DiskANN

DiskANN: Fast accurate billion-point nearest neighbor search on a single node,
S. J. Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, Advances in Neural Information Processing Systems (NeurIPS), vol. 32, 2019.

AiSAQ: All-in-Storage ANNS with Product Quantization for DRAM-free Information Retrieval,
Kento Tatsuno, Daisuke Miyashita, Taiga Ikeda, Kiyoshi Ishiyama, Kazunari Sumiyoshi, Jun Deguchi, arXiv:2404.06004, 2024.

FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search,
Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, Harsha Vardhan Simhadri, arXiv:2105.09613, 2021.

CUBRID

목표

- DiskANN 를 분석해서 디스크 기반 k-ANN 인덱스에 대한 지식 축적
- CUBRID에 디스크 기반 k-ANN 인덱스 구현에 참고

발표 내용

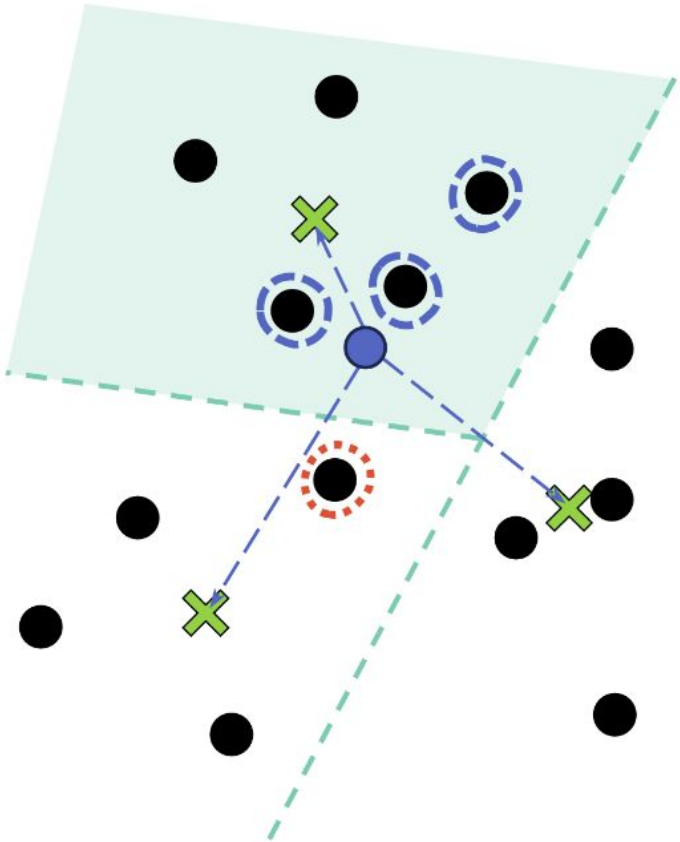
DiskANN: Fast accurate billion-point nearest neighbor search on a single node

AiSAQ: All-in-Storage ANNS with Product Quantization for DRAM-free Information Retrieval

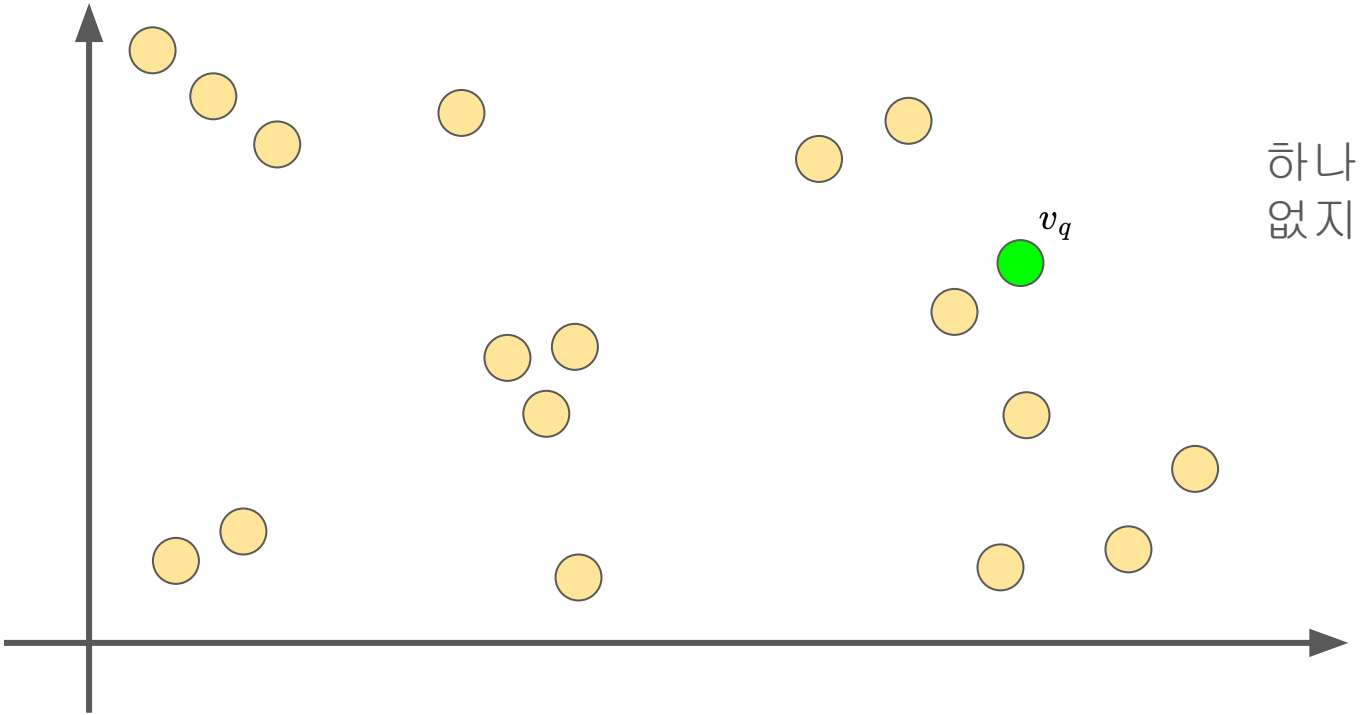
FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search

CUBRID

10억개의 고차원 벡터 데이터셋에 대해서
어떤 쿼리 벡터가 주어졌을 때,
k개의 가장 가까운 이웃을
어떻게 구할 수 있을까?



