

Дополнительные вопросы и ответы

Дополнительные вопросы

1. Как Qdrant реализует фильтрацию по метаданным при поиске ближайших соседей с помощью HNSW, и почему pre-filtering эффективнее post-filtering? Объясните роль инвертированного индекса в этом процессе.
1. В чем основные различия между скалярным и продуктовым квантованием в Qdrant? Как эти методы влияют на память, скорость и точность поиска?
1. Как устроена иерархия слоев в HNSW-графе, и почему это позволяет достигать логарифмической сложности поиска? Какие ограничения классического HNSW решает Qdrant?

Вопросы и ответы

1. Q&A

1. Объясните, каким образом Qdrant реализует эффективную фильтрацию по метаданным (payload) в процессе ANN-поиска с использованием HNSW-графа. В чем заключается отличие механизма pre-filtering в Qdrant от наивного post-filtering, и почему это критически важно для производительности и полноты результатов? Приведите формальное описание работы инвертированного индекса и его интеграции с обходом графа HNSW.

На основании предоставленного учебного материала, реализация эффективной фильтрации в Qdrant, ее отличия от наивных подходов и интеграция с HNSW-графом объясняются следующим образом.

1. Реализация эффективной фильтрации в Qdrant: Механизм Pre-filtering

Способность Qdrant эффективно сочетать ANN-поиск с фильтрацией по метаданным (payload) является одним из его ключевых преимуществ, решающим задачу **фильтрованного ANN-поиска**. Вместо наивных подходов Qdrant реализует механизм, известный как **"pre-filtering"**.

Суть этого механизма заключается в том, что условие фильтрации применяется *до и во время* обхода графа HNSW, а не после завершения поиска. Этот подход обеспечивается глубокой интеграцией двух ключевых структур данных: инвертированного индекса для метаданных и графа HNSW для векторов.

Процесс выглядит следующим образом:

1. Сначала, с использованием инвертированного индекса, формируется множество идентификаторов (ID) векторов, S_{filter} , которые полностью удовлетворяют заданному условию фильтрации.
2. Затем, ANN-поиск по графу HNSW выполняется таким образом, что в качестве кандидатов на ближайших соседей рассматриваются только те узлы графа, чьи ID принадлежат множеству S_{filter} .

Ключевая инновация реализации Qdrant состоит в том, как именно происходит обход графа. Если в процессе поиска алгоритм достигает узла, который не принадлежит множеству S_{filter} , этот узел игнорируется как потенциальный результат. Однако, в отличие от примитивных реализаций, его соседи по графу все равно могут быть рассмотрены для дальнейшего обхода. Это позволяет поддерживать связность поиска и продолжать навигацию по графу, даже если он "разрезан" условиями фильтра. Такой подход позволяет избежать катастрофического падения производительности, характерного для многих других реализаций HNSW при фильтрации.

2. Отличие Pre-filtering от Post-filtering и его критическая важность

Различие между механизмами pre-filtering в Qdrant и наивным post-filtering является фундаментальным и критически важным для производительности и полноты (recall) результатов поиска.

- **Наивный Post-filtering:** Этот подход разделяет поиск и фильтрацию на два независимых этапа.

1. Сначала выполняется стандартный ANN-поиск для нахождения k ближайших соседей к вектору запроса,

игнорируя любые метаданные.

2. Затем из полученных k результатов отбрасываются те, которые не удовлетворяют условиям фильтра.

Недостатки Post-filtering:

- **Низкая полнота результатов (Recall):** Этот подход неэффективен и, как указано в материале, "часто возвращает пустой результат". Это происходит, когда ни один из k ближайших векторов не соответствует фильтру, даже если в базе данных существуют другие, чуть более удаленные векторы, которые ему соответствуют. В этом случае пользователь получает неполный или пустой ответ, хотя релевантные данные существуют.
 - **Низкая производительность:** Выполняется полноценный ANN-поиск, результаты которого могут быть полностью отброшены на втором этапе. Это является нерациональным использованием вычислительных ресурсов.
- **Qdrant Pre-filtering:** Этот подход интегрирует фильтрацию непосредственно в процесс поиска.
 1. Условие фильтра применяется на лету во время обхода графа HNSW.
 2. Поиск целенаправленно ведется только среди подмножества векторов, удовлетворяющих фильтру.

