Технический отчёт (STM)

1. Описание задачи.

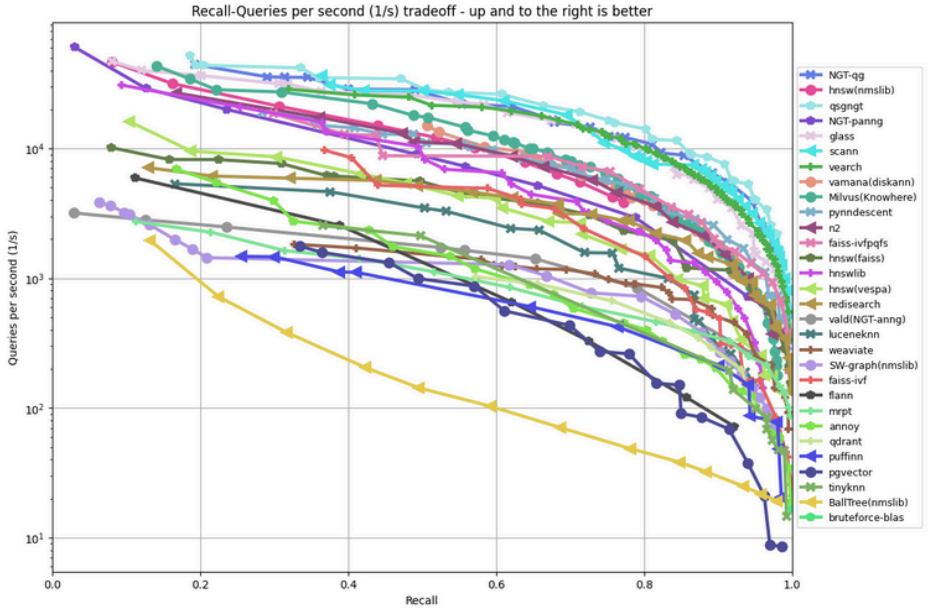
Задача: нахождение или реализация алгоритма приближенного поиска ближайших соседей (ANN - Approximate Nearest Neighbours).
Бенчмарки (датасеты): SIFT, GIST.
Расстояние: L2 Euclid.
Метрика: recall@10.
Ограничение по времени: 1800 секунд на построение индекса, 100 секунд на запросы.
Ограничение по памяти: 1 ГБ оперативной памяти. 10 ГБ постоянной.

2. Выбор алгоритма.