시작 예제 데이터셋 (상상의 나래)



하나하나 다 비교할 순 없지

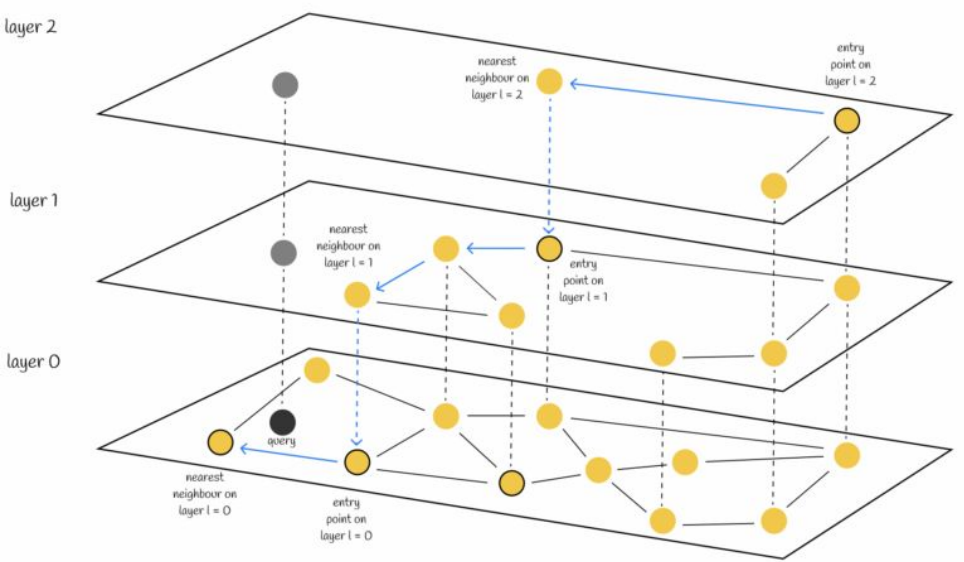
CUBRID

Background

- 수십억 개의 n차원 벡터 데이터를 어떻게 저장하고 검색할 것인가?
- 차원의 저주로 n이 커질수록 선형 검색은 저장 공간과 처리 시간이 불필요하게 증가
- 따라서 대부분의 연구는 현실적인 해결책으로 약간의 정확도를 포기하고 ANN 검색
- ANN 알고리즘에서 고려해야 할 성능
  - Recall
  - Latency
  - Throughput
- 현재 시점에서 검색 시간 대비 recall에서 가장 뛰어난 알고리즘은 주로 그래프 기반 방법
  - HNSW
  - NSG

CUBRID

Related Works



HNSW (Microsoft, 2019)

- 메모리에서 운영하는 것에 대한 자원 한계
- SSD에 저장할 수 있지만 검색 성능 한계

CUBRID

DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node

DiskANN?

- 64GB 램을 가진 노드에서, 수백 차원 이상의 10억 개의 벡터에 대해 **SSD에 저장**하며 검색 시 95% 이상의 1-recall@1, 5m latency 제공
- Vamana 알고리즘은 NSG, HNSW 보다 더 작은 diameter의 그래프 생성
  : sequential disk read를 최소화
- Vamana 그래프를 In-memory 에서도 활용 가능. 성능은 HNSW, NSG과 동등하거나 나음
- 대규모 데이터셋의 중첩된 파티션(overlapping partitions)을 위한 더 작은 Vamana 인덱스들은 손쉽게 병합.
  : 전체 데이터셋에 대해 단일 인덱스를 한 번에 구축했을 때와 거의 동일한 검색 성능
- Vamana가 범용 벡터 압축 기법 (Product Quantization)과 결합하여 DiskANN 시스템을 구축할 수 있음. Full-precision 벡터는 디스크에 저장하고, Compressed 벡터는 메모리에 캐시

CUBRID

Graph 관련 Notation

- 데이터셋을 P 집합으로 정의하고, $|P| = n$
- P 는 각 점에 해당하는 vertices와 vertex 사이의 edge를 가지는 directed graph
- 이러한 그래프를 $G = (P, E)$ 라 표기
- 어떤 directed graph 에서 $p \in P$ 일 때,
  $N_{out}(p)$ 는 p 에서 들의 집합
- x_p 는 p에 해당하는 벡터 데이터를 의미
- $d(p, q) = ||x_p - x_q||$ 는 두 점 p, q 사이의 거리 (metric distance)

연습장

The **Vamana** Graph Construction Algorithm

Background: Graph-based ANNs algorithm

- 인덱스 구축 시
  - 데이터셋 P 의 geometric property 특성을 고려해서 $G = (P, E)$ 를 생성 (*metric 공간*)
- 검색 시
  - 질의 벡터 x_q 에 대해서 $s \in P$ 에서 출발하여
  - 점차적으로 x_q 에 가까운 지점을 향해 G 그래프를 탐
    - 질의 벡터에 도달했을 때 (수렴),
      내가 접근했던 이웃 노드 중 가장 가까운 k개 반

Algorithm 1: GreedySearch(s, x_q, k, L)

Data: Graph G with start node s, query x_q, result size k, search list size $L \geq k$
Result: Result set $\mathcal{L}$ containing k-approx NNs, and a set $\mathcal{V}$ containing all the visited nodes

```
begin
    initialize sets L ← {s} and V ← ∅
    while L \ V ≠ ∅ do
        let p* ← arg min_{p∈L\V} ||x_p − x_q||
        update L ← L ∪ N_out(p*) and
          V ← V ∪ {p*}
        if |L| > L then
            update L to retain closest L
              points to x_q
    return [closest k points from L; V]
```

The **Vamana** Graph Construction Algorithm

Two Algorithms:

- GreedySearch
- RobustPrune

Algorithm 1: GreedySearch$(s, \mathtt{x}_q, k, L)$

Data: Graph G with start node s, query $\mathtt{x}_q$, result size k, search list size $L \geq k$
Result: Result set $\mathcal{L}$ containing k-approx NNs, and a set $\mathcal{V}$ containing all the visited nodes

begin
- initialize sets $\mathcal{L} \leftarrow \{s\}$ and $\mathcal{V} \leftarrow \emptyset$
- **while** $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ **do**
  - let $p* \leftarrow \arg\min_{p \in \mathcal{L} \setminus \mathcal{V}} ||\mathtt{x}_p - \mathtt{x}_q||$
  - update $\mathcal{L} \leftarrow \mathcal{L} \cup N_{\text{out}}(p^*)$ and $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
  - **if** $|\mathcal{L}| > L$ **then**
    - update $\mathcal{L}$ to retain closest L points to $\mathtt{x}_q$
- return [closest k points from $\mathcal{L}$; $\mathcal{V}$]

