = \sum_{m=1}^{M} q^\top c^m[i_x^m]$$

   Для поддержания погрешности квантизирования на достаточно низ-

ком уровне для ANN поиска требуется очень большая кодовая книга. Однако создание таких кодовых книг затруднительно как с точки зрения требований к обработке данных, так и с точки зрения объема памяти.

Есть VQ алгоритмы с разными стратегиями квантизации и процедурами изучения кодовой книги, например Product Quantization (PQ, раздел 2.4.3), Optimized Product Quantization (OPQ, раздел 2.4.4), Residual Quantization (RQ или RVQ, раздел 2.4.6) и Additive Quantization (AQ, раздел 2.4.5). Для сжатия данных M индексов кодового слова $\{i_x^1, i_x^2, \dots, i_x^M\}$ сохранены вместо изначального d-мерного вектора x, что позволяет хранить очень большие наборы данных в основной памяти одной машины.

2.4.2. Scalar Quantization (SQ)

Скалярная квантизация (SQ) [15] – это метод квантизации, при котором каждый элемент вектора квантизируется независимо (обрабатывается каждый скаляр).

Пусть есть вектор $x = (x_1, x_2, \dots, x_d) \in \mathbb{R}^d$, тогда скалярный квантователь q равен:

$$q(x) = (q_1(x_1), q_2(x_2), \dots, q_d(x_d))$$

где каждый q_i выбирает ближайшее значение из заранее построенного набора уровней квантизации (конечного множества допустимых значений).

2.4.3. Product Quantization (PQ)

Продуктовый квантователь (product quantizer) [6] [5], преодолевает ошибку квантизации, разделяя вектор $y \in \mathbb{R}^d$ на m подвекторов $y = (y^1, y^2, \dots, y^m)$, где $y^j \in \mathbb{R}^{d/m}$. Каждый подвектор квантизируется независимо с использованием подквантователя q^j.

Каждый подквантователь q^j имеет отдельную кодовую книгу $C^j = (c_i^j)_{i=1}^k$ мощности k. Следовательно $C = C^1 \times C^2 \times \cdots \times C^m$, то есть мощность кодовой книги продуктового квантователя (product quantizer) равна k^m.

Таким образом, продуктовый квантователь имеет множество центроидов k^m, в то время как требуется только хранение и обучение m кодовых книг с мощностью k. Продуктовый квантователь кодирует вектор y в короткий код путём объединения кодов, полученных с помощью подквантователей $e(y) = (i_1, \dots, i_m)$, таким образом, что

$$q(y) = (q^1(y^1) \dots q^m(y^m)) = (c_{i_1}^1 \dots c_{i_m}^m)$$

Для кода $(i_1, \dots, i_m)$ требуется $\lceil \log_2 k^m \rceil$ бит памяти.

2.4.4. Optimized Product Quantization (OPQ)

Оптимизированная продуктовая квантизация (OPQ) [9] – вариация PQ.

Одним из основных недостатков PQ является то, что вектор x вынужден вписываться в модель, которая предполагает, что данные были получены из статически независимых подпространств (из-за чего компоненты, сильнее всего влияющие на расстояние, могут оказаться разделены). Меньшая ошибка квантизации может быть достигнута, если в модель добавить больше сте-

пеней свободы. В частности, поскольку вращение является ортогональным преобразованием (сохраняет длины векторов и скалярное произведение), можно поэкспериментировать с вращением исходного пространства векторов перед обучением кодовых книг, чтобы уменьшить ошибку квантизации.

В стандартном PQ исходный вектор разбивается на независимые подпространства, затем каждое подпространство квантизируется отдельно, таким образом получается компактный код.

В свою очередь, в OPQ перед разбиением на подпространства вводится оптимальное ортогональное преобразование (поворот), то есть вектор $x \rightarrow R \cdot x$, где R – ортогональная матрица.

OPQ ищет матрицу R, такую что:

$$\min_{R, C^1, \ldots, C^M} \sum_{i=1}^{N} \| x_i - R^\top q(R x_i) \|^2$$

где R – матрица поворота, q – продуктовый квантователь, C^j – кодовые книги для каждого подпространства.

2.4.5. Additive Quantization (AQ)

Пусть есть M кодовых книг $C_1, C_2, \ldots, C_M$ размера K. Модель AQ [2] кодирует вектор $x \in \mathbb{R}^d$ как сумму M кодовых слов (из каждой кодовой книги по одному слову). Более детально, вектор кодируется с помощью M-набора идентификаторов (ID) кодовых слов $(i_1, i_2, \ldots, i_M)$, где каждый идентификатор находится в диапазоне от 1 до K. Тогда код декодируется

следующим образом:

$$x \approx x' = \sum_{m=1}^{M} c^m(i_m), i_m \in 1 \dots K$$

где $c^m(k)$ – это k-ый элемент в m-ой кодовой книге

2.4.6. Residual Vector Quantization (RVQ)

RVQ [17] [11] – это рекурсивный алгоритм, в котором квантователь с номером $n + 1$ кодирует остаток от квантователя с номером n.

Пусть есть вектор $x \in \mathbb{R}^D$. Из первой кодовой книги выбирается ближайший центроид c_1.

Затем вычисляется остаток $r_1 = x - c_1$.

Для остатка r_1 используется вторая кодовая книга, выбирается ближайший центроид c_2.

Затем вычисляется остаток $r_2 = r_1 - c_2$.

Данные действия повторяются n раз $r_n = r_{n-1} - c_n$.

В результате получается код $c_1, c_2, \dots, c_n$.

2.4.7. RaBitQ

RaBitQ [7] – метод квантизации, преобразующий координаты вектора в компактное бинарное представление с сохранением дополнительной информации о диапазонах значений.

На вход поступают вектор данных o_r и вектор запроса q_r, RaBitQ сначала нормализует их на основе опорного вектора c (например, центроида набора

данных или кластера). Нормализованные вектора выражаются как:

$$o = \frac{o_r - c}{\|o_r - c\|}, q = \frac{q_r - c}{\|q_r - c\|}$$

RaBitQ вычисляет евклидово расстояние между исходными векторами (то есть между o_r и q_r) как:

$$\|o_r - q_r\|^2 = \|(o_r - c) - (q_r - c)\|^2 = \|o_r - c\|^2 + \|q_r - c\|^2 - 2 \cdot \|o_r - c\| \cdot \|q_r - c\| \cdot \langle q, o \rangle$$

где $\|o_r - c\|$ и $\|q_r - c\|$ расстояния векторов данных и запросов вектора c, который может быть предварительно вычислен перед вычислением расстояния и использован повторно. Таким образом, RaBitQ квантизирует нормальный вектор данных o и фокусируется на оценке $\langle q, o \rangle$.

RaBitQ вначале генерирует случайную ортогональную матрицу P и проецирует вектор o с помощью матрицы P, то есть $o_p = P \cdot o$. Так как P ортогональная матрица, она сохраняет внутреннее произведение

$$\langle q_p, o_p \rangle = \langle q, o \rangle$$

. Таким образом, есть возможность оценить сжатие o_p и оценку $\langle q_p, o_p \rangle$. Учитывая, что o_p имеет единичную норму, RaBitQ использует следующую кодовую книгу

$$C = \{+\frac{1}{\sqrt{D}}, -\frac{1}{\sqrt{D}}\}$$

где каждое кодовое слово это также единичный вектор, и o_p квантизован к ближайшему кодовому слову в C. По сути, это означает выбор между -1 и +1 в соответствии со знаком для каждого измерения o_p. Квантизированный вектор обозначается как $\tilde{o_p}$.

2.4.8. LVQ

Локально-адаптированная векторная квантизация (LVQ) [4] [7].

Для каждого вектора $x = (x_1, x_2, \dots, x_d)$, $\mu = (\mu_1, \mu_2, \dots, \mu_d)$ – центры всех векторов в X. LVQ сначала центрирует вектор по среднему значению, вычитая его среднее значение μ, в результате чего получается $x' = x - \mu$. Затем вычисляется минимальное и максимальное значения центрировонного вектора, $l = \min_j x'_j$ и $u = \max_j x'_j$, и делит диапазон $[l, u]$ на 2^B равных интервала, где B – количество бит на размер. Каждая координата x'_j квантизируется до ближайшего интервала:

$$q(x) = (q(x_1 - \mu_1; B, l, u), \dots, q(x_d - \mu_d; B, l, u)) = q(x'_j; B, l, u) = l + \Delta \left\lfloor \frac{x'_j - l}{\Delta} + \frac{1}{2} \right\rfloor$$

где $\Delta = \frac{u-l}{2^B - 1}$, q – функция скалярной квантизации.

2.4.9. LoRANN

Низкоранговый метод приближенных ближайших соседей / низкоранговая квантизация (Low-Rank Approximate Nearest Neighbours / Low-Rank Quantization) – это библиотека ANN, основанная на низкоранговой факторизации.

Все векторы (или набор векторов) рассматривается как матрица $X \in \mathbb{R}^{N \times D}$ (N – число векторов, D – размерность). LoRANN делает низкоранговую аппроксимацию: $X \approx UV$, где $U \in \mathbb{R}^{N \times r}$, $V \in \mathbb{R}^{r \times D}$, а r – ранг, намного меньше D. Таким образом, вместо хранения $N \times D$ чисел, хранить ся $N \times r + r \times D$, что при подходящем r даёт значительную экономию.

2.5. Инвертированные индексы

Индекс хранит в себе некоторое множество векторов базы данных, вектора можно добавлять к уже созданному индексу. При поиске индекс получает вектор запроса и находит ближайший вектор к вектору-запросу. Существует множество вариантов этой функциональности: вместо просто ближайшего соседа возвращается k ближайших соседей; вместо фиксированного числа соседей возвращаются только векторы в пределах определённого диапазона; можно выполнять параллельный поиск пакетов векторов.

2.5.1. Inverted File (IVF)

Структура инвертированного индекса [15] [14] [9], основанная на VQ, позволяет разделить пространство набора данных на k областей (ячейки Вороного), которые соответствуют k центроидам кодовой книги, то есть множество C центроидов имеет мощность k. Каждой области P_i сопоставляется ближайший центроид c_i, то есть

$$P_i = \{x \in X | c_i = argmin_{c \in C} \|x - c\|_2\}$$

Внутри каждой области P_i остаточные векторы $\{r_x = x - c_i\}_{x \in P_i}$ дополнительно квантуются с помощью PQ.