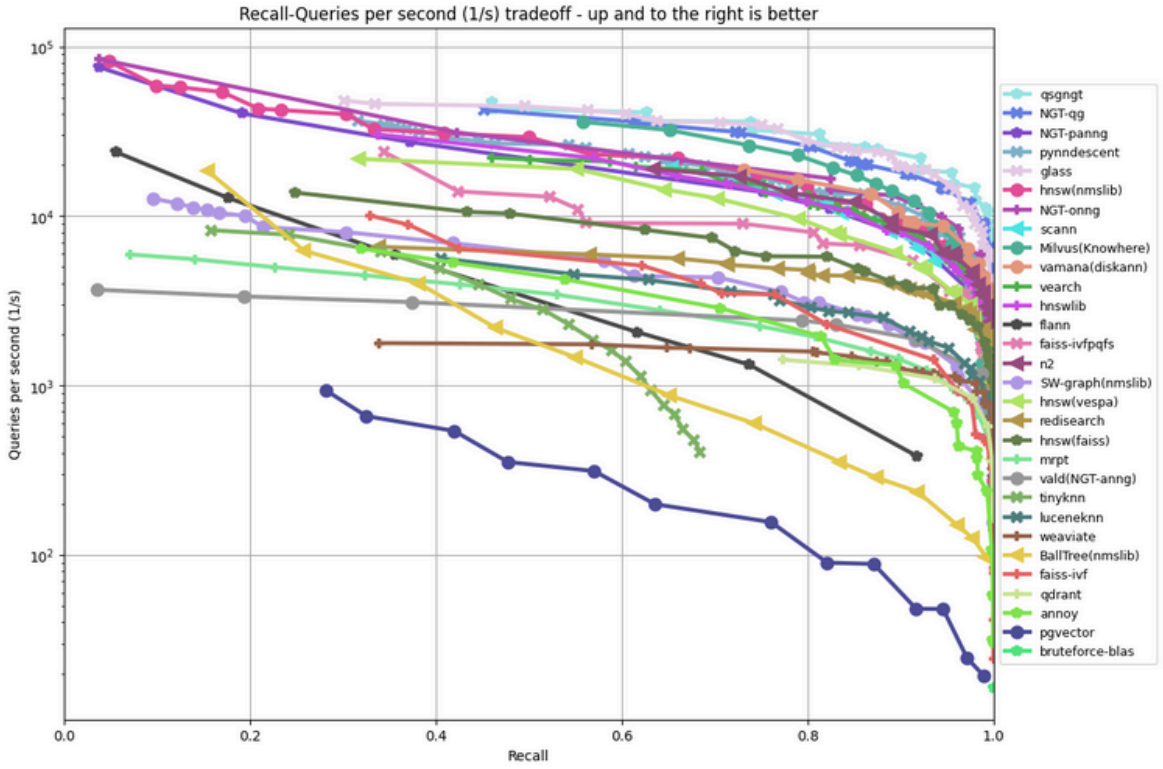
Для начала мы провели анализ существующих решений и отобрали несколько претендентов. Для изучения решений мы обратились к общедоступным сайтам с бенчмарками на различных датасетах (https://ann-benchmarks.com/). Каждый отдельный алгоритм был рассмотрен в соотношениях Recall@k/QueryPerSecond.

Рассмотрим для примера несколько бенчмарков с сайта.

Glove:


Recall-Queries per second (1/s) tradeoff - up and to the right is better
Queries per second (1/s)
Recall
NGT-qg
hnsw(nmslib)
qsgngt
NGT-panng
glass
scann
vearch
vamana(diskann)
Milvus(Knowhere)
pynndescent
n2
faiss-ivfpqfs
hnsw(faiss)
hnswlib
hnsw(vespa)
redisearch
vald(NGT-anng)
luceneknn
weaviate
SW-graph(nmslib)
faiss-ivf
flann
mrpt
annoy
qdrant
puffinn
pgvector
tinyknn
BallTree(nmslib)
bruteforce-blas

Sift-128:


Recall-Queries per second (1/s) tradeoff - up and to the right is better
Queries per second (1/s)
Recall
qsgngt
NGT-qg
NGT-panng
pynndescent
glass
hnsw(nmslib)
NGT-onng
scann
Milvus(Knowhere)
vamana(diskann)
vearch
hnswlib
flann
faiss-ivfpqfs
n2
SW-graph(nmslib)
hnsw(vespa)
redisearch
hnsw(faiss)
mrpt
vald(NGT-anng)
tinyknn
luceneknn
weaviate
BallTree(nmslib)
faiss-ivf
qdrant
annoy
pgvector
bruteforce-blas

Как видим, в лидерах стабильно оказываются алгоритмы **NGT-qg**, **HNSW**, **qsgngt** (Huawei) и **glass** (Zilliz). Сравним их между собой и выберем оптимальный вариант для каждой нашей задачи. Есть два принципиально разных подхода: NGT и HNSW.

- **NGT-qg** (Neighborhood Graph and Tree - Quantized Graph)
  Алгоритм, основывающийся на построении графов. Хорошо подходит для высокоразмерых датасетов благодаря квантизации в методе qg, имеет высокую точность и хорошие параметры по скорости. Однако его недостатком является очень низкая скорость построения индекса, что не уложится в 1800 секунд для GIST. Также он занимает приблизительно в 2 раза больше памяти в сравнении с другими и не приспособлен к распараллеливанию.

- **qsgngt** (HWTL_SDU-ANNS).
  Алгоритм, продолжающий идею построения графов из библиотеки NGT. Обладает примерно теми же преимуществами и недостатками, по скорости местами чуть быстрее, местами чуть медленнее.