Преимущества Pre-filtering:

- **Высокая полнота результатов:** Поиск изначально ограничен только релевантным подмножеством данных. Алгоритм найдет k ближайших соседей внутри этого отфильтрованного подмножества, гарантируя, что все возвращенные результаты соответствуют фильтру и являются наиболее близкими из возможных.
- **Высокая производительность:** Алгоритм не тратит время на вычисление расстояний до векторов, которые заведомо не пройдут фильтрацию. Интеграция с HNSW позволяет эффективно перемещаться по графу даже в условиях строгой фильтрации, избегая деградации производительности.

Таким образом, pre-filtering является критически важным, поскольку он гарантирует как корректность и полноту результатов, так и высокую производительность фильтрованного поиска, что является основной причиной для выбора Qdrant для production-сценариев со сложными запросами.

3. Формальное описание инвертированного индекса и его интеграция с HNSW

Определение 5: Инвертированный индекс (Inverted Index)

Для обеспечения быстрой фильтрации Qdrant строит инвертированный индекс по полям метаданных (payload). Формально, этот индекс представляет собой структуру данных, которая отображает значения полей в список идентификаторов (ID) векторов, обладающих этими значениями.

Например, для поля `color` со значением `blue` инвертированный индекс будет содержать запись: `{"color": "blue"} → [id_1, id_5, id_42, ...]`

Как указано в материале, Qdrant поддерживает сложные фильтры, для которых строятся соответствующие индексы, включая вложенные JSON-структуры, числовые диапазоны, геолокацию и сложные логические комбинации (AND/OR).

Интеграция инвертированного индекса с обходом графа HNSW

Интеграция этих двух механизмов является основой для эффективного pre-filtering и происходит в два этапа:

1. **Формирование множества допустимых ID:** Перед началом ANN-поиска система обращается к инвертированному индексу с условиями фильтра из запроса. В результате выполнения логических операций над списками ID, соответствующих каждому условию, формируется итоговое множество S_{filter} , содержащее ID всех векторов, которые удовлетворяют полному фильтру.
2. **Модифицированный обход графа HNSW:** Алгоритм поиска HNSW, начиная с точки входа на верхнем слое, выполняет итеративный спуск и поиск. На каждом шаге обхода, когда рассматривается узел-кандидат, его ID проверяется на принадлежность множеству S_{filter} .
 - Если ID узла **принадлежит** S_{filter} , он рассматривается как валидный кандидат в ближайшие соседи, и расстояние до него вычисляется.
 - Если ID узла **не принадлежит** S_{filter} , он немедленно отбрасывается как кандидат. Однако, что является ключевым моментом, его соседи по графу не отбрасываются, а добавляются в очередь для дальнейшего рассмотрения. Это позволяет алгоритму "перепрыгивать" через неподходящие узлы, не теряя связности графа и продолжая эффективный поиск в других его частях.

Этот синтез инвертированного индекса и модифицированного алгоритма обхода HNSW позволяет Qdrant выполнять фильтрованный поиск с высокой производительностью и точностью, что является его отличительной архитектурной особенностью.

2. Q&A

2. Сравните математические основы и практические последствия применения скалярного квантования (Scalar Quantization, SQ) и продуктового квантования (Product Quantization, PQ) в Qdrant. Как эти методы влияют на точность поиска, использование памяти и вычислительную эффективность? Опишите процесс квантования и деквантования для SQ, а также схему кодирования и асимметричного вычисления расстояний для PQ.

На основе предоставленного учебного материала, сравнение скалярного (Scalar Quantization, SQ) и продуктового (Product Quantization, PQ) квантования в Qdrant, их математических основ и практических последствий выглядит следующим образом.

Сравнение Скалярного и Продуктового Квантования в Qdrant

Квантование в Qdrant — это метод, направленный на сокращение точности представления векторов с целью уменьшения занимаемой памяти и ускорения вычислений.

1. Скалярное квантование (Scalar Quantization, SQ)

Математические основы и процесс квантования/де-квантования

Скалярное квантование преобразует 32-битные числа с плавающей запятой (float32) в 8-битные целые числа (int8). Процесс основан на линейном отображении диапазона значений.

- **Математическая деривация:** Пусть дан вектор $\vec{v} = (v_1, \dots, v_d) \in \mathbb{R}^d$.
 1. **Определение диапазона:** Для обеспечения устойчивости к выбросам в наборе данных определяются значения, задающие рабочий диапазон. В Qdrant для этого используются 2-й и 98-й перцентили, обозначаемые как p_2 и p_{98} соответственно.
 2. **Вычисление шага квантования:** Для b -битного квантования (в Qdrant обычно $b = 8$, что соответствует $2^8 = 256$ уровням) вычисляется шаг квантования Δ .