2.5.2. Inverted Multi-index (IMI)

Инвертированная многоиндексная структура [14] [3] использует идею PQ для индексации и может генерировать k^m областей с m кодовыми книгами по k подцентроидов каждый.

Следуя идее PQ, инвертированная многоиндексная структура строится в виде многомерной таблицы. Записи этой таблицы соответствуют всем возможным наборам кодовых слов из кодовых книг, соответствующих различным группам измерений. Эта многомерная таблица заменяет ”плоскую” таблицу, содержащую записи, соответствующие кодовым словам стандартной структуре инвертированного индекса. Аналогично стандартной структуре инвертированного индекса, каждая запись многоиндексной таблицы соответствует части исходного векторного пространства и содержит список точек, которые попадают в эту часть. Кроме того, при объединении векторных списков для определенного количества записей, наиболее близких к вектору запроса, создается список кандидатов, так же как и в стандартной структуре инвертированного индекса.

Есть большая коллекция $\mathbb{D}$ из N M-мерных векторов

$$\mathbb{D} = \{p_1, p_2, \dots, p_N\}, p_i \in \mathbb{R}^M$$

Конструкция стандартной структуры инвертированного индекса начинается с обучения кодовой книги $\mathbb{W}$ с K M-мерными векторами $\mathbb{W} = \{w_1, w_2, \dots, w_K\}$ с помощью алгритма k-средних. Затем изначальный набор данных разбивается на K списков $\mathbb{W}_1, \mathbb{W}_2, \dots, \mathbb{W}_K$, где каждый список $\mathbb{W}_i$ содержит все векторы, которые попадают в его ячейку Вороного в $\mathbb{R}^M$, то есть $\mathbb{W}_i = \{p \in \mathbb{D} \| i = arg\min_j d(p, w_j)\}$. Здесь d – это мера расстояния в $\mathbb{R}^M$

2.6. Графовые индексы

В данном разделе рассматриваются графовые индексы [10], обеспечивающие высокую точность поиска за счёт хранения связей между элементами. Однако это достигается ценой повышенного потребления памяти и более сложной структуры данных.

Общая идея данных индексов заключается в построении графа, в котором вершины соответствуют векторным объектам, а рёбра соединяют близкие элементы. Поиск ближайших соседей сводится к обходу графа, начиная с некоторой стартовой точки и постепенно переходя к более близким вершинам.

2.6.1. NSW

NSW (Navigable Small World, навигационный граф малого мира) [1] – однослойный граф, в котором реализованы свойства ”малого или тесного мира” (Small World). Ключевая идея данного метода заключается в создании структуры, где любой узел можно достичь за небольшое количество шагов, используя только локальную информацию о соседях.

NSW строится итеративно. Каждый новый вектор соединяется с f ближайшими соседями из числа уже существующих в графе.

Поиск в NSW реализует стратегию жадного обхода:

1. Выбирается случайная точка входа.
2. Алгоритм вычисляет расстояние от вектора-запроса q до всех соседей текущего узла.
3. Переход в ближайший к q узел.

4. Повторять процесс, пока не будет достигнут локальный минимум (ни один сосед не ближе к запросу, чем текущий узел).

Для повышения точности поиск обычно запускается несколько раз из различных случайных точек входа.

Главный недостаток базового NSW – высокая вероятность ”застревания” в локальном минимуме на ранних этапах поиска.

2.6.2. HNSW

HNSW (Hierarchical Navigable Small World, иерархический навигационный граф малого мира) [18] является одним из самых популярных и быстрых алгоритмов для поиска ближайших соседей.

HNSW – это многослойная графовая структура данных, представляющая собой иерархию слоев $\{\mathbb{L}_0, \mathbb{L}_1, ..., \mathbb{L}_{max}\}$

Каждый вектор данных вставляется в слой l с определённой вероятностью, основанной на экспоненциальном затухании с ростом номера слоя.

Слой $\mathbb{L}_0$ содержит все векторы из базы данных и обладает максимальной связностью.

Верхние слои содержат небольшое количество узлов и обеспечивают быстрый переход по уровням.

Каждому вектору при добавлении в граф случайным образом назначается максимальный уровень, на котором он присутствует. Таким образом формируется многоуровневая структура, напоминающая наслоение уровней.

Поиск в HNSW – это процесс последовательного уменьшения расстояния до вектора-запроса q:

1. Точка входа. Поиск начинается с фиксированной или случайной точки входа на самом верхнем уровне $\mathbb{L}_{max}$.
2. Жадный спуск. Алгоритм на каждом уровне графа выполняет жадный спуск, то есть переходит к соседу, расстояние до которого наименьшее от q.
3. Финальный этап. На уровне $\mathbb{L}_0$ происходит поиск с использование ограниченного числом efSearch списка кандидатов.

2.6.3. DiskANN

DiskANN [12] реализован с помощью графа Vamana. Архитектура метода опирается на две идеи:

- Сжатие векторов. Используется PQ для хранения сжатых копий векторов в RAM для быстрой оценки расстояний.
- Граф хранится на диске. Полные, высокоточные векторы и структура их связей хранятся на SSD.

Vamana – это графовая структура, похожая на NSW, но с оптимизированным радиусом связей (алгоритм соединяет узлы так, чтобы они покрывали как можно более широкие углы зрения).

Поиск в DiskANN устроен следующим образом:

1. Поиск в RAM. Сначала алгоритм ищет кандидатов в оперативной памяти, используя квантизированные с помощью PQ векторы. Данный подход позволяет быстро сузить область поиска до небольшого списка потенциальных соседей.
2. Уточнение на SSD. Алгоритм делает ограниченное количество обра-

щений к SSD, чтобы подгрузить полные векторы из списка кандидатов и вычислить точные расстояния.

3. Лучевой поиск (beam search). Вместо классического жадного поиска используется ”лучевой поиск”, который исследует сразу несколько перспективных направлений, чтобы не пропустить оптимальный узел за одно чтение блока с диска.

2.7. Существующие работы

В настоящее время существует большое количество библиотек и специализированных систем, предназначенных для хранения, индексирования и поиска векторных представлений данных. Данные решения отличаются архитектурой, поддерживаемыми алгоритмами индексирования, методами квантизации, возможностями масштабирования и требованиями к вычислительным ресурсам.

В рамках данного раздела рассматриваются наиболее распространённые библиотеки и системы, используемые для организации векторного поиска: Faiss, Bm25, Milvus, Qdrant и pgvector. Анализируются их основные возможности, преимущества и ограничения.

2.7.1. Faiss

Faiss (Facebook AI Similarity Search) [14] представляет собой библиотеку, разработанную компанией Meta AI для эффективного поиска ближайших соседей в высокоразмерных пространствах.

Библиотека ориентирована на работу с большими объёмами данных

и поддерживает как центральные, так и графические процессоры. Faiss предоставляет широкий набор алгоритмов индексирования, методов кластеризации и квантизации.

Основные возможности Faiss:

- поддержка индексов IVF, IMI, HNSW, NSG и kNN Graph;
- поддержка методов квантизации PQ, OPQ, SQ, AQ и RQ;
- реализация алгоритмов кластеризации (*k*-means, *k*-means++, spherical *k*-means);
- оптимизация вычислений для CPU (SIMD) и GPU (CUDA).

Преимущества Faiss:

- высокая производительность;
- эффективное использование памяти;
- возможность комбинирования различных методов индексирования;
- поддержка GPU-ускорения.

Недостатки Faiss:

- отсутствие встроенного механизма хранения данных;
- необходимость ручного подбора параметров индексов;
- высокие требования к памяти при использовании некоторых графовых структур.

2.7.2. Bm25

Bm25 (Best Matching 25) – эта классическая функция ранжирования для полнотекстового поиска и информационного поиска. В отличие от методов

векторного поиска, Bm25 основывается на лексическом совпадении слов запроса и документа. Она оценивает релевантность документа запросу на основе вхождений термов, при этом учитывает частоту термов в документе (TF), обратную документную частоту терма (IDF) и нормализацию по длине документа. Bm25 основан на вероятностной модели релевантности и был разработан в рамках OKAPI / City University и дальнейших работ С. Робертсона и соавторов.

При оценивании Bm25 учитывает несколько условий.

1. TF - чем чаще слово встречается, тем выше становится оценка. Это увеличение оценки нелинейно, начиная с какого-то значения влияние на результат уменьшается.
2. IDF - слова, встречающиеся во всех документах коллекции слабо влияют на оценку, в то время как редкие термы - сильно.
3. Нормализация по длине документа. Данное условие необходимо, так как более длинный документ содержит больше слов и имеет большую вероятность содержать термы запроса. Для уменьшения этого преимущества и необходима нормализация по длине документа.

Несмотря на отсутствие семантического анализа, Bm25 часто используется совместно с векторным поиском в рамках гибридных систем.

Определение 2.25. *Гибридный поиск – это подход, при котором одновременно используются методы лексического поиска и семантического (векторного) поиска.*

Результаты различных поисковых механизмов объединяются посредством суммирования оценок релевантности, взвешивания либо последую-

щего переранжирования.

2.7.3. Milvus

Milvus представляет собой распределённую векторную базу данных с открытым исходным кодом, предназначенную для хранения и поиска по большим наборам векторных данных.

Архитектура Milvus построена по принципу cloud-native и поддерживает масштабирование на несколько вычислительных узлов.

Основные возможности:

- поддержка индексов IVF, HNSW, DiskANN;
- поддержка методов квантизации SQ, PQ и OPQ;
- фильтрация по метаданным;
- поддержка гибридного поиска;
- поддержка многовекторных запросов.

Плюсы Milvus:

- Полноценная векторная база данных
- Хорошо масштабируется
- Поддерживает real-time вставку и обновление данных
- Есть фильтры, гибридный поиск и другое
- Работает в виде сервера или кластера
- Использует Faiss/HNSW/DiskANN – можно выбирать движок

Минусы Milvus:

- Более сложная настройка

- Требует отдельного сервера / кластера
- Имеет большие накладные расходы по ресурсам
- Для простых задач может быть избыточен

2.7.4. Qdrant

Qdrant – это открытая (open-source) векторная база данных / движок поиска по похожести (vector similarity search engine), предназначенный для хранения и поиска по эмбеддингам + метаданным для каждой точки. Qdrant хранит объекты в виде точек, содержащих векторное представление, идентификатор и набор метаданных. Все векторы в одной коллекции имеют одну размерность и одну метрику расстояния.