- **HNSW (nmslib)** (Hierarchical navigable small world)
  Алгоритм построения графов по сгруппированным уровням (“малые миры”). Сама идея - SOTA по скорости и потреблению памяти во многих задачах, требует минимум памяти на хранение графов и быстро строит индексы. Доступно распараллеливание вычислений. Алгоритм представлен в нескольких библиотеках.

- **Glass** (Zilliz HNSW-ANN lib).
  Построенная на HNSW алгоритме библиотека. Дорабатывает классический HNSW и почти везде показывает себя лучше, особенно по скорости.

Исходя из наших потребностей в быстром построении индекса и ограничении по памяти, лучше всего взять HNSW, в особенности **Glass.** Мы также учитывали другие интересные алгоритмы, такие как разделяющий на области **ANNOY** и основанный на сжатии пространства **FAISS** (Facebook search), однако они заметно хуже в плане скорости построения индекса и использования памяти (все замеры есть на сайте ANN-bench).

3. Выбор языка и библиотеки.

Для реализации индекса и поиска был выбран язык **python3** и разные реализации HNSW: собственная, от hnswlib (header-only версия из nmslib, C++ и Python версии) и от Zilliz (**glassppy**). Мы решили протестировать эти четыре вариации и посмотреть, какая из них подойдёт нам лучше всего. Для оптимизации подсчётов в рамках теста был добавлен алгоритм PCA - Метод Главных Компонент, позволяющий сжать n-мерный вектор в k-мерный с помощью разложения сингулярной матрицы или построения ковариационной матрицы для получения собственных векторов (eigenvalues).

По результатам тестов получилось следующее:

1. Собственная реализация работает только с помощью PCA и выдаёт невысокий recall@10 (Меньше 0.1), не всегда укладываясь по времени сборки или подсчёта query (Связано это с реализацией на чистом python + numpy, что не позволило использовать ни параллельные вычисления, ни скорость компилируемых языков, поэтому данный вариант нам не подошёл).
2. Реализации в hnswlib были протестированы на sift. Первый результат проверки python версии на sift при параметрах ef_construction=150, M=16, num_workers=4:

```
SIFT build (seconds): 179.0
SIFT Seq Recall: 0.694
SIFT Seq QPS: 0.5
SIFT 2 Threads Recall: 0.684091
SIFT 2 Threads QPS: 0.86
SIFT 4 Threads Recall: 0.684375
SIFT 2 Threads QPS: 0.92
```

   Версия на C++ (те же параметры).

```
SIFT build (seconds): 373.0
SIFT Seq Recall: 0.694545
SIFT Seq QPS: 0.55
SIFT 2 Threads Recall: 0.698058
SIFT 2 Threads QPS: 1.01
SIFT 4 Threads Recall: 0.691964
SIFT 2 Threads QPS: 1.08
```

   Как можем заметить, ценой большего времени имплементации и и меньшего потребления памяти получаются приблизительно равные

результаты. Но из-за возможности на python легко параллелить решение мы предпочтем его.

3. Pyglass (Zilliz). В результате тестирования было выявлено, что алгоритм очень плохо работает с очень большими датасетами и ему невозможно выдавать их порционно (как в случае в hnswlib), плюс он требует дополнительного сохранения датасета повторно для запуска Searcher метода, что исключает его из списка претендентов на лучший алгоритм.

По итогам тестирования результаты показал только hnswlib. Его мы и решаем доработать в нашем нынешнем решении. Дальше мы будем показывать решение с учётом данного метода.

Hierarchical navigable small world

Как было сказано, алгоритм построен на графах, располагающихся в иерархической уровневой системе. Каждый уровень представляет из себя так называемый малый мир, в котором расстояния между точками незначительны. Подробнее ознакомиться с его структурой можно в данной работе, написанной авторами библиотеки hnswlib: https://arxiv.org/pdf/1603.09320.