Algorithm 2: RobustPrune$(p, \mathcal{V}, \alpha, R)$

Data: Graph G, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, degree bound R
Result: G is modified by setting at most R new out-neighbors for p

begin
- $\mathcal{V} \leftarrow (\mathcal{V} \cup N_{\text{out}}(p)) \setminus \{p\}$
- $N_{\text{out}}(p) \leftarrow \emptyset$
- **while** $\mathcal{V} \neq \emptyset$ **do**
  - $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
  - $N_{\text{out}}(p) \leftarrow N_{\text{out}}(p) \cup \{p^*\}$
  - **if** $|N_{\text{out}}(p)| = R$ **then**
    - break
  - **for** $p' \in \mathcal{V}$ **do**
    - **if** $\alpha \cdot d(p^*, p') \leq d(p, p')$ **then**
      - remove p' from $\mathcal{V}$

The **Vamana** Graph Construction Algorithm

GreedySearch Algorithm

- 그래프 구축 시 고려해야할 것
  - $GreedySearch\ (s, x_q, k, L)$ 가
    - local minima에 쉽게 빠지지 않고
    - ANN에 빠르게 수렴할 수 있는 sparse graph를 어떻게 구성할 것인가?
  - 이를 보장하기 위한 충분조건: **희소 근접 그래프 (Sparse Neighborhood Graph, SNG)**
    - 각 노드에서, 질의점으로 가는 경로 상에 항상 더 가까운 이웃이 존재하는 그래프를 구축
    - 임의의 쿼리 q와 임의의 노드 p에 대해, p에서 출발했을 때 거리를 줄여주는 이웃을 따라가면 결국엔 q를 찾을 수 있다.
      - 연결성
      - 단조적 경로
    - 이상적인 그래프 구성이지만, 그래프 구축에 높은 비용이 발생 O(n^2)
    - 기존에 SNG 그래프의 특성을 근사하게 만드는 연구 (NSW 등), 그래프의 diameter를 조절할 수 있는 유연성이 제한적

CUBRID

Algorithm 1: GreedySearch(s, x_q, k, L)

Data: Graph G with start node s, query x_q, result size k, search list size $L \geq k$

Result: Result set $\mathcal{L}$ containing k-approx NNs, and a set $\mathcal{V}$ containing all the visited nodes

begin

- initialize sets $\mathcal{L} \leftarrow \{s\}$ and $\mathcal{V} \leftarrow \emptyset$
- **while** $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ **do**
  - let $p* \leftarrow \arg\min_{p \in \mathcal{L} \setminus \mathcal{V}} ||\mathrm{x}_p - \mathrm{x}_q||$
  - update $\mathcal{L} \leftarrow \mathcal{L} \cup N_{\mathrm{out}}(p^*)$ and $\mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
  - **if** $|\mathcal{L}| > L$ **then**
    - update $\mathcal{L}$ to retain closest L points to x_q
- return [closest k points from $\mathcal{L}$; $\mathcal{V}$]

L 집합 (우선순위큐) 중

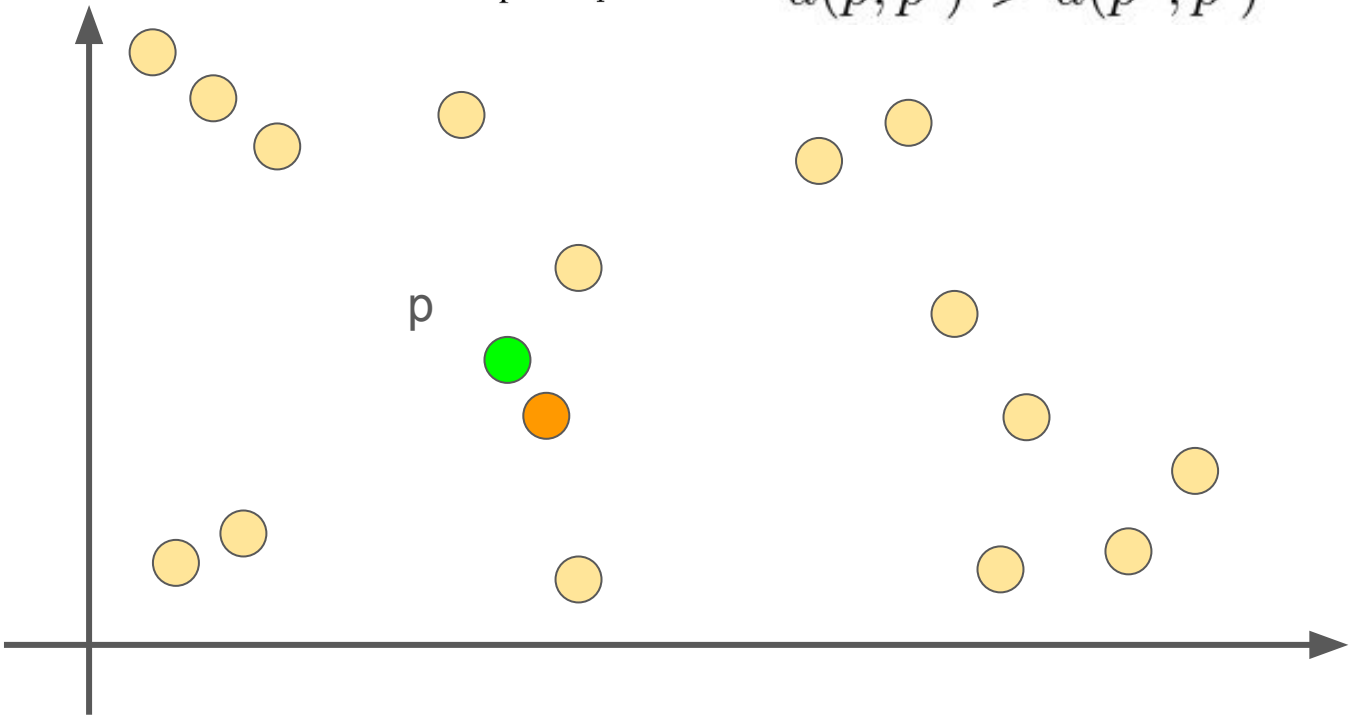
x_q 에 가장 가까운점

CUBRID

The **Vamana** Graph Construction Algorithm

SNG Graph

out-neighbors of each point p are determined as follows: initialize a set $S = P \setminus \{p\}$. As long as $S \neq \emptyset$, add a directed edge from p to p^*, where p^* is the closest point to p from S, and remove from S all points p' such that $d(p, p') > d(p^*, p')$