В отличие от простых библиотек ANN, Qdrant – полноценная база, с API, хранением, возможностью фильтрации по метаданным, управлением коллекциями и масштабируемостью.

Особенности Qdrant:

1. Методы индексирования и структуры:
    - Плотные (dense) векторы. Для данных векторов используется алгоритм HNSW. При добавлении точек строится граф, который ускоряет ANN поиск.
    - Разреженные (sparse) векторы. Qdrant поддерживает специальный индекс sparse-вектора для данных с большим числом нулевых компонентов (sparse-векторов).
    - Каждая точка может помимо вектора содержать метаданные. Это даёт возможность комбинировать векторный поиск и филь-

трацию / условия по атрибутам.

2. Виды квантизации:
    - SQ8
    - PQ
    - Binary / 1-bit / 2-bit Quantization
    - Rescoring (переоценка) – при квантизации можно сначала искать по сжатым векторам, получить кандидатов, а потом пересчитать точное расстояние по оригинальным векторам.
    - Гибкие настройки хранения: RAM и on-disk – можно хранить квантизированные (сжатые) векторы в RAM, а оригинальные – на диске, в зависимости от требований по производительности / памяти / диску.
3. Виды поиска:
    - ANN
    - Поиск по sparse-векторам
    - Комбинированные / гибридные запросы
    - Поиск + фильтрация по метаданным

Плюсы Qdrant:

- Гибкость – позволяет хранить и dense, и sparse-векторы
- Удобство, так как является готовой векторной базой данных с API, коллекциями, хранением, CRUD, фильтрацией, не нужно собирать вручную.
- Уменьшение потребляемой памяти за счёт квантизаций
- Масштабируемость

- Простота развёртывания и использования

Минусы Qdrant:

- Высокие накладные расходы
- Ограниченный набор индексов, основным индексом является HNSW
- Задержки, связанные с сетевым взаимодействием

2.7.5. Pgvector

Pgvector - расширение для PostgreSQL, предоставляющее возможности хранения данных в виде векторов и поиска ближайших соседей с помощью средств системы управления базами данных.

Плюсы pgvector:

- Удобство и интеграция с PostgreSQL
- Можно использовать SQL и транзакции
- Комбинирование раазличных методов (например сначала фильтруем и сортируем данные, затем уже выполняем векторный поиск)

Минусы pgvector:

- Низкая производительность по сравнению со специальными системами векторного поиска
- Низкая масштабируемость при работе с данными большого объёма

2.8. Анализ ограничений существующих решений

2.8.1. Проблема компромиссов

На данный момент не существует универсального подхода, гарантирующего высокую точность, низкое потребление памяти и высокую скорость поиска. Например, графовые индексы, такие как HNSW, обеспечивают высокую точность, но требуют большой объём оперативной памяти для хранения, тогда как IVFPQ и PQ занимают малый объём памяти и ускоряют поиск, при этом сильно теряя в точности. Таким образом, выбирая тот или иной метод индексирования, приходится соглашаться с выдвинутым компромиссом.

2.8.2. Проблема аппаратных архитектур

Различия между оснащением современных вычислительных систем тоже накладывают дополнительные условия при выборе индексирования. Например, графовые индексы эффективнее работают на центральном процессоре, в то время как методы PQ и IVFPQ - на графическом процессоре, из-за массовых параллельных вычислений расстояний.

2.8.3. Проблема фиксированного метода

Большинство существующих систем используют один способ индексирования для всего набора данных, что ограничивает систему, так как для конкретных задач может потребоваться разделение данных на отдельные сегменты, в которых используются различные способы индексирования и квантизаций.

2.8.4. Проблема масштабирования

При работе с высокоразмерными данными и большим объёмом векторов возрастают потребление памяти и время построения индексов. Для решения данной проблемы вычислительную нагрузку распределяют между различными устройствами, именно таким решение должна обладать система, эффективно работающая при векторном поиске.

2.8.5. Требования к разрабатываемой системе

Из проблем, описанных выше, можно перечислить основные качества, которые необходимы в системе векторного поиска:

- возможность настраивать соотношения между скоростью, точностью и объёмом памяти;
- поддержка распределения индексов между доступными устройствами;
- возможность комбинировать разные способы индексирования и квантизации;
- распределение вычислительной нагрузки между доступными устройствами.

С учётом сформулированных требований в рамках данной работы предлагается архитектура гибридной системы векторного поиска.

3. РАЗРАБОТАННЫЙ ПОДХОД

3.1. Постановка задачи

В рамках данной работы предлагается гибридный подход к организации векторного поиска, основанный на совместном использовании локальных индексов и глобального графового индекса.

Основная идея заключается в разделении всей базы данных на несколько сегментов или кластеров, для каждого из которых строится отдельный локальный индекс. Локальным индексом могут служить различные методы приближённого поиска, например IVF, PQ или IVFPQ.

Поверх всех векторов дополнительно строится глобальный графовый индекс HNSWPQ. Он используется для навигации по пространству данных с помощью компактных PQ-кодов.

В этих условиях необходимо:

- реализовать эффективный выбор кандидатов, по кластерам которых будет производиться поиск;
- добавить возможность параллельного выполнения вычислений;
- обеспечить баланс между скоростью поиска и точностью результата.

Таким образом, задача заключается в разработке подхода, позволяющего ограничить область поиска и эффективно организовать вычисления в условиях больших объёмов данных и различных ресурсов.

3.2. Технологический стек

В ходе анализа существующих решений были рассмотрены современные системы и методы поиска, такие как Faiss, Annoy и Bm25 в таблице 3.1, а также векторные базы данных Milvus, Qdrant и расширение pgvector для PostgreSQL. Сравнительный анализ баз данных представлен в приложении, таблица 1.

Таблица 3.1 – **Сравнительный анализ существующих алгоритмов**

Метод	**Описание**	**Тип поиска**	**Принцип**	**Когда лучше использовать**
Faiss	Библиотека с открытым исходным кодом, разработанная Facebook AI Research.	Векторный	Кластеризация и квантование	Миллиарды векторов, GPU
Annoy	Библиотека на C++ с оберткой для Python, разработанная в Spotify.	Векторный	Случайные деревья	Малые/средние данные, RAM
Bm25	Функция ранжирования документов на основе частоты запросов терминов в документе и обратной частиты документов (TF + IDF).	Текстовый	Статистика частоты слов	Поиск точных терминов

Из таблицы можно сделать вывод, что по всем свойствам больше всех подходит Faiss для реализации практической части работы. Так как для системы критически важен векторный поиск с возможностью комбинировать методы квантизаций при обработке огромного количества векторов.

В качестве средства хранения векторных данных в разработанной системе был выбран PostgreSQL с расширением pgvector. В отличие от остальных, pgvector не является отдельной системой, а представляет собой расширение реляционной базы данных, что обеспечивает тесную интеграцию с существующей инфраструктурой хранения данных. Это позволяет выполнять

сложные SQL-запросы без необходимости внедрения дополнительного сервиса.

Кроме того, в предложенной архитектуре главная нагрузка по выполнению запроса возлагается на библиотеку Faiss, что снижает требования к производительности встроенных механизмов поиска pgvector. Ключевой ролью базы данных в данном случае является надёжное хранение векторных данных и метаданные, а также получение данных для построения индекса.

Векторные базы данных, такие как Milvus, Qdrant, Chroma и Weaviate, предоставляют готовые решения для поиска, однако ограничивают гибкость в выборе и комбинировании методов индексации.

В качестве языка программирования для реализации системы был выбран rust. Rust обеспечивает производительность, сопоставимую с языками низкого уровня, такими как C И C++, при этом предоставляя современные механизмы для обеспечения безопасности памяти без исспользования сборщика мусора.

Дополнительным преимуществом является высокий уровень поддержки параллельного и конкурентного выполнения, позволяющий эффективно использовать множество различных устройст при выполнении поиска. В рамках данной работы расмотрены такие устройства как CPU и несколько GPU.

Помимо этого, rust упрощает взаимодействие с библиотекой Faiss через FFI(Foreign Function Interface).

3.3. Общая идея метода поиска

Вся база данных разбивается на кластеры, то есть на множества векторов. Для каждого кластера строится свой заданный локальный индекс.

При этом поверх всех данных из базы данных строится глобальный граф поиска HNSWPQ (HNSW – иерархический навигационный граф малого мира, PQ – Product Quantization). В узлах графа хранятся квантизированные с помощью PQ версии векторов. Данное решение было принято в связи с большим потреблением памяти при построении и хранении графа. Изначальные векторы имеют большую размерность, что существенно увеличивает затраты памяти при хранении. Кроме того, на квантизированных версиях быстрее высчитываются расстояния необходимые при создании и добавлении ребёр. Из ближайших кандидатов выбираем определённое количество и между ними появляются рёбра. Для нахождения ближайших соседей необходимо большое количество вычислений расстояний. Все описание выше изображено на рисунке 3.1

Предлагаемый подход к решению здачи приближённого поиска ближайших соседей основан на комбинировании поиска по глобальной графой структуре и по локальным индексам, построенных для отдельных кластеров данных.

Основная идея заключается в том, чтобы ограничить область поска за счёт предварительного нахождения наиболее перспективных кандидатов. Для этого используется двухуровневая система.