Что выделяет его на фоне других:

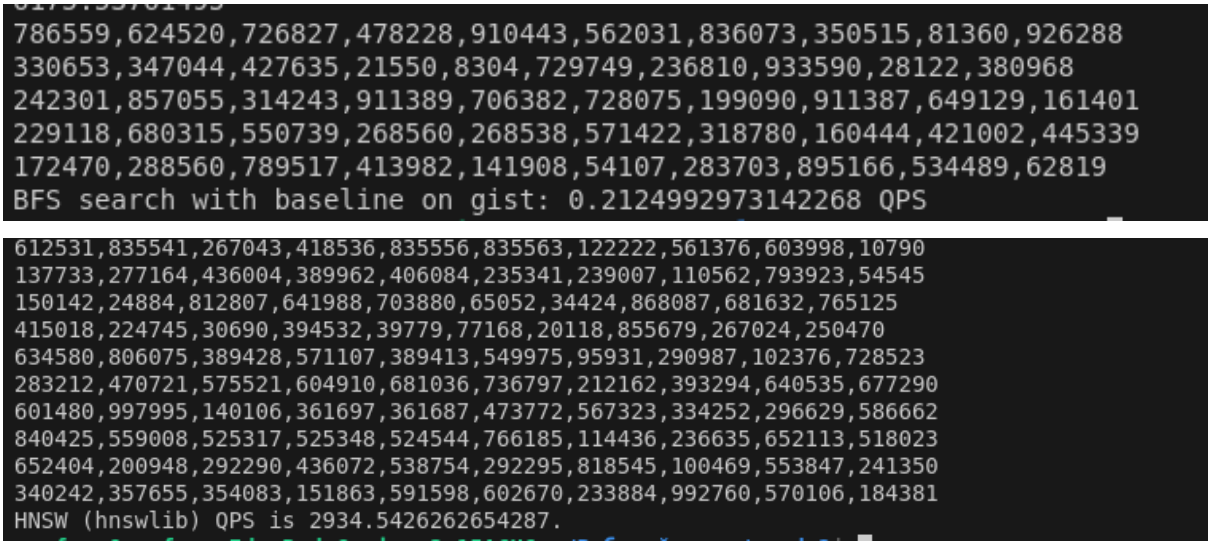
+ Минимальные затраты по памяти относительно других алгоритмов, таких как NGT, FAISS, ANNOY при сохранении той же или большей скорости поиска и создания индекса.
+ Возможность распараллелить вычисления на CPU.
+ Легко имплементировать.
+ Удобная настройка гиперпараметров.

Однако есть у него и недостатки:

- Медленнее реализаций на GPU (FAISS).
- Не самая высокая предельная скорость среди всех алгоритмов.

Важное уточнение: в связи с тем, что хорошо подобранный hnsw показывал себя по скорости как bruteforce, мы провели собственные замеры, сравнивающие выполнение всех query для датасета GIST. По результатам замеров мы обнаружили очень сильное расхождение с показателями на тестовой системе и у себя. Связано это, скорее всего, с тем, что вместо выполнения всех k query в одной программе подсчет разделяется на запуск k программ search, в результате

чего получается не сильно отличный от baseline результат. Ниже приведено сравнение BFS и HNSW на всех query для GIST, где хорошо видна разница между этими алгоритмами. Формула QPS = len(queries)/time_elapsed.



На скриншоте приведены сравнения скорости BFS (Baseline) и HNSW на GIST1M. Как видим, разница в скорости равна примерно 14 тысяч раз при recall(HNSW | GIST) = 0.8.

Гиперпараметры

В ходе исследования гиперпараметров мы пришли к следующим значениям:

M (bidirectional connections) = 42 для SIFT и 24 для GIST.
Ef_coeff = 350 SIFT, 300 GIST.
Ef_search = 32 SIFT, 24 GIST.
num_workers = 4 для index и 1 для search.