$$\Delta = \frac{p_{98} - p_2}{2^b - 1}$$

- **Пошаговая деривация:**
 - $p_{98} - p_2$: Вычисляется размах значений между 98-м и 2-м перцентилями, что отсекает экстремальные выбросы.
 - $2^b - 1$: Определяется количество интервалов, на которые будет разделен диапазон. Для $b = 8$ это 255 интервалов.
 - Деление первого на второе дает размер одного шага (интервала) квантования.
- 3. **Процесс квантования:** Каждая компонента v_i исходного вектора преобразуется в целочисленное значение v'_i .

$$v'_i = \text{round} \left(\frac{v_i - p_2}{\Delta} \right)$$

- **Пошаговая деривация:**
 - $v_i - p_2$: Значение компоненты сдвигается так, чтобы нижняя граница диапазона (p_2) соответствовала нулю.
 - $\frac{\cdot}{\Delta}$: Полученное значение делится на шаг квантования, чтобы определить, в какой из $2^b - 1$ интервалов оно попадает.
 - $\text{round}(\cdot)$: Результат округляется до ближайшего целого числа, которое и является квантованным

представлением компоненты.

4. **Процесс де-квантования:** Для восстановления приближенного исходного значения \hat{v}_i из квантованного v'_i выполняется обратное преобразование.

$$\hat{v}_i = v'_i \cdot \Delta + p_2$$

■ **Пошаговая деривация:**

- $v'_i \cdot \Delta$: Целочисленный индекс умножается на шаг квантования для восстановления его приблизительного положения в исходном диапазоне.
- $\dots + p_2$: К результату прибавляется значение нижней границы диапазона (p_2) для возврата к исходной шкале значений.

Влияние на производительность и ресурсы

- **Использование памяти:** SQ снижает потребление RAM в 4 раза, так как каждый float32 (4 байта) заменяется на int8 (1 байт).
- **Вычислительная эффективность:** Qdrant может вычислять метрики расстояния непосредственно на квантованных векторах, используя быстрые целочисленные SIMD-инструкции (например, AVX). Это значительно ускоряет вычисления, что является критически важным для производительности.
- **Точность поиска:** Вносимая ошибка квантования считается приемлемой для большинства практических задач, включая RAG (Retrieval-Augmented Generation).

2. Продуктовое квантование (Product Quantization, PQ)

Математические основы и схема кодирования

Продуктовое квантование обеспечивает еще более сильное сжатие данных, что особенно полезно для хранения векторов на диске.

• **Математическая деривация и схема кодирования:**

1. **Разбиение вектора:** Исходный вектор $\vec{v} \in \mathbb{R}^d$ разбивается на m непересекающихся суб-векторов $\vec{v}_1, \dots, \vec{v}_m$, каждый из которых имеет размерность d/m .
2. **Создание кодовых книг:** Для каждого из m подпространств независимо создается своя кодовая книга (codebook) C_j . Каждая книга C_j состоит из k векторов-центроид (обычно $k = 256$), которые находятся с помощью алгоритма кластеризации k-means на наборе суб-векторов из соответствующего подпространства.
3. **Кодирование:** Каждый суб-вектор \vec{v}_j исходного вектора заменяется индексом i_j ближайшего к нему центроида из соответствующей кодовой книги C_j . Таким образом, весь вектор \vec{v} представляется в виде набора из m целочисленных индексов (i_1, \dots, i_m) .

Асимметричное вычисление расстояний (Asymmetric Distance Computation)

Для вычисления расстояния между вектором запроса q (который не сжат) и сжатым вектором v используется высокоэффективный асимметричный метод.

• **Схема вычисления:**

1. Вектор запроса q также разбивается на m суб-векторов (q_1, \dots, q_m) , аналогично векторам в базе.
2. Для каждого подпространства j предварительно вычисляется таблица расстояний между суб-вектором запроса q_j и всеми k центроидами из кодовой книги C_j .
3. Итоговое расстояние между q и v аппроксимируется как сумма предвычисленных значений из этих таблиц, соответствующих индексам (i_1, \dots, i_m) , которыми закодирован вектор v .

Этот метод описан в материале как "чрезвычайно быстрый".

Влияние на производительность и ресурсы

- **Использование памяти:** PQ обеспечивает "еще более сильное сжатие" по сравнению с SQ. Qdrant использует этот метод для хранения векторов на диске, что позволяет работать с наборами данных, значительно превышающими объем доступной RAM.
- **Вычислительная эффективность:** Асимметричное вычисление расстояний является чрезвычайно быстрым, так как основная часть вычислений сводится к извлечению предвычисленных значений из таблиц и их суммированию.