На первом уровне выполняется глобальный поиск по графу, построенному на основе всех векторов. На данном этапе находятся вектора-кандидаты,

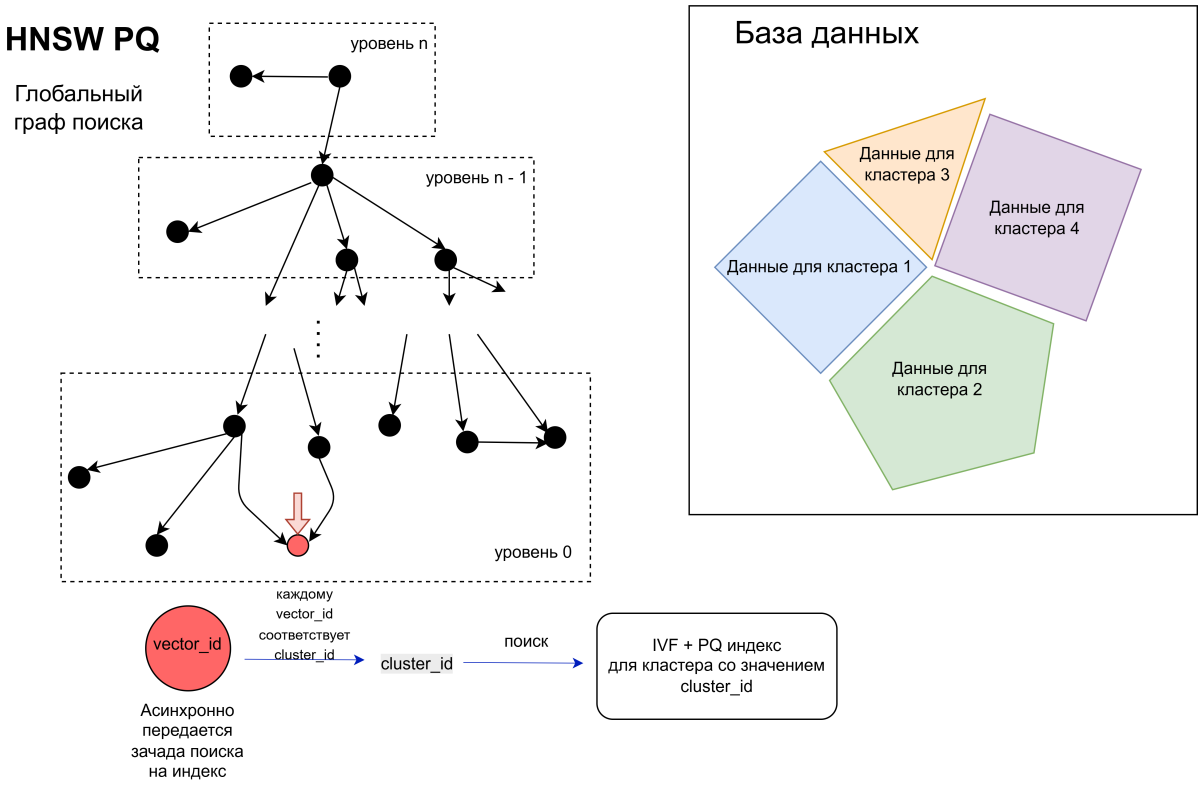

Рис. 3.1 – Схема поиска в системе

близкие к запросу. Каждый найденный кандидат связан с определённым кластером, что позволяет определить множество кластеров, потенциально содержащих ближайших соседей.

На втором уровне на выбранных кластерах выполняется поиск с использованием локальных индексов.

Ключевой особенностью подхода является то, что этапы глобального и локального поиска частично исполняются параллельно. По мере нахождения кандидатов в ходе обхода графа, соответствующие кластеры немедленно передаются на обработку, и поиск внутри них запускается асинхронно. Это позволяет эффективно использовать вычислительные ресурсы.

Кроме того, поиск по кластерам выполняется параллельно с использованием как CPU, так и GPU ресурсов, тем самым обеспечивая гибкость системы.

На завершаюшем этапе результаты поиска из различных кластеров объединяются, после чего выполняется переранжирование – из базы данных достаются изначальные вектора и происходит точное сравнение векторов, k ближайших являются результатом поиска. Данный этап позволяет компенсировать погрешности, возникающие при приближённом поиске.

Таким образом, рассмотренный метод сочетает в себе эффективную стратегию выбора области поиска и параллельную обработку данных, обеспечивающий баланс между скоростью и точностью.

3.4. Архитектура сборки системы

При создании индексов взаимодействия в системе происходят по схеме, отображённой на рисунке 3.2.

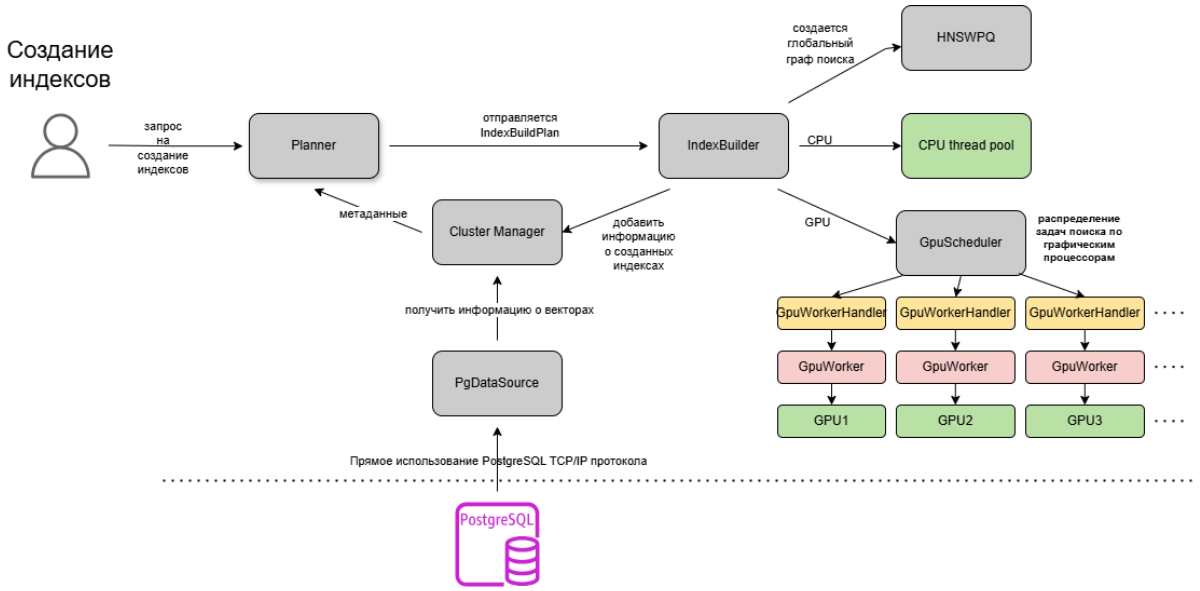

Рис. 3.2 – Создание индексов и глобального графа поиска

Как видно на рисунке, создание индексов и глобального графа происходит в несколько этапов:

1. Разделение данных на сегменты с помощью планировщика.

2. Построение локальных индексов.
3. Построение глобального графа поиска HNSWPQ.
4. Инициализация таких структур как множество потоков для CPU, GpuScheduler, а также для каждого GPU свои GpuWorker и GpuWorkerHandler.
5. Добавление информации о созданых индексах в ClusterManager.

3.4.1. Кластеризация данных с помощью планировщика

На первом шаге исходная база данных разбивается на непересекающиеся сегменты (кластеры, множества векторов). Каждый сегмент задаётся начальной позицией *start* и количеством векторов *num*.

Формально кластер можно представить как:

$$\mathbb{S}_i = \{x_{start_i}, x_{start_i+1}, \dots, x_{start_i+num_i-1}\}$$

где $\mathbb{S}_i$ - i-й кластер данных.

Для каждого сегмента далее строится отдельный локальный индекс. Это позволяет использовать различные параметры и разные методы индексирования для разных частей базы данных.

В планировщик поступают характеристики доступных устройств. В зависимости от доступной памяти, вида устройства, на данном этапе CPU или GPU, и типа индекса решается: хватает ли допустимой памяти для создания конкретного индекса.

Цель кластеризации:

- уменьшение области поиска;

- обеспечения независимости подмножества данных;
- создание условий для параллельной обработки.

3.4.2. Построение локальных индексов

Для каждого кластера выбирается конфигурация индекса, включающая:

- тип поиска;
- метод квантизации;
- устройство выполнения: CPU или GPU;
- количество векторов в кластере;
- количество памяти, выделенной на создание индекса.

Если индекс должен работать на CPU, он строится непосредственно с использованием выбранной конфигурации. Если индекс должен быть размещён на GPU, сначала создаётся CPU-версия индекса, после чего она передаётся в GpuScheduler.

Таким образом, система поддерживает гибридное выполнение, при котором разные кластеры могут обслуживаться разными устройствами.

Формально процесс можно описать так:

$$\mathbb{I}_i = BuildIndex(S_i, C_i)$$

где $\mathbb{I}_i$ – локальный индекс для сегмента S_i, а C_i – конфигурация инцекса.

Каждый индекс хранится либо в памяти CPU, либо переносится на GPU. Информация о созданном индексе сохраняется в ClusterManager.

Индексы могут быть различными, поддерживаются индексы доступные в бибилиотеке Faiss (индексы доступные на CPU, не всегда поддерживают

перенос на GPU, нужно сверять с документацией).

3.4.3. Построение глобального графа поиска HNSWPQ

Глобальный граф поиска в системе используется в качестве механизма маршрутизации, эффективно определяющего области пространства, потенциально содержащие ближайших соседей запроса.

В данной работе используется модификация графового индекса HNSW (иерархического навигационного графа малого мира), дополненная квантизированием векторов с помощью Product Quantization (PQ), что позволяет ускорить вычисления расстояний и уменьшить требования к памяти.

После построения локальных индексов выполняется обучение модели Product Quantization.

Для обучения выбираются подмножества векторов из всех сегментов. Количество обучающих векторов из каждого сегмента выбирается пропорционально размеру сегмента:

$$n_i = \frac{|S_i|}{N} \cdot T$$

где:

- n_i – количество обучающих векторов из сегмента;
- $|S_i|$ – размер сегмента;
- N – общее количесвто векторов в базе;
- T – общий размер обучающей выборки.

Для предотвращения исчезновения малых сегментов используется минимальное число обучающих элементов из каждого сегмента.

Далее каждый вектор размерности d разбивается на m подвекторов:

$$x = [x_{(1)}, x_{(2)}, \dots, x_{(m)}]$$

Для каждого подпространства обучается отдельный набор центроидов:

$$\mathbb{C}_j = \{c_{j,1}, c_{j,2}, \dots, c_{j,k}\}$$

где:

- j – номер подпространства;
- $k = 2^{nbits}$ – количество центроидов;
- $\mathbb{C}_j$ – кодовая книга для j-го подпространства.

После обучения каждый вектор может быть представлен компактным PQ-кодом:

$$code(x) = [q_1, q_2, \dots, q_m]$$

где q_j – индекс ближайшего центроида в j-м подпространстве.

После обучения PQ-модели строится глобальный графовый индекс, хранящий не исходные векторы, а их PQ-коды.

Каждой вершине графа соответствует:

- PQ-код вектора;
- идентификатор исходного вектора;
- идентификатор кластера;
- список соседей на каждом уровне графа.

Добавление нового вектора в глобальный граф состоит из следующих этапов:

1. квантизация вектора;

2. выбор уровня вершины;
3. жадный спуск по верхним уровням;
4. поиск кандидатов на нижних уровнях;
5. выбор соседей;
6. установление связей.

Более подробное описание этапов будет представлено далее.