CUBRID

The **Vamana** Graph Construction Algorithm

The Robust Pruning Procedure

- SNG 속성을 만족하는 그래프의 경우, 그래프의 diameter가 클 수 있다.
  - 모든 vertex가 한 선 위에 늘어서는 line graph
  - 디스크에서 많은 순차적 읽기를 수행해야 하는 문제점
- 각 vertex에서 쿼리 vertex 까지의 거리가 $\alpha > 1$ 의 multiplicative factor 만큼 줄어들도록 설계
- 이 때 각 vertex에서 다른 모든 vertex를 보는 경우 $RobustPrune\ (p,\ P \setminus \{p\},\ \alpha,\ n-1)$
  - 인덱스 구축 비용이 너무 커짐
- Vamana는 을 $RobustPrune\ (p,\ V,\ \alpha,\ R)$ 사용
  - V는 n-1 vertices보다 더 적게 “신중히” 선택
  - R은 degree bound

Algorithm 2: RobustPrune$(p, \mathcal{V}, \alpha, R)$

Data: Graph G, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, degree bound R

Result: G is modified by setting at most R new out-neighbors for p

begin

$\mathcal{V} \leftarrow (\mathcal{V} \cup N_{\text{out}}(p)) \setminus \{p\}$

$N_{\text{out}}(p) \leftarrow \emptyset$

while $\mathcal{V} \neq \emptyset$ **do**

$p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$

$N_{\text{out}}(p) \leftarrow N_{\text{out}}(p) \cup \{p^*\}$

if $|N_{\text{out}}(p)| = R$ **then**

`break`

for $p' \in \mathcal{V}$ **do**

if $\alpha \cdot d(p^*, p') \leq d(p, p')$ **then**

`remove` p' `from` $\mathcal{V}$

- V와 p의 out-neighbor에 대한 재조정
  - 가장 가까운점 하나는 무조건 연결
  - 선택된 가까운 점에 대해서 shadow 되는 다른 점은 대상에서 제거
    - a 값에 따라 shadow 영역 조정
    - V가 greedy search라면 detour가 제거됨

CUBRID

Algorithm 2: RobustPrune$(p, \mathcal{V}, \alpha, R)$

Data: Graph G, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, degree bound R

Result: G is modified by setting at most R new out-neighbors for p

begin

$\mathcal{V} \leftarrow (\mathcal{V} \cup N_{\text{out}}(p)) \setminus \{p\}$

$N_{\text{out}}(p) \leftarrow \emptyset$

while $\mathcal{V} \neq \emptyset$ **do**

$p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$

$N_{\text{out}}(p) \leftarrow N_{\text{out}}(p) \cup \{p^*\}$

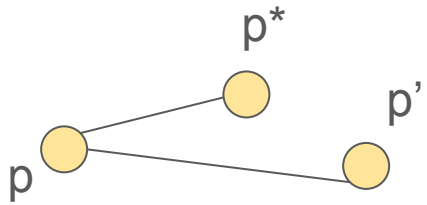
if $|N_{\text{out}}(p)| = R$ **then**

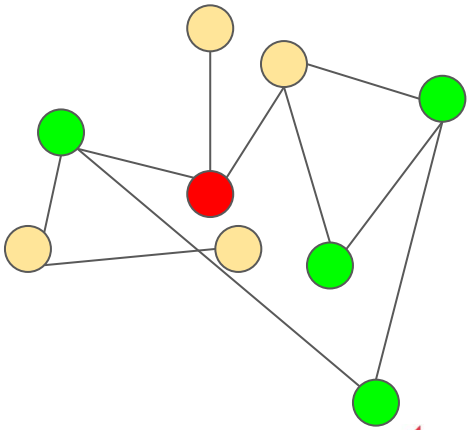
`break`

for $p' \in \mathcal{V}$ **do**

if $\alpha \cdot d(p^*, p') \leq d(p, p')$ **then**

`remove` p' `from` $\mathcal{V}$





CUBRID

The **Vamana** Graph Construction Algorithm

Vamana Indexing Algorithm

- Vamana: directed graph G를 iterative manner로 구축
  - 초기화
    - 각 vertex가 R개의 랜덤 이웃을 가지는 정규 그래프로 초기화
    - dataset P의 medoid s를 시작노드로 설정
  - Step 1: p ∈ P 인 모든 점들을 무작위 순서로 순회하며, 각 단계에서 그래프를 업데이트하여 GreedySearch(s, x_p, 1, L)이 p에 수렴하도록 점차 개선
    - 1) GreedySearch (s, x_p, 1, L)을 실행하며 탐색 중 방문한 모든 점을 V_p에 수집
    - 2) RobustPrune (p, V_p, a, R)을 실행하여 G를 업데이트
    - 3) 모든 p` ∈ Nout(p) 에 대해서 backward edge (p` -> p)를 추가하여 그래프 G를 갱신
      - 탐색 경로의 vertex 들과 p 사이의 연결성 보장
      - p`의 degree가 R을 초과하는 경우, RobustPrune (p`, N_out(p`), a, R)을 실행
  - Step 2: Step1의 전체 과정을 사용자 정의 a > 1로 한번 더 수행
    - 단조적으로 메도이드로부터 줄어드는 간선 중, 멀리 있는 에지를 남기기 위함

CUBRID

The **Vamana** Graph Construction Algorithm

Vamana Indexing Algorithm

Algorithm 3: Vamana Indexing algorithm

Data: Database P with n points where i-th point has coords x_i, parameters α, L, R
Result: Directed graph G over P with out-degree <=R
begin
  initialize G to a random R-regular directed graph
  let s denote the medoid of dataset P
  let σ denote a random permutation of $1..n$
  for $1 \leq i \leq n$ **do**
    let $[\mathcal{L}; \mathcal{V}] \leftarrow$ GreedySearch$(s, x_{\sigma(i)}, 1, L)$
    run RobustPrune$(\sigma(i), \mathcal{V}, \alpha, R)$ to update out-neighbors of $\sigma(i)$
    for *all points* j *in* $N_{\text{out}}(\sigma(i))$ **do**
      if $|N_{\text{out}}(j) \cup \{\sigma(i)\}| > R$ **then**
        run RobustPrune$(j, N_{\text{out}}(j) \cup \{\sigma(i)\}, \alpha, R)$ to update out-neighbors of j
      else
        update $N_{\text{out}}(j) \leftarrow N_{\text{out}}(j) \cup \sigma(i)$

CUBRID

Step 1

Step 2

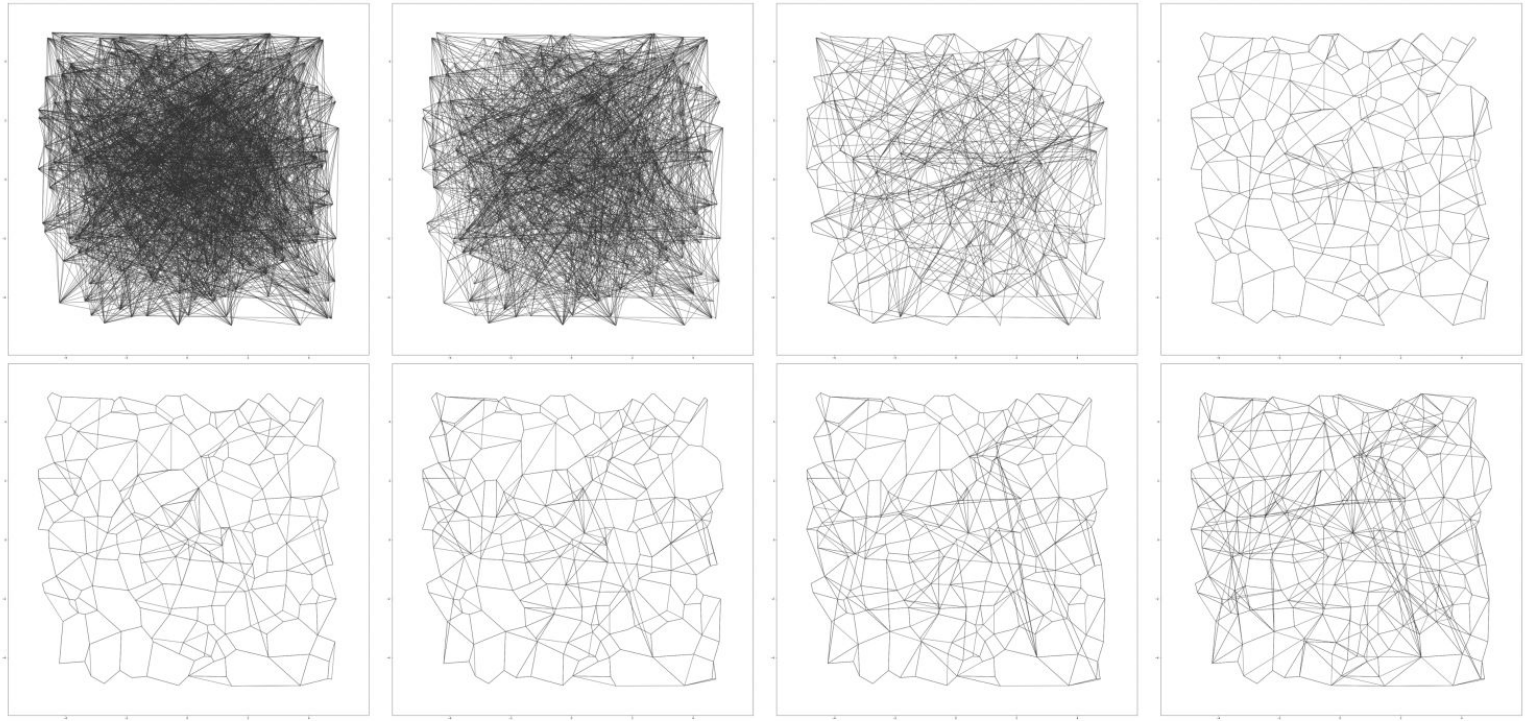


Figure 1: Progression of the graph generated by the Vamana indexing algorithm described in Algorithm 3 on a database with 200 points in 2 dimensions. Notice that the algorithm goes through the first pass with $\alpha = 1$, followed by the second pass where it introduces long range edges.

CUBRID

The **Vamana** Graph Construction Algorithm

Comparison of **Vamana** with HNSW and NSG

- HNSW
  - NSW는

DiskANN: Constructing SSD-Resident Indices

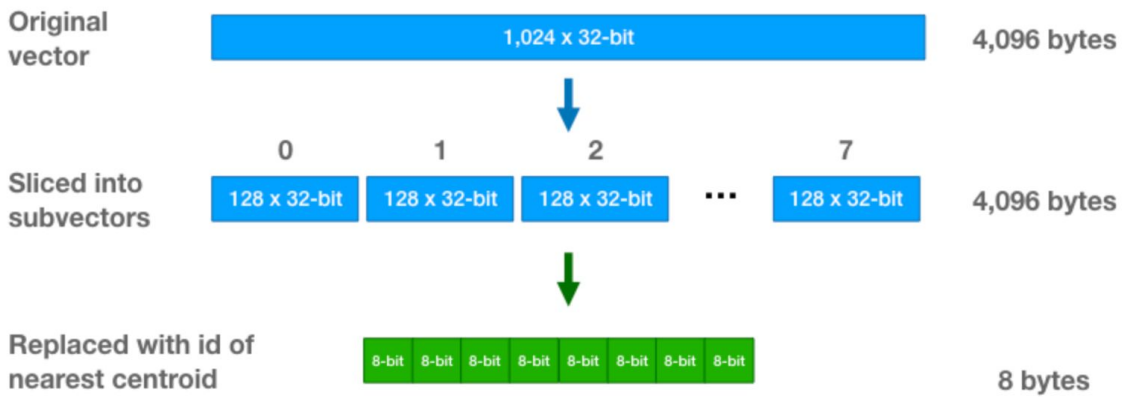
The DiskANN Index Design

- DiskANN의 핵심 아이디어
  - 데이터셋 P에 대해 Vamana를 실행
  - 그 결과 그래프를 SSD에 저장
  - 검색 시 GreedySearch 시 점 p의 out-neibors가 필요할 때마다 SSD에서 해당 정보를 읽어오기
- ----------
- 기존 방법들의 한계
  - 100차원에서 10억개의 점을 저장하는 것은 RAM 용량을 훨씬 초과하게 됨 (4 * 100 * 1B ~= 400G)
- 핵심 질문
  - 1) 어떻게 하면 10억개의 점에 대한 그래프를 구축할 것이고
  - 2) 전체 벡터 데이터를 메모리에 저장할 수 없다면, 어떻게 검색 시 벡터 간 거리를 계산할까?
- 제안하는 해결 방법
  - 1) 각 데이터 포인트를 여러 개의 centroid로 할당해 overlapping cluster를 만들고
    각 클러스터에 할당된 점들에 대해 Vamana 인덱스 구축한 뒤 단순한 간선 합집합 형식으로 병합
  - 2) 각 데이터 포인트 $p \in P$ 에 대해 압축된 벡터를 메인 메모리에 저장, 그래프는 SSD에 저장
    그래프 구축 시에는 full-precision vector, 검색 시에는 compressed vector 사용

CUBRID

PQ (Product Quantization)

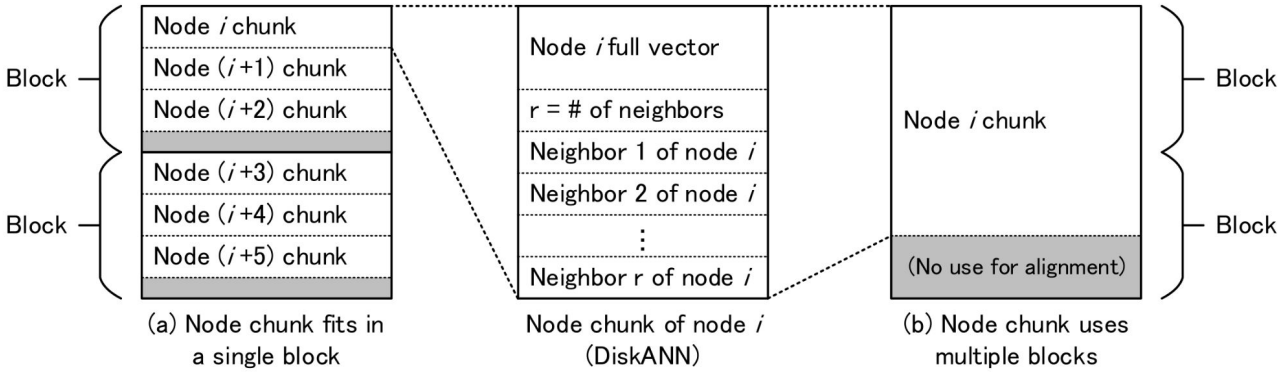
- 원래 벡터를 여러 개의 저차원 부분 벡터(sub-vector)로 분할하고, 각 부분을 **사전(codebook)에서 가장 가까운 중심점(centroid)**으로 치환.
- centroid는 각 부분을 K-means로 학습하여 코드북 생성
- 쿼리 벡터에 대해서 각 서브 벡터와 centroid와의 거리로 codebook 값 설정



DiskANN: Constructing SSD-Resident Indices

The DiskANN Index Layout

- 모든 데이터의 compressed vector는 메인 메모리에 저장하고 그래프와 full-precision vector는 SSD에 저장
- 각 점 i에 대해, 그 점의 full-precision vector x_i를 저장한 뒤, 최대 R개의 이웃의 ID를 이어서 저장
- 어떤 노드의 degree가 R 보다 작으면 0으로 패딩



CUBRID

DiskANN: Constructing SSD-Resident Indices

The DiskANN Beam Search

- 질의 벡터 x_q에 대해서 최근접 이웃을 찾을 때 GreedySearch로, 필요할 때 마다 SSD에서 이웃 집합 N_out(p*)를 읽어오는 것이 자연스러운 방식.
  - SSD round-trip이 많아져 latency가 커진다.
- neighbor 정보를 순차적으로 가져오는 대신, L \ V에서 W (4개, 8개) 를 한번에 읽어온다. 새로 가져온 이웃을 포함해 상위 L개의 후보를 다시 L로 갱신
  - **SSD에서 작은 수의 랜덤 섹터를 읽는 것은 한 섹터를 읽는 시간과 거의 비슷**
- SSD의 I/O 요청 큐를 포화 상태로 만들어 최대 읽기 처리량을 끌어낼 수 있으나, 디스크 읽기 지연 시간이 1 밀리초를 넘어버리는 문제, 따라서 더 낮은 부하로 동작 시키는 것이 필요
- 낮은 Beam Width (W) 값을 2, 4, 8로 동작하는 것이 latency와 throughput 사이의 균형을 잘 맞춰주므로, 이 설정에서 SSD load factor가 30~40%, 검색을 하는 각 스레드의 I/O 처리 시간이 40~50%를 차지하는 것을 보임

CUBRID

DiskANN: Constructing SSD-Resident Indices

DiskANN Implicit Re-Ranking Using Full-Precision Vectors

Algorithm 1 DiskANN Beam Search with re-ranking

Data: Graph G with entrypoint s, query q, number of top candidates k, beamwidth w,and search list size $L \geq k$
Result: *PQ compressed* node set $\mathcal{L}$ and *full-precision* node set $\mathcal{V}$, each containing top-k neighbor candidates of q
$\mathcal{L} \leftarrow \{s\}, \mathcal{V} \leftarrow \emptyset$
while $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ **do**
  let $\mathcal{P}$ denote top-w closest nodes to q in $\mathcal{L}$
  for $p \in \mathcal{P}$ **do**
    get node chunk of p from storage
  end for
  append *PQ vectors* of $N_{out}(\mathcal{P})$ from RAM to $\mathcal{L}$
  append *full-precision vectors* of $\mathcal{P}$ from node chunk to $\mathcal{V}$
  $\mathcal{L} \leftarrow \mathcal{L} \cup N_{out}(\mathcal{P}), \mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{P}$
  if $|\mathcal{L}| > L$ **then**
    set $\mathcal{L}$ with top-L closest nodes in *PQ space* to q in $\mathcal{L}$
  end if
end while
sort $\mathcal{V}$ by their *full-precision* distance to q
return top-k closest nodes to q in $\mathcal{V}$

CUBRID

Evaluation

- 실험 머신
  - z840: a bare-metal mid-range workstation with dual Xeon E5-2620v4s (16 cores), 64GB DDR4 RAM, and two Samsung 960 EVO 1TB SSDs in RAID-0 configuration.
  - M64-32ms: a virtual machine with dual Xeon E7-8890v3s (32-vCPUs), 1792GB DDR3 RAM that we use to build a one-shot in-memory index for billion point datasets.
- 실험 비교 대상
  - HNSW
  - NSG