- **Точность поиска:** Расстояние является аппроксимацией, что подразумевает более высокую потерю точности по сравнению со скалярным квантованием.

Сравнительный итог

| Характеристика | Скалярное квантование (SQ) | Продуктовое квантование (PQ) | | :--- | :--- | :--- | | **Математическая основа** | Линейное преобразование каждой компоненты вектора в целое число (int8). | Разбиение вектора на суб-векторы и кодирование каждого суб-вектора индексом ближайшего центроида из кодовой книги. | | **Использование памяти** | Снижение в 4 раза (float32 -> int8). | Значительно более сильное сжатие, позволяющее хранить данные на диске и работать с наборами, превышающими RAM. | | **Вычислительная эффективность** | Ускорение вычислений расстояний за счет использования быстрых целочисленных SIMD-инструкций. | Чрезвычайно быстрое асимметричное вычисление расстояний с использованием предвычисленных таблиц. | | **Точность поиска** | Вносимая ошибка приемлема для большинства задач (например, RAG). | Аппроксимация расстояния, что подразумевает потенциально большую потерю точности. | | **Основной сценарий в Qdrant** | Ускорение вычислений и экономия RAM для векторов, находящихся в оперативной памяти. | Сжатие векторов для хранения на диске, работа с очень большими наборами данных. |

3. Q&A

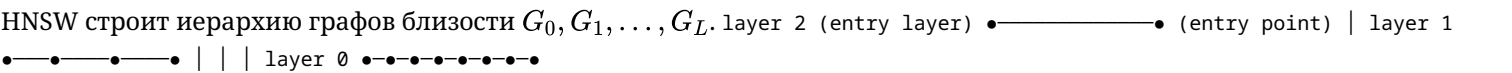
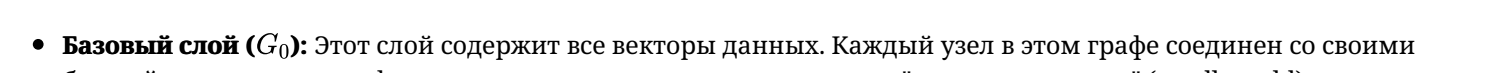
3. Проанализируйте архитектуру HNSW, используемую в Qdrant, с точки зрения сложности поиска и вставки элементов. Как иерархическая структура графа и вероятностное распределение узлов по слоям обеспечивают логарифмическую сложность поиска? Какие ограничения классического HNSW существуют в отношении фильтрации и масштабирования, и каким образом Qdrant преодолевает эти ограничения на уровне системы?

На основе предоставленного учебного материала, анализ архитектуры HNSW в Qdrant, ее ограничений и системных решений Qdrant выглядит следующим образом.

1. Архитектура HNSW и сложность поиска и вставки

Алгоритм Hierarchical Navigable Small World (HNSW) является основной структурой индексации в Qdrant для решения задачи приблизительного поиска ближайшего соседа (Approximate Nearest Neighbor, ANN).

1.1. Структура HNSW

HNSW строит иерархию графов близости G_0, G_1, \dots, G_L . layer 2 (entry layer)  (entry point) | layer 1 

- **Базовый слой (G_0):** Этот слой содержит все векторы данных. Каждый узел в этом графе соединен со своими ближайшими соседями, формируя структуру с характеристиками "маленького мира" (small world).
- **Верхние слои ($G_l, l > 0$):** Каждый последующий слой G_l представляет собой разреженный подграф нижележащего слоя G_{l-1} . Эти слои содержат "длинные" ребра, которые позволяют быстро перемещаться по всему векторному пространству.

1.2. Алгоритм вставки элемента

Процесс добавления нового элемента q в индекс HNSW состоит из следующих шагов:

1. **Выбор максимального слоя:** Для нового элемента q случайным образом, с экспоненциально убывающей вероятностью, выбирается максимальный слой l_{max} , на котором он будет представлен.
2. **Поиск точки входа:** Поиск начинается с единственной точки входа на самом верхнем слое иерархии, L .
3. **Итеративный спуск:** На каждом слое l от L до $l_{max} + 1$ выполняется жадный поиск для нахождения элемента, ближайшего к q . Найденный узел на слое l используется как точка входа для поиска на слое $l - 1$.
4. **Вставка и соединение:** Начиная со слоя $\min(L, l_{max})$ и до базового слоя (0), для элемента q выполняется поиск M ближайших соседей. Между q и найденными соседями устанавливаются ребра. Для контроля степени связности узлов применяются специальные эвристики.

1.3. Алгоритм поиска и его сложность

Поиск k ближайших соседей к запросу q выполняется следующим образом:

1. **Поиск точки входа:** Аналогично вставке, поиск начинается с верхнего слоя L .
2. **Жадный спуск:** На слоях от L до 1 выполняется жадный поиск, чтобы найти узел, наиболее близкий к q . Этот узел становится точкой входа для поиска на следующем, более плотном слое.
3. **Поиск на базовом слое:** На слое G_0 выполняется более тщательный поиск (например, beam search) для нахождения итогового набора из k ближайших соседей.

Согласно учебному материалу, **средняя сложность поиска в HNSW составляет $O(\log N)$** , где N — общее число векторов в базе данных.

2. Обеспечение логарифмической сложности поиска

Логарифмическая сложность поиска в HNSW достигается за счет синергии двух ключевых аспектов его архитектуры: иерархической структуры и вероятностного распределения узлов.

- **Иерархическая структура графа:** Иерархия слоев позволяет реализовать стратегию поиска от грубого к точному. **Верхние слои содержат "длинные" ребра**, которые соединяют удаленные друг от друга части пространства, обеспечивая быстрое перемещение на большие расстояния на начальных этапах поиска. **Нижние слои, в свою очередь, содержат "короткие" ребра**, обеспечивая высокую точность локального поиска на финальных этапах. Этот многоуровневый подход позволяет избежать полного перебора всех узлов.
- **Вероятностное распределение узлов:** Узлы для верхних слоев выбираются вероятностно. Как указано в материале, это **"обеспечивает логарифмическое уменьшение числа узлов с ростом номера слоя"**. Поскольку количество узлов на каждом последующем уровне уменьшается экспоненциально, общее количество слоев в иерархии пропорционально $\log N$. Так как алгоритм поиска на каждом слое посещает ограниченное количество узлов, общая сложность оказывается логарифмической.

3. Ограничения классического HNSW и решения Qdrant

Учебный материал выделяет следующие ограничения классического алгоритма HNSW и описывает, как Qdrant их преодолевает на системном уровне.

3.1. Ограничения в отношении фильтрации

- **Ограничение:** Классический алгоритм HNSW в его базовой форме **не поддерживает эффективную префильтрацию (pre-filtering)**. Попытка выполнить поиск по заранее отфильтрованному подмножеству данных часто приводит к значительной деградации производительности, поскольку структура графа может оказаться несвязной для отфильтрованного набора.
- **Решение Qdrant:** Qdrant решает эту проблему, реализуя собственный механизм фильтрации на системном уровне.
 1. **Инвертированный индекс (Inverted Index):** Qdrant строит инвертированные индексы по полям метаданных (payload), которые отображают значения полей в списки ID векторов. Это позволяет быстро формировать множество ID векторов S_{filter} , удовлетворяющих заданному условию фильтра.
 2. **Интегрированный Pre-filtering:** ANN-поиск по графу HNSW выполняется таким образом, что **рассматриваются только узлы, принадлежащие множеству S_{filter}** . Ключевая инновация Qdrant заключается в том, что его реализация HNSW эффективно работает с такими разреженными наборами. Во время обхода графа, если узел не принадлежит S_{filter} , он игнорируется, но **его соседи все равно могут быть рассмотрены**. Это позволяет поддерживать связность поиска и избегать катастрофического падения производительности, характерного для других реализаций.

3.2. Ограничения в отношении масштабирования

- **Ограничение:** HNSW как алгоритм сам по себе **не включает в себя механизмы продуктового квантования (Product Quantization) или шардинга**. Эти механизмы критически важны для масштабирования на большие объемы данных и управления потреблением ресурсов.
- **Решение Qdrant:** Qdrant решает эти ограничения, **"реализуя данные механизмы на уровне всей системы, поверх своей кастомизированной реализации HNSW"**.
 - **Квантование:** Qdrant внедряет скалярное и продуктивное квантование для сжатия векторов. Это позволяет снизить потребление RAM (в 4 раза для скалярного квантования) и ускорить вычисления расстояний, что является формой вертикального масштабирования и оптимизации производительности.
 - **Шардинг:** Хотя детали реализации шардинга не раскрываются в предоставленном тексте, в нем указывается,

что Qdrant спроектирован для горизонтального масштабирования и масштабирования коллекций, что подразумевает наличие механизма шардинга на системном уровне для распределения данных по нескольким узлам.

Таким образом, Qdrant расширяет возможности классического HNSW, интегрируя его с дополнительными системными компонентами, такими как инвертированные индексы, механизмы квантования и шардинга, что делает его мощным и масштабируемым решением для production-сценариев.