1. Квантизация вектора

   Перед вставкой исходный вектор преобразуется в PQ-код:

$$code(x) = \boldsymbol{PQEncode}(x)$$

   Это позволяет выполнять дальнейшие операции поиска и построения графа без хранения полного вектора.

2. Выбор уровня вершины

   Для каждой новой вершины случайным образом выбирается максимальный уровень. Уровни выбираются так, что чем выше уровень, тем меньше вершин он содержит.

3. Жадный спуск по верхним уровням

   Вставка начинается из текущей вершины $entry\_point$. Рассматривая всех соседей текущей вершины, находим с наименьшим расстоянием относительно добавляемого вектора, после этого алгоритм переходит на следующий уровень графа.

4. Поиск кандидатов на нижних уровнях

После жадного спуска на уровнях, где должна размещаться новая вершина, необходимо выполнить более широкий поиск кандидатов. Для этого используется параметр $ef\_construction$, определяющий ширину поиска при построении графа.

Алгоритм поддерживает две структуры:

- очередь кандидатов для обхода;
- множество лучших найденных вершин.

Поиск продолжается до тех пор, пока существуют перспективные кандидаты, расстояние до которых меньше худшего элемента в текущем множестве лучших вершин.

5. Выбор соседей

   После получения кандидатов выбирается не более m соседей. Используется эвристика, направленная на выбор не только ближайших, но и достаточно разнесённых вершин.

   Кандидат добавляется в список соседей, если он не является слишком близким к уже выбранным вершинам. Это позволяет избежать ситуации, когда все соседи новой вершины расположены в одной локальной области.

   Условие можно описать следующим образом:

$$dist(c, s) \geq \alpha \cdot dist(c, q)$$

   где:

- c – кандидат;
- s – уже выбранная вершина;
- q – добавляемая вершина;
- α – параметр стабилизации графа.

6. Установление связей

   После выбора соседей новая вершина соединяется с нимим на соответствующем уровне. Также выполняется обратное соединение: выбранные соседи получают связь с новой вершиной.

   Если количество соседей превышает допустимое значение, удаляется наиболее удалённая вершина.

   На нулевом уровне допускается больше соседей: $M_{max} = 2M$

   На верхних уровнях используется ограничение: $M_{max} = M$

Использование HNSW с PQ обеспечивает:

- быстрое приближённое вычисление расстояний;
- эффективную навигацию по пространству данных;
- компактное хранение информации;
- возможность использования графа в качестве маршрутизатора.

При этом граф не используется для получения окончательных результатов, а служит для получения кандидатов и определения перспективных кластеров.

3.4.4. Инициализация структур

На данном этапе инициализируются такие структуры как:

- множество потоков для CPU
- разпределитель задач для GPU (GpuScheduler)
- структуры необходимые для работы с GPU (GpuWorkerHandler – очередь задач для каждого GPU, GpuWorker – исполнитель задач для каждого GPU)

3.4.5. Псевдокод построения

Далее приведён алгоритм 3.1 построения гибридной системы векторного поиска на основе локальных индексов и глобального графа HNSW-PQ.

Алгоритм 3.1. *Построение гибридной системы HNSW-PQ.*

1. *Инициализировать множество кластеров* $Clusters \leftarrow \varnothing$;
2. *Инициализировать множество сегментов* $Segments \leftarrow \varnothing$;
3. *Для каждой конфигурации индекса* $config$:

   3.1 *Создать сегмент данных* $segment$;

   3.2 *Построить локальный индекс:*

   $$index \leftarrow BuildLocalIndex(segment, config)$$

   3.3 *Зарегистрировать индекс и получить идентификатор кластера:*

   $$cluster\_id \leftarrow RegisterIndex(index)$$

   3.4 *Добавить пару* $(cluster\_id, segment)$ *в множество* $Clusters$;

3.5 Добавить сегмент в множество $Segments$;

4. *Собрать обучающую выборку:*

$$Samples \leftarrow CollectTrainingSamples(Segments)$$

5. *Обучить модель Product Quantization:*

$$PQ \leftarrow TrainPQ(Samples)$$

6. *Создать глобальный граф:*

$$G \leftarrow CreateHNSWPQ(PQ)$$

7. *Для каждого сегмента* $segment \in Segments$:

7.1 Загрузить векторы сегмента:

$$V \leftarrow LoadVectors(segment)$$

7.2 Выполнить PQ-квантизацию:

$$Codes \leftarrow EncodeBatchPQ(V)$$

7.3 Для каждого PQ-кода выполнить вставку вершины в граф:

$$InsertEncoded(G, code)$$

8. *Вернуть построенный граф* G.

□

Далее приведён алгоритм 3.2 добавления новой вершины в глобальный граф HNSW-PQ.

Алгоритм 3.2. *Добавление вершины в граф HNSW-PQ.*

1. *Выполнить PQ-квантизацию нового вектора:*

$$code \leftarrow PQEncode(x)$$

2. *Сгенерировать уровень новой вершины:*

$$level \leftarrow RandomLevel()$$

3. *Создать новую вершину графа;*
4. *Если граф пуст:*
   - 4.1 *Назначить новую вершину входной точкой графа;*
   - 4.2 *Завершить алгоритм.*
5. *Инициализировать текущую вершину:*

$$current \leftarrow entry\_point$$

6. *Выполнить жадный спуск по верхним уровням графа:*
   - 6.1 *Для каждого уровня от максимального до* $level + 1$*:*
     - 6.1.1 *Выполнить переход к ближайшему соседу.*
7. *Для каждого уровня от* $level$ *до* 0*:*
   - 7.1 *Выполнить расширенный поиск кандидатов:*

$$Candidates \leftarrow SearchLayer(current)$$

   - 7.2 *Выбрать оптимальных соседей:*

$$Neighbors \leftarrow SelectNeighbors(Candidates)$$

7.3 Добавить связи между вершинами;

8. Если уровень новой вершины больше максимального уровня графа:

8.1 Обновить входную точку графа.

□

3.5. Архитектура поиска в системе

Этап поиска выполняется для каждого входного запроса. При поиске взаимодействия в системе происходят по схеме, отображённой на рисунке 3.3 и рисунке 3.4.

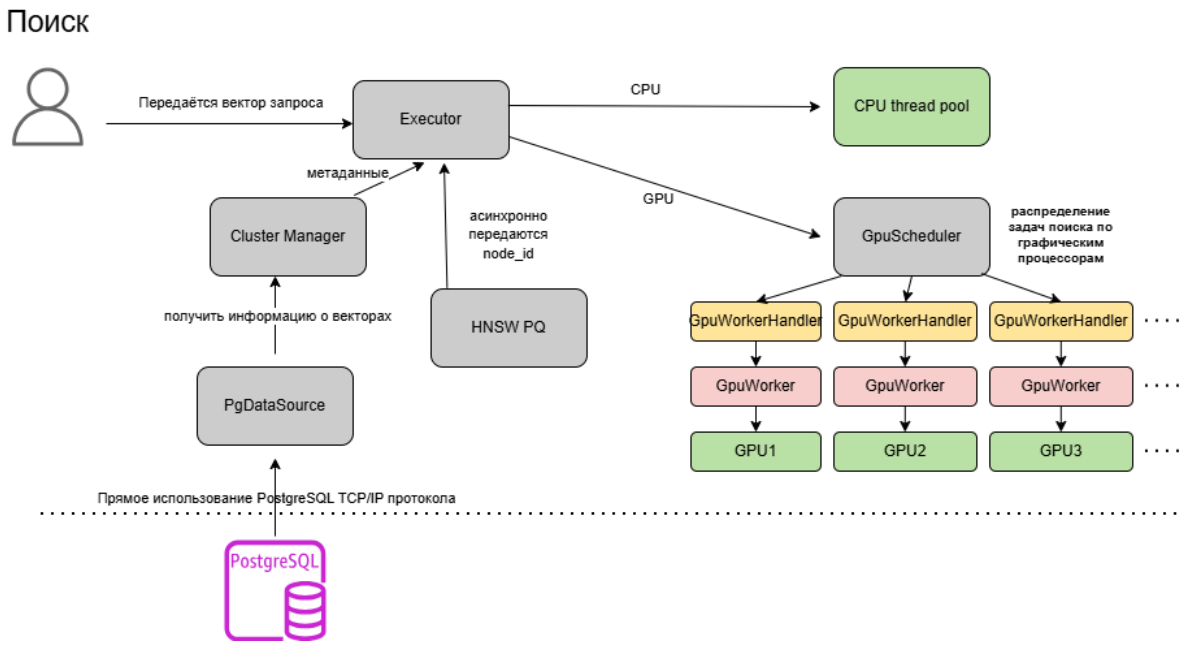

Рис. 3.3 – Архитектура поиска

Из рисунков можно сделать выводы, поиск происходит в несколько этапов:

1. Глобальный поиск кандидатов;
2. Распределение задач поиска в Dispatcher;
3. Объединение результатов (merge);
4. Переранжирование (rerank).

Перед началом поиска для запроса строится таблица расстояний LUT.