CUBRID

Evaluation

In-Memory Search Performance

- SIFT1M (128 dim), GIST1M (960 dim), DEEP1M (96 dim)
- 세 알고리즘 모두 최적의 파라미터를 선택 (by parameter sweep)
  - HNSW: M = 128, efC = 512
  - NSG
    - SIFT1M, GIST1M은 공식 레포지토리 설정 사용
    - DEEP1M에서는 R = 60, L = 70, C = 500
  - Vamana: R = 70, L = 75, α = 1.2
- 본 연구의 초점은 SSD 기반 탐색이므로, Vamana를 위한 별도의 탐색 알고리즘을 구현하지 않고, 대신 NSG 레포지토리의 최적화된 탐색 알고리즘을 Vamana 인덱스에 적용

CUBRID

Evaluation

In-Memory Search Performance

- VAMANA 그래프의 구조가 인-메모리 환경에서 충분한 검색 성능을 보임

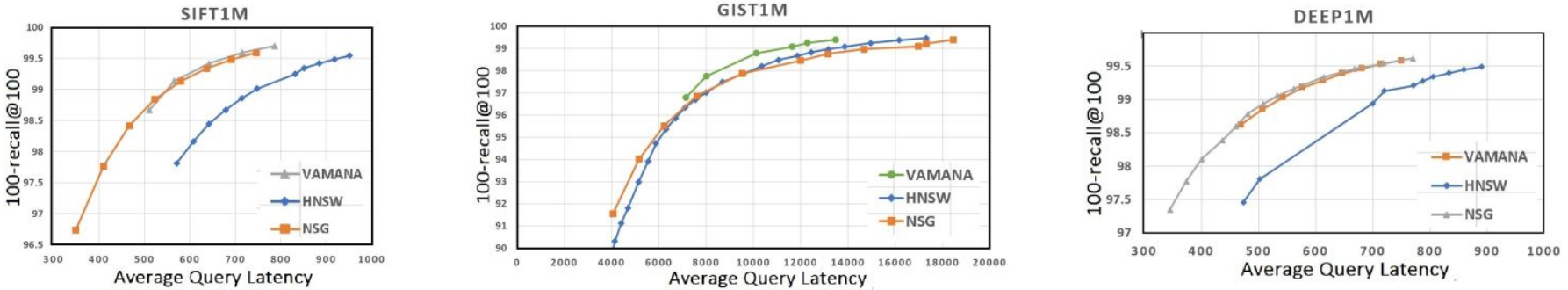


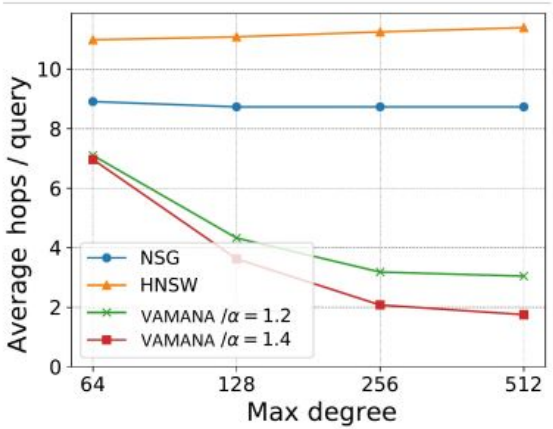
Figure 3: Latency (microseconds) vs recall plots comparing HNSW, NSG and Vamana.

CUBRID

Evaluation

of Hops

- # of hops 수는 디스크 읽기 수와 비례
- HNSW와 NSG에서는 hop 수 감소가 일정 수준에서 정체되는 경향을 보이는 반면, Vamana는 최대 차수(R)와 α가 커질수록 hop 수가 줄어드는 것을 확인

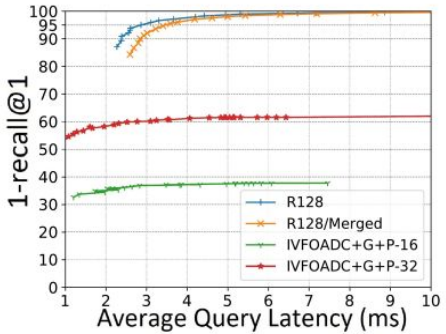


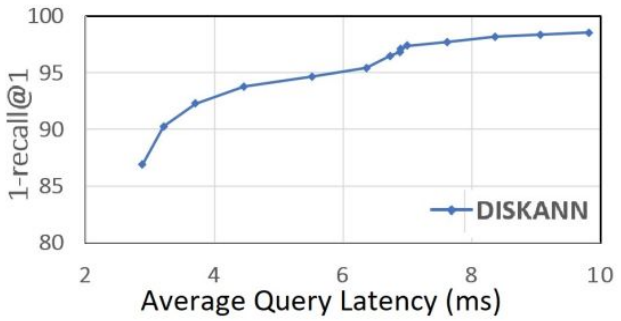
CUBRID

Evaluation - Comparison on Billion-Scale Datasets

One-Shot Vamana vs Merged Vamana

- 전체 10억 포인트 데이터셋에 대해 단일 인덱스 vs 여러 인덱스를 샤드로 구성
- Merged Vamana 구축 방법
  - 점들을 k-means 클러스터링으로 여러 샤드로 나눔
  - 각 샤드의 점들을 주변 가장 가까운 점에 중복해서 저장
  - 각 샤드에 대해서 Vamana 그래프 구축
  - 중복된 점을 통해 샤드들은 연결됨
- 단일 인덱스 대비, 동일한 목표 recall에 대해 20% 이내의 추가 latency 정도 차이





CUBRID

AiSAQ: All-in-Storage ANNS with Product Quantization for DRAM-free Information Retrieval

Background

- 억 단위 데이터셋의 경우에 인덱스 검색을 위해 DRAM에 저장하는 경우에 **비용**이 매우 크기 때문에 SSD 스토리지 활용 필요
- RAG에서 다수의 외부 지식 소스를 필요로 할 때 **여러 인덱스간 스위치**가 필요
- 모든 인덱스를 RAM에 상주시키거나, 요청마다 스토리지에 접근하는 문제

CUBRID

Introduction

- DiskANN은 디스크 기반 ANN의 **de facto standard**
- 모든 PQ 압축 벡터를 RAM에 유지하므로, 그 압축률에 따라 데이터셋의 크기에 대한 진정한 확장성을 가지지 못함
- 여러 인덱스간 전환도 DRAM 로딩 시간의 문제 발생
- 이 문제로 AiSAQ는 DRAM 사용량을 데이터셋의 크기에 관계없이 약 10M로 유지하고, 지연 시간은 미미하게 증가하도록 수정

CUBRID

Contributions

- 억 단위 SIFT1B 데이터셋을 포함한 어떤 크기의 데이터셋에서도, RAM 사용량 ~10MB로 DiskANN 수준의 높은 Recall 달성.
- SSD 데이터 배치 최적화 덕분에, 작은 메모리 사용량으로도 95% 이상의 1-recall@1을 밀리초 수준 지연 시간으로 달성.
- 질의 검색 전 인덱스 로드 시간이 사실상 무시할 수 있을 정도로 줄어, 여러 대규모 인덱스 간 전환 가능.
- 동일한 벡터 공간을 공유하는 여러 데이터셋 간 PQ 중심(centroid) 공유로 전환 시간이 서브밀리초 수준까지 단축.
- 다중 서버 검색 환경에서 DRAM+SSD 총 비용 최적화를 실현.

