Vector Database Management Systems

James Jie Pan · Jianguo Wang · Guoliang Li

@hgryoo

CUBRID

2025-05-29

CUBRID