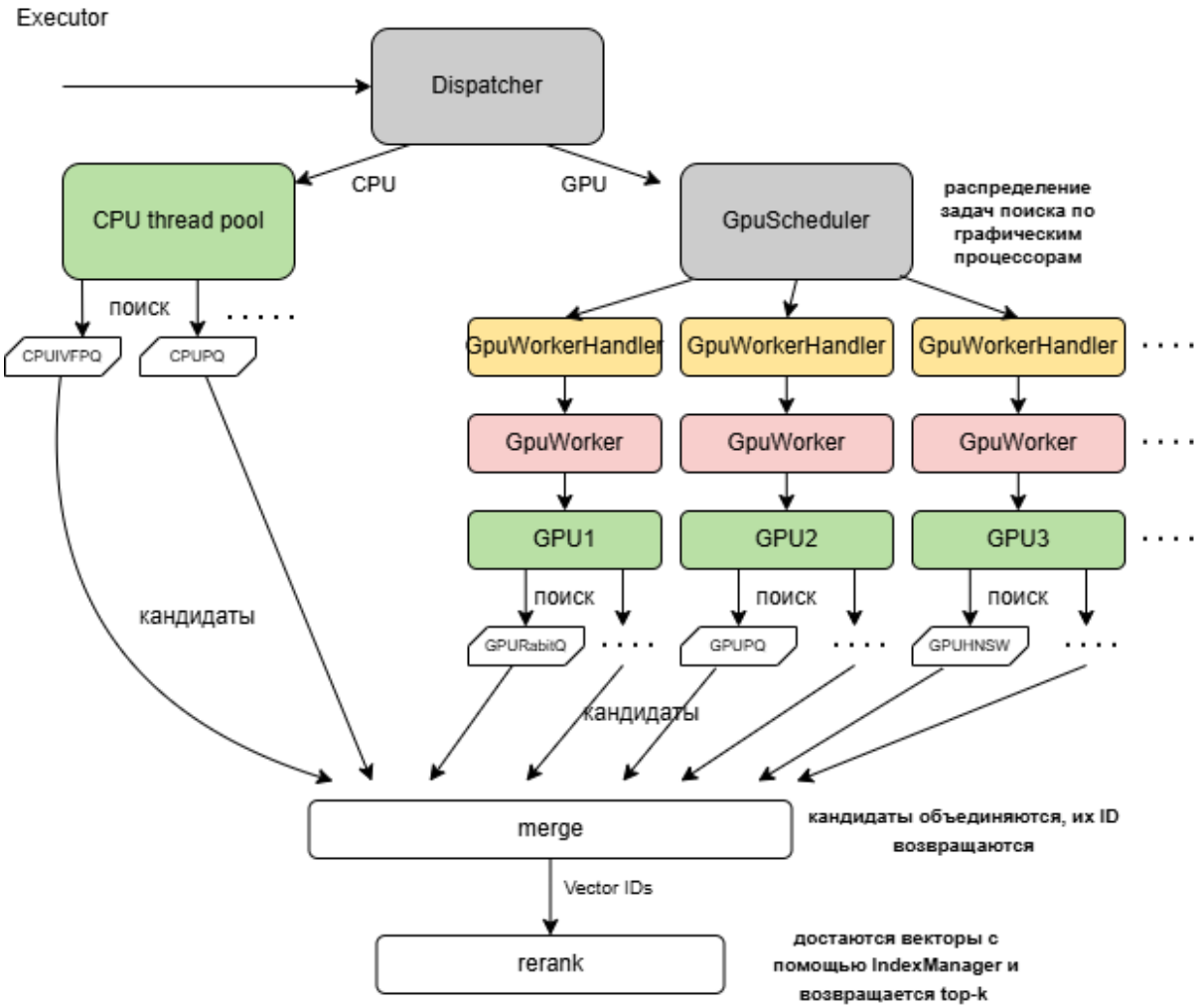

Рис. 3.4 – Схема асинхронного поиска в системе

Для каждого подпространства вычисляется расстояние между соответствующим подвектором запроса и всеми центроидами PQ-кодовой книги:

$$LUT_j[t] = dist(q_{(j),c_{j,t}})$$

Тогда расстояние между запросом и PQ-кодом вектора вычисляется как:

$$dist(q, code(x)) = sum_{j=1}^{m} LUT_j[code_j]$$

Это позволяет быстро оценивать расстояния без восстановления исходных векторов.

3.5.1. Глобальный поиск кандидатов

Глобальный поиск в системе выполняется за счёт использования графовой структуры HNSWPQ и предназначен для определения векторов-кандидатов близких к запросу.

В предлагаемом подходе используется модификация стандартного алгоритма поиска, основанная на асинхронной обработке кандидатов.

В классическом варианте HNSW сначала формируется полный набор кандидатов, после чего выбираются лучшие из них. В данной системе кандидаты обрабатываются по мере их обнаружения в процессе обхода графа.

Во время поиска поддерживаются два множества:

- очередь кандидатов для обхода;
- текущее множество лучших элементов.

Для каждого посещённого вектора рассматриваются его соседи, для кото-

рых вычисляются расстояния до запроса.

Если вектор удовлетворяет условиям попадания в текущее множество ближайших векторов, то он становится кандидатом и информация о нём немедленно передаётся в Dispatcher для дальнейшей обработки.

3.5.2. Распределение задач поиска в Dispatcher

По мере появления кандидатов, в Dispatcher передаются идентификаторы кластеров. С их помощью Dispatcher достает из ClusterManager дополнительную информацию.

Диспетчер принимает решение о запуске поиска в кластерах на основе поступающих кандидатов. Для этого используются следующие критерии:

- минимальное количество попаданий;
- предотвращение повторной обработки одного кластера.

Далее немного подробнее про эти критерии:

- Минимальное количество попаданий

  В процессе работы Dispatcher учитывает количество кандидатов, относящихся к каждому кластеру. Для этого используется счётчик попаданий. Кластер будет обрабатываться после того, как значение счетчика станет больше либо равным пороговому значению (изначально заданному).

- Предотвращение повторной обработки одного кластера

  Для каждого кластера поиск осуществляется не более одного раза. Dispatcher хранит множество уже обработанных кластеров и при повторном появлении кандидатов из того же кластера не запускает

новый поиск на соответствующему локальному индексу.

Это позволяет избежать дублирования вычислений и снизить нагрузку на систему.

3.5.3. Локальный поиск на кластерах

Для выбранных кластеров запускается поиск с ипользованием локальных индексов. Dispatcher с помощью ClusterManager определяет где должен обрабатываться кластер (на CPU или на GPU. Если на GPU, то какой номер у этого GPU). Таким образом, осуществляется следующее:

- задачи для CPU исполняются в множестве потоков;
- задачи для GPU передаются в GpuScheduler, который по номеру GPU дальше пересылает задачу в очередь для указанного GPU (в GpuWorkerHandler).

3.5.4. Объединение результатов (merge)

Результаты выполнения задач передаются с использование каналов. Каждая задача после завершения отправляет результат обратно в основной поток. Сообщения могут содержать либо результат поиска либо информацию об ошибке.

Результаты поиска поступают асинхронно и объединяются в общий список.

3.5.5. Переранжирование (rerank)

Для полученных кандидатов достаются исходные векторы, после чего вычисляется точное расстояние до запроса по евклидовой метрике. Результаты сортируются, и выбираются k ближайших соседей. Этап переранжирования компенсирует ошибки приближённого поиска, тем самым повышая точность результата.

3.5.6. Псевдокод поиска

Далее приведён алгоритм 3.3 поиска ближайших соседей в графе HNSW-PQ.

Алгоритм 3.3. *Поиск ближайших соседей в HNSW-PQ.*

1. *Построить таблицу расстояний:*

$$LUT \leftarrow ComputeLUT(query)$$

2. *Инициализировать текущую вершину:*

$$current \leftarrow entry\_point$$

3. *Для каждого уровня от максимального до первого:*

   3.1 Выполнить жадный поиск ближайшего соседа.

4. *На нулевом уровне выполнить расширенный поиск:*

$$Candidates \leftarrow SearchLayer(current, ef\_search)$$

5. *Определить наиболее релевантные кластеры;*

6. *Для каждого выбранного кластера:*

    6.1 *Выполнить локальный поиск внутри соответствующего индекса.*

7. *Объединить и отсортировать найденные результаты;*
8. *Осуществить переранжирование на найденных векторах;*
9. *Вернуть k ближайших соседей.*

□

4. РЕЗУЛЬТАТЫ

4.1. Конфигурация тестового стенда

Апробация проводилась на вычислительной системе со следующими характеричтиками:

- операционная система: Ubuntu Linux;
- CPU: Intel Core i7-2600;
- количество физических ядер: 4;
- количество логических ядер: 8;
- тактовая частота процессора: до 3.80ГГц;
- RAM: 32 ГБ DDR3;
- GPU: NVIDIA GeForce RTX 4060;
- объём видеопамяти: 8 ГБ.

4.2. Виды базы данных

Для проведения экспериментов использовались различные базы данных:

1. Пример базы данных
2. GIST1M

4.2.1. Пример базы данных

Для проведения экспериментов использовалась база данных, содержащая 100.000 векторов размерности 64.

Векторы генерировались случайным образом и нормализовались к единичной длине. Для хранения данных использовалось расширение pgvector для PostgreSQL.

Создание таблицы выполнялось следующим SQL-запросом:

Листинг 4.1: Создание таблицы векторов

```
CREATE TABLE vectors {
  id BIGSERIAL PRIMARY KEY,
  embedding vector(64)
};
```

Листинг 4.2: Функция генерации случайного вектора

```
CREATE OR REPLACE FUNCTION random_unit_vector(dim int)
RETURNS vector
LANGUAGE plpgsql AS
DECLARE
  arr float4[];
  norm float8;
BEGIN
  SELECT array_agg(random())
  INTO arr
  FROM generate_series(1, dim);


  SELECT sqrt(sum(x * x)) INTO norm
  FROM unnest(arr) x;


  RETURN (SELECT array_agg(x / norm) FROM unnest(arr) x)::vector;
END;
;
```

Листинг 4.3: Заполнение таблицы

```
INSERT INTO vectors (id, embedding)
SELECT gs - 1, random_unit_vector(64)
FROM generate_series(1, 100000) gs;
```

Параметры тестирования

Во всех экспериментах использовались параметры глобального графа, приведённые в таблице 4.1.

Таблица 4.1 – Параметры глобального графа HNSW-PQ

Параметр	Значение
$graph\_k$	64
$max\_per\_cluster$	1
$local\_k$	1000
min_hits	1
$distance\_threshold$	100.0
$early\_distance\_threshold$	1.0
max_level	16
m	16
$ef\_construction$	32
ef_search	32
k	50

query = [0.5; 64]

CPU-PQ индекс (один сегмент)

Таблица 4.2 – CPU-PQ индекс (один сегмент)

m	nbits	Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
4	8	44.3	37.8	10.6	22.2	0.14
8	8	88.5	38.6	10.7	26.7	0.14
16	8	**173.7**	**38.7**	**10.8**	**28.7**	**0.96**
32	8	343.9	38.7	10.8	37.0	1.0
4	4	3.7	38.2	10.8	31.7	0.02
8	4	5.6	38.2	10.7	35.8	0.0
16	4	9.9	38.4	10.8	64.4	0.06
32	4	**18.3**	**38.7**	**10.8**	**79.8**	**0.92**

Из результатов тестирования можно сделать выводы, что при увеличении параметра m происходит ощутимый рост точности поиска, так например при использовании параметров $m = 4$ и $nbits = 8$ точность равна 0.14, в то время как при $m = 16$ и таком же значении $nbits$ точность равна 0.96. При увеличении значения параметра m до 32 точность становится равна 1.0, однако время построения данного индекса и дальнейший поиск с его помощью слишком увеличилось. Если же изменять параметр $nbits$, а именно сделать его равным 4, то снижается качество результатов, так как уменьшается количество центроидов в кодовых книгах.