CUBRID

Preliminaries

- Graph-Based ANN Algorithms
- Index Switch for Multiple Source
- DiskANN

Proposed Method

- PQ 벡터들도 노드 청크에 함께 저장
- 하나의 블록 (페이지)의 크기를 넘지 않도록 R의 크기를 조절

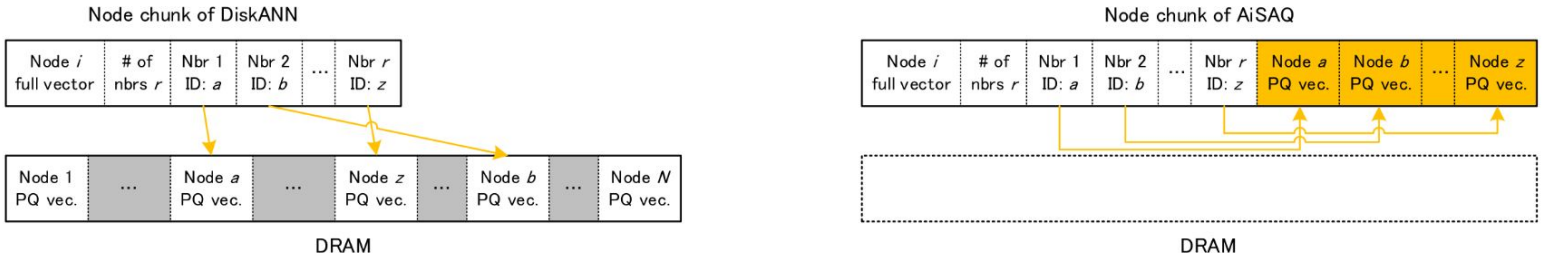


Figure 2: Data placements of a node chunk and memory of DiskANN (left) and proposed method AiSAQ (right)

Memory Usage and Index load time

- 엔트리포인트는 어차피 읽을 것이므로 엔트리포인트의 PQ
- 각 hop을 읽을 때의 R 차수 만큼의 PQ

Table 2: Memory Usage (MB) by DiskANN and AiSAQ

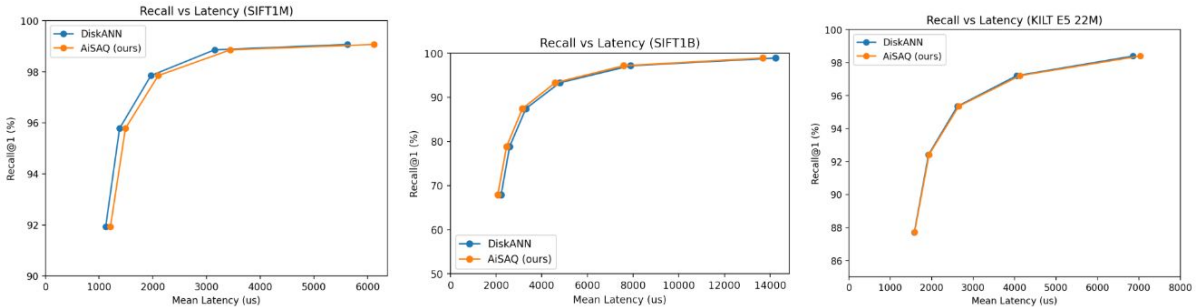
	b_{PQ}	DiskANN	AiSAQ (ours)
SIFT1M	128 bytes	146	11
SIFT1B	32 bytes	31,303	11
KILT E5 22M	128 bytes	2,803	14

Table 3: Index load time (ms) of DiskANN and AiSAQ

	DiskANN	AiSAQ (ours)
SIFT1M	46.8	0.6
SIFT1B	16,437.4	0.6
KILT E5 22M	1,121.4	2.0

CUBRID

Performance (Recall, Latency, Memory Usage)



(a) SIFT1M (b) SIFT1B (c) KILT E5 22M

Figure 3: Recall vs latency graphs of DiskANN and AiSAQ

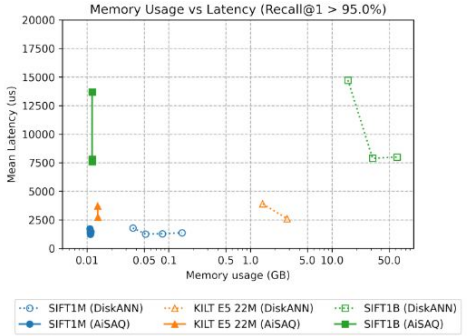


Figure 4: Latency vs memory usage of DiskANN and AiSAQ

CUBRID

FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search

Introduction

- 최근접 탐색(Nearest Neighbor Search, NNS) 문제는 주어진 데이터셋 P와 어떤 거리함수가 있을 때, 질의점 q에 대해 가장 가까운 k개의 이웃을 효율적으로 찾는 자료구조를 설계
- 차원의 저주로 정확도와 하드웨어 자원의 균형을 맞추는 ANN 알고리즘이 대세
- **실제 운영 환경에서 발생하는 데이터의 삽입과 삭제를 다루지 못함**
- 실시간으로 데이터 변화가 반영된 인덱스가 필요
- Fresh-ANN 문제
  - 시간에 따라 변하는 데이터셋 P
  - 특정 시점 t의 활성 데이터셋 P_t에 대한 ANN 탐색
- Fresh-ANN 시스템 요구사항
  - 삽입, 삭제, 검색을 모두 지원하며 검색 성능, 삽입/삭제 처리량, 하드웨어 비용 등을 평가 만족

Discussion on implementing a disk-based ANN index in CUBRID