Идеи для улучшения

К сожалению, параллельные вычисления на GIST не проходят по памяти, судя по всему. Решением этой проблемы в условиях малой оперативной памяти (1 ГБ) может быть квантизация датасета (**Product Quantization, Binary Quantization** с переводом датасета в uint8) Подробнее об этом можно прочитать в этой статье:

https://weaviate.io/blog/ann-algorithms-hnsw-pq. Однако в условиях нашей задачи это было невыполнимо. Для того, чтобы выполнить квантизацию и достичь хорошей точности, требуется не менее 30 минут, что не укладывается в наш лимит. На графике ниже представлены отношения времени в минутах к итоговой метрике recall@k.

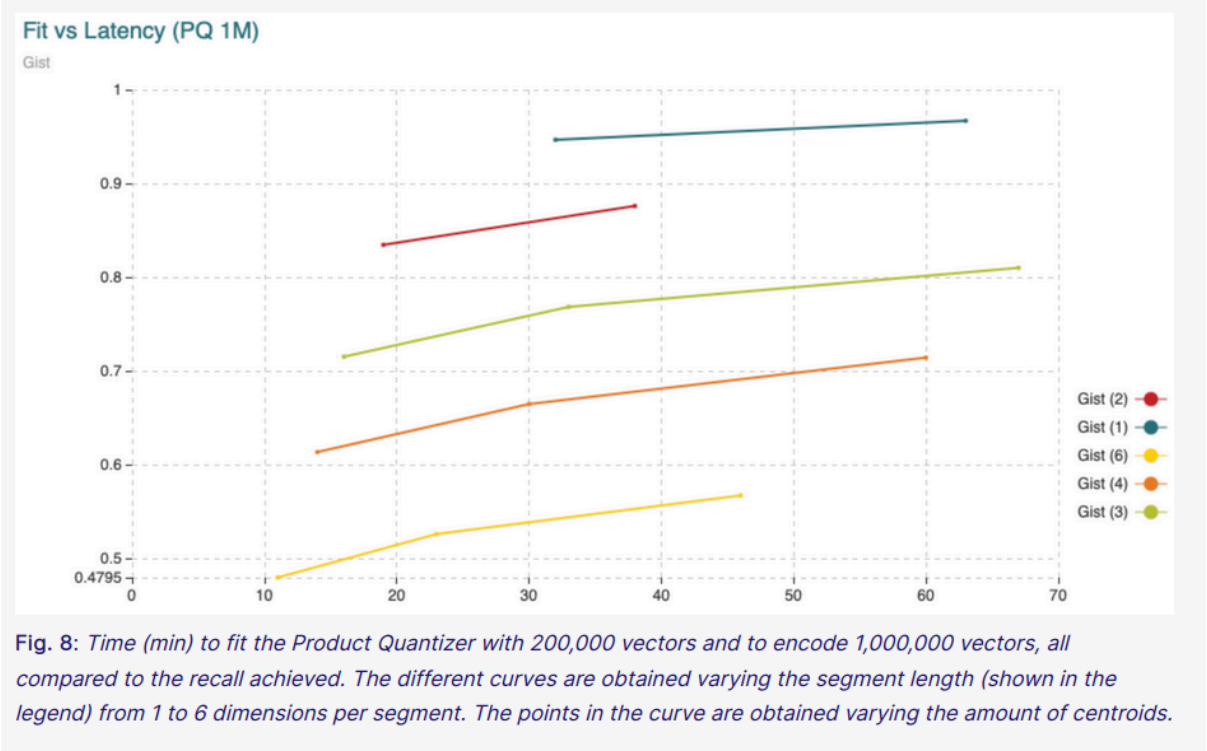


Fig. 8: *Time (min) to fit the Product Quantizer with 200,000 vectors and to encode 1,000,000 vectors, all compared to the recall achieved. The different curves are obtained varying the segment length (shown in the legend) from 1 to 6 dimensions per segment. The points in the curve are obtained varying the amount of centroids.*

Нами был опробован метод PCA, однако он требует много памяти для вычисления собственных чисел и векторов и на GIST он приводил к переполнению памяти, а на SIFT - к потере точности. Однако такие методы имеют место в случае меньших ограничений.