Изменяя параметры m и $nbits$, можно заключить, что наилучший компромисс между точностью, скоростью и потреблением памяти достигается при $m = 16$ и $nbits = 8$.

GPU-PQ индекс (один сегмент)

Перенос индекса с центрального процессора на графический практически не повлиял на точность поиска, но позволил ускорить поиск на пару миллисекунд. Из-за накладных расходов время построения индекса на GPU увеличилось.

CPU-RaBitQ индекс (один сегмент)

Таблица 4.3 – CPU-RaBitQ индекс (один сегмент)

nbits	Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
4	3.8	39.2	10.9	38.9	1.0
6	6.2	38.8	10.9	38.2	1.0
8	15.3	38.8	10.8	35.1	1.0

По результатам тестирования видно, что метод RaBitQ даже при малом количестве бит квантизации имеет высокую точность. Во всех значениях параметров была достигнута точность 1.0. При этом время построения индекса по сравнению например с PQ остаётся ниже. Увеличение параметра *nbits* повлияло на время построения индекса, однако на предоставленных данных нельзя заметить как меняется точность.

CPU-IVF индекс (один сегмент)

Таблица 4.4 – CPU-IVF индекс (один сегмент)

x	nprobe	Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска CPU (ms)	Время поиска GPU (ms)	Точность
128	8	2.0	38.9	10.9	26.2	24.1	0.24
256	8	2.6	39.0	11.0	25.3	26.8	0.4
512	8	3.5	38.6	10.8	27.1	25.6	0.5
128	16	2.0	38.8	10.8	31.8	25.8	0.52
256	16	2.5	38.8	10.8	28.1	30.1	0.62
512	16	3.6	38.7	10.8	36.5	27.5	0.6
128	32	1.9	39.1	11.0	41.0	30.8	0.7
256	32	2.5	39.4	11.1	39.6	30.4	0.76
512	32	3.6	38.8	10.9	35.1	33.3	0.78

Из тестов, описанных выше, можно заметить, что увеличение параметров x, отвечающего за число кластеров IVF, и *nprobe* повышает точность поиска. На малых значениях параметра *nprobe* просматривается недостаточное количество списков, из-за чего точность поиска снижается, в то время как на $nprobe = 32$ достигается точность 0.78 и возрастает время поиска. Результат при $nprobe = 32$ является наилучшем в приведённых сериях испытаний.

GPU-IVF индекс (один сегмент)

Также как в случае с GPU-PQ точность поиска не изменилась, скорость поиска уменьшается при увеличении *npbore* за счёт распараллеливания вычислений расстояний, возросло только время построения индекса из-за

накладных расходов.

CPU-IVFPQ индекс (один сегмент)

Таблица 4.5 – CPU-IVFPQ индекс (один сегмент)

x	nprobe	m	nbits	Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска CPU (ms)	Время поиска GPU (ms)	Точность
128	8	8	8	93.7	39.3	10.9	22.5	23.8	0.12
256	8	8	8	92.6	38.9	10.8	20.9	23.5	0.32
512	8	8	8	93.0	39.0	10.8	23.2	23.7	0.28
128	16	8	8	91.9	39.0	10.8	30.0	24.6	0.24
256	16	8	8	94.9	39.3	10.9	23.9	23.7	0.44
512	16	8	8	93.6	39.1	10.8	25.9	23.8	0.34
128	32	8	8	94.6	39.4	10.9	30.1	25.8	0.22
256	32	8	8	95.5	39.4	11.0	27.8	24.9	0.44
512	32	8	8	94.0	39.1	10.7	25.2	29.4	0.36
128	8	16	8	185.6	39.5	10.9	25.9	24.9	0.24
256	8	16	8	186.8	39.6	10.9	23.4	22.7	0.4
512	8	16	8	184.3	39.1	10.8	26.1	23.9	0.5
128	16	16	8	182.3	39.2	11.0	24.1	25.7	0.46
256	16	16	8	187.3	39.6	11.0	26.1	27.6	0.58
512	16	16	8	184.1	39.3	10.8	27.4	26.8	0.54
128	32	16	8	183.4	39.2	10.9	31.8	29.9	0.6
256	32	16	8	185.1	39.1	10.9	30.6	31.9	0.66
512	32	16	8	185.2	39.2	10.9	33.4	34.0	0.66
128	8	32	4	19.9	39.2	11.2	31.2	55.3	0.24
256	8	32	4	20.5	39.5	11.0	32.6	30.0	0.38
512	8	32	4	21.8	39.8	11.1	34.3	35.0	0.46
128	16	32	4	20.0	39.6	11.1	41.5	38.1	0.48

x	nprobe	m	nbits	Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска CPU (ms)	Время поиска GPU (ms)	Точность
256	16	32	4	20.3	39.2	11.0	40.3	44.2	0.6
512	16	32	4	21.3	39.4	11.0	44.1	40.5	0.56
128	32	32	4	20.0	39.4	11.1	59.1	51.0	0.6
256	32	32	4	21.2	39.8	11.0	52.1	52.3	0.7
512	32	32	4	21.6	39.5	11.1	60.7	49.7	0.74

Из результатов тестирования можно сделать вывод, что при увеличении параметра *nprobe* улучшается точность поиска. Такая же тенденция видна при увеличении параметров m и $nbits$, так например при $m = 8$ точность горазда хуже чем при $m = 16$ или $m = 32$, с параметром $nbits = 4$ точность хуже, чем при $nbits = 8$. Время построения индексов меньше при меньших значениях параметров.

GPU-IVFPQ индекс (один сегмент)

Результаты GPU-реализации IVFPQ показали, что использование графического процессора позволяет сократить время выполнения поиска без ухудшения качества результата. Наиболее заметное ускорение достигается при обработке больших наборов кандидатов, поскольку вычисления расстояний между PQ-кодами хорошо подходят для параллельного выполнения на GPU. При этом время построения индекса увеличивается из-за затрат на перенос структуры индекса и PQ-кодов в видеопамять.

CPU-IVFRabitQ индекс (один сегмент)

Таблица 4.6 – CPU-IVFRabitQ индекс (один сегмент)

x	nprobe	nbits	Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
128	8	4	5.4	40.0	10.9	27.8	0.24
256	8	4	5.9	39.6	11.0	29.2	0.4
512	8	4	7.2	39.4	11.2	34.4	0.5

Результаты показывают, что комбинирование IVF и RabitQ обеспечивает такое же качество поиска, что и классический IVF. При этом использование RabitQ приводит к увеличению времени построения локального индекса из-за дополнительного этапа квантизации. Увеличение числа кластеров IVF положительно влияет на точность поиска, однако сопровождается ростом времени поиска.

CPU-HNSW индекс (один сегмент)

Таблица 4.7 – CPU-HNSW индекс (один сегмент)

x	efSearch	Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
16	128	20.0	40.2	11.1	23.5	0.76
32	128	36.1	40.0	11.2	26.6	0.86

x	efSearch	Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
64	128	38.2	40.4	11.1	30.5	0.92
128	128	38.8	40.0	11.2	34.4	0.94
16	256	19.7	40.3	11.1	32.0	0.88
32	256	36.9	40.4	11.1	30.8	0.96
64	256	38.5	40.3	11.2	42.9	0.96
128	256	39.4	40.6	11.2	42.0	0.98
16	512	**19.6**	**40.3**	**11.3**	**37.2**	**1.0**
32	512	36.9	40.3	11.1	41.9	1.0
64	512	38.3	40.2	11.0	53.2	0.98
128	512	38.8	40.0	11.3	60.0	1.0

По результатам видно, что графовый индекс HNSW имеет лучшие показатели точности среди всех представленных ранее методов. При увеличении параметра efSearch происходит улучшение точности поиска, так как алгоритм просматривает большее число кандидатов на одном уровне во время обхода графа. Наилучший результат, а именно точность равная 1.0, достигнут при значении параметра $efSearch = 512$. Тем не менее, увеличение параметров графа влечёт увеличение затрат памяти на хранение и времени поиска.

CPU-LSH индекс (один сегмент)

Индекс LSH на используемом наборе данных показал очень низкую точность, в многих конфигурациях точность оставалась на значениях близких к нулю. Послужить данной проблеме может высокая плотность и случайность распределения векторов в пространстве признаков.

CPU-HNSW + GPU-IVFPQ (два сегмента)

Таблица 4.8 – CPU-HNSW + GPU-IVFPQ (два сегмента)

x-hnsw	efSearch	x-ivf	nprobe	m	nbits
32	512	256	16	16	8
32	512	256	32	16	8
32	512	256	64	16	8

Таблица 4.9 – CPU-HNSW + GPU-IVFPQ (два сегмента)

Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
16.8 + 191.2	41.7	129.6	51.8	0.76
16.8 + 191.4	41.6	120.5	52.8	0.88
17.0 + 193.0	41.6	127.4	56.9	0.92

Из результатов комбинирования методов CPU-HNSW и GPU-IVFPQ можно сделать вывод, что использование различных индексов при поиске оправдано. Глобальный граф HNSW обеспечивает эффективную навигацию по пространству данных, после чего локальный поиск выполняется с использованием GPU-ускоренного IVFPQ. Несмотря на использование двух различных индексов, итоговое качество поиска ограничено точностью локального IVF-индекса. Полученные результаты показывают, что комбинирование различных методов индексирования позволяет распределять нагрузку между CPU и GPU, настраивая систему под различные требования к производительности и объёму памяти.

4.2.2. GIST1M

В данном разделе представлены результаты тестирования разработанной системы на наборе данных GIST1M, содержащем 1 миллион векторов высокой размерности.

Для оценки качества работы использовались запросы из файла `gist_query.fvecs`. В качестве метрик рассматривались:

- время построения локальных индексов;
- время обучения PQ-модели глобального графа;
- время построения графовой структуры;
- время выполнения поиска;
- точность поиска.

Эксперименты проводились отдельно для наборов из 10 и 1000 запросов.

Параметры тестирования

Для тестирования 10 запросов использовались параметры глобального графа, приведённые в таблице 4.10.

Таблица 4.10 – Параметры глобального графа HNSW-PQ

Параметр	Значение
graph_k	100
max_per_cluster	1
local_k	128
min_hits	1

Параметр	Значение
$distance\_threshold$	100.0
$early\_distance\_threshold$	1.0
max_level	12
m	96
$ef\_construction$	64
ef_search	64
k	50

Параметр $graph\_k$ определяет количество кандидатов, рассматриваемых глобальным графом, $local\_k$ задаёт число результатов локального поиска, а параметры $ef\_construction$ и ef_search определяют качество построения и поиска в графовой структуре HNSW-PQ.

В качестве локальных индексов использовалась гибридная схема, включающая HNSW на CPU и IVF на GPU.

Результаты на 10 запросах.

CPU-HNSW + GPU-IVF (два сегмента)

Таблица 4.11 – CPU-HNSW + GPU-IVF (два сегмента) на 10 запросах

x-hnsw	efSearch	x-ivf	nprobe
16	128	256	16
16	256	256	16
32	256	256	16
32	256	256	32

Таблица 4.12 – CPU-HNSW + GPU-IVF (два сегмента) на 10 запросах. Продолжение

Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
373.0 + 69.4	3181.1	21732.1	250.6 – 443.8	0.62 – 0.96
365.0 + 68.3	3133.8	20308.0	223.4 – 416.2	0.7 – 0.97
1245.7 + 68.2	3097.1	19393.8	267.7 – 427.1	0.82 – 1.0
1221.8 + 67.2	3051.3	18452.9	326.3 – 621.8	0.86 – 1.0

Вывод по результатам для 10 запросов

Результаты показывают, что увеличение параметров локальных индексов приводит к росту точности поиска. При переходе от конфигурации x-HNSW = 16, $efSearch = 128$ к конфигурации x-HNSW = 32, $efSearch = 256$ точность увеличилась с диапазона $0.62-0.96$ до $0.86-1.0$.

При этом увеличение точности сопровождается ростом времени построения индексов и времени поиска. Наибольшее влияние на время работы оказало увеличение числа соседей HNSW и параметра *nprobe*, приводящее к расширению области поиска.

Таким образом, для небольшого числа запросов гибридная схема демонстрирует возможность достижения высокой точности при сохранении приемлемого времени ответа.

Для тестирования 1000 запросов использовались параметры глобального графа, приведённые в таблице 4.13.

Результаты на 1000 запросах.

Таблица 4.13 – Параметры глобального графа HNSW-PQ

Параметр	Значение
graph_k	200
max_per_cluster	1
local_k	256
min_hits	1
distance_threshold	100.0
early_distance_threshold	1.0
max_level	16
m	120
ef_construction	64
ef_search	64
k	100

CPU-HNSW + GPU-IVF (два сегмента)

Таблица 4.14 – CPU-HNSW + GPU-IVF (два сегмента)на 1000 запросах

x-hnsw	efSearch	x-ivf	nprobe
16	128	256	16
16	256	256	16
32	256	256	16
32	256	256	32

Таблица 4.15 – CPU-HNSW + GPU-IVF (два сегмента) на 1000 запросах. Продолжение

Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
338.0 + 65.3	2810.3	15259.7	140.3 – 547.6	0.35 – 1.0
344.8 + 65.8	2846.4	15086.2	136.9 – 549.8	0.41 – 1.0
1169.9 + 65.6	2887.6	16308.9	128.4 – 568.5	0.64 – 1.0

Время создания лок. индексов (sec)	Время тренировки PQ-модели графа (sec)	Время вставки векторов в граф (sec)	Время поиска (ms)	Точность
1185.4 + 66.7	2948.2	16685.1	173.8 – 763.1	0.72 – 1.0

Вывод по результатам для 1000 запросов

При увеличении числа запросов наблюдается более выраженное влияние параметров индексирования на качество поиска.

Точность увеличивается от диапазона $0.35-1.0$ до $0.72-1.0$, однако одновременно возрастает время поиска и вычислительная нагрузка на систему.

Кроме того, можно заметить увеличение времени построения локальных индексов при увеличении параметра $efSearch$ у индекса HNSW. Время обучения PQ-модели практически не изменяется, так как оно зависит в основном только от размера обучающих данных.

4.3. Выводы по результатам экспериментов

Проведённая серия испытаний показала, что система способна предоставить варианты различных компромиссов между такими характеристиками как точность поиска, скорость выполнения запроса и потреблением памяти.

Было выявлено, что лучшая точность поиска у графового индекса HNSW, однако использование данного метода сопровождается значительным потреблением памяти и увеличением времени построения индекса, как видно из таблицы 4.7. Методы квантизации PQ и IVFPQ сильно уменьшают объём требуемой памяти, из-за чего ухудшается точность, таблицы 4.2, 4.5.

Эксперименты также подтвердили эффективность при использовании графических процессоров для методов квантизации, так как вычисление расстояний можно распараллелить. Из результатов видно, что комбинирование различных методов индексаций и квантизаций эффективно, благодаря чему систему можно подстраивать под требования конкретных задач и доступные вычислительные ресурсы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы была разработана гибридная система векторного поиска. Были исследованы различные способы организации поска ближайших соседей, включая такие методы как графовые индексы и методы квантизаций. В работе реализован подход, основанный на совместном использовании глобального индекса HNSW-PQ и локальных индексов, каждый из которых соответствует определённому сегменту данных, эти сегменты не пересекаются. Предложенная архитектура разделяет процесс поиска на два уровня: глобальную навигацию, представляющую собой поиск кандидатов по графу, и локальный поиск по индексам, выбранным из поска по графу. В глобальном графе использовался метод квантизации PQ, что позволило уменьшить объём памяти, необходимый для хранения данного подхода, так как вместо исходных векторов в вершины графа содержатся компактные PQ-коды.

В работе были реализованы:

- механизм построения локальных индексов с поддержкой различных методов поиска и квантизации;
- глобальный графовый индекс HNSW-PQ, асинхронно обрабатывающий локальные индексы;
- поддержка параллельного выполнения поиска на CPU и GPU;
- механизм поиска, использующий различные типы индексов.

Система была реализована на языке Rust с использованием библиотеки Faiss и базы данных PostgreSQL. Архитектура допускает использование различных типов индексов, поддерживаемых библиотекой Faiss, для каждого

сегмента данных, при этом предоставляем возможность переноса индекса с CPU на GPU, если он поддерживается Faiss.

Следовательно цель выпускной квалификационной работы достигнута, а именно разработана система векторного поиска с возможностью комбинирования различных способов индексации и квантизации.

В качестве направлений дальнейших работ можно выделить следующие цели:

- Добавить возможность пересоздания индексов.
- Реализовать сохранение и загрузку индексов из файла.
- Улучшить планировщик, распределяющий задачи между устройствами.
- Улучшить GPU-планировщик, а именно учитывать текущую нагрузку на доступные GPU.
- Поддерживать пакетную обработку запросов.
- Оптимизировать алгоритм построения глобального графа.
- Автоматически подбирать параметры индексов, основываясь на свойствах данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Список литературы

1. Alexander Ponomarenko A.L. Yury Mal'kov / Approximate nearest neighbor search small world approach. 2011.
2. Babenko A. L.V. / Additive quantization for extreme vector compression. 2014.
3. Babenko A. L.V. / The inverted multi-index. 2015.
4. Cecilia Aguerrebere M.H. Ishwar Singh Bhati / Similarity search in the blink of an eye with compressed indices. 2023.
5. Fabien Andre N.L.S. Anne-Marie Kermarrec / Quicker adc: Unlocking the hidden potential of product quantization with simd. 2019.
6. Herve Jegou M.D. Romain Tavenard / Searching in one billion vectors: re-rank with source coding. 2011.
7. Hui Li X.Y. Shiyuan Deng / Saq: Pushing the limits of vector quantization through code adjustment and dimension segmentation. 2025.
8. James Jie Pan G.L. Jianguo Wang / Survey of vector database management systems. 2024.
9. Julieta Martinez J.J.L. Holger H. Hoos / Stacked quantizers for compositional vector compression. 2014.

10. Mengzhao Wang Q.Y. Xiaoling Xu / A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. 2021.
11. Shicong Liu H.L. Junru Shao / Generalized residual vector quantization for large scale data. 2016.
12. Suhas Jayaram Subramanya R.K. Devvrit / Diskann: Fast accurate billion-point nearest neighbor search on a single node. 2019.
13. Tiezheng Ge Q.K. Kaiming He / Optimized product quantization. 2013.
14. Wei Chen F.Z. Jincai Chen / Vector and line quantization for billion-scale similarity search on gpus. 2019.
15. Xiang Wu S.K. Ruiqi Guo / Local orthogonal decomposition for maximum inner product search. 2019.
16. Xinyan Dai K.K.W.N. Xiao Yan / Norm-explicit quantization: Improving vector quantization for maximum inner product search.
17. Yongjian Chen C.W. Tao Guan / Approximate nearest neighbor search by residual vector quantization. 2010.
18. Yu. A. Malkov D.A.Y. / Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. 2016.

ПРИЛОЖЕНИЕ

П.1. Первая глава приложения

Таблица П.16 – Сравнительный анализ существующих векторных баз данных

Метод	**Описание**	**Плюсы**	**Минусы**
Chroma	Современная векторная векторная база данных с открытым исходным кодом, ориентированная на простоту и разработку приложений с использованием LLM.	• простота установки и эксплуатации • встроенная векторизация	• ограниченность мощностью одной машины
Pinecone	Полностью облачная (SaaS) векторная база данных.	• работает через API • стабильно низкая задержка(latency) • быстрая работа с фильтрами в реальном времени	• закрытый код • нельзя запустить локально без интернета или в закрытом контуре

pgvector	Расширение для базы данных PostgreSQL, которое превращает её в полноценное векторное хранилище.	• векторы хранятся в тех же таблицах, что и обычные данные • возможность использовать SQL функциональности • надежность	• индексы IVFFlat/HNSW могут работать медленее, чем в специализированных векторных базах данных
Qdrant	Современная векторная база данных, написанная на rust.	• высокая скорость поиска при умеренном потреблении RAM • продвинутая фильтрация • можно запустить как в Docker, так и в облаке	• документации и готовых решений меньше, так как библиотека новая
Milvus	Векторная база данных с открытым исходным кодом, спроектированная для хранения, индексации и поиска миллиардов векторов эмбеддингов.	• масштабируемость, база данных способная включать в себя десятки серверов • поддержка GPU • разнообразие индексов	• требует много ресурсов • сложность установки, настройки и управления

Weaviate	Векторная база данных с открытым исходным кодом, упрощающая разработку приложений с искуственным интеллектом.	• встроенные модули для классификации данных и RAG • гибридный поиск	• потребление памяти, требователен к RAM при росте данных • требует строгого описания классов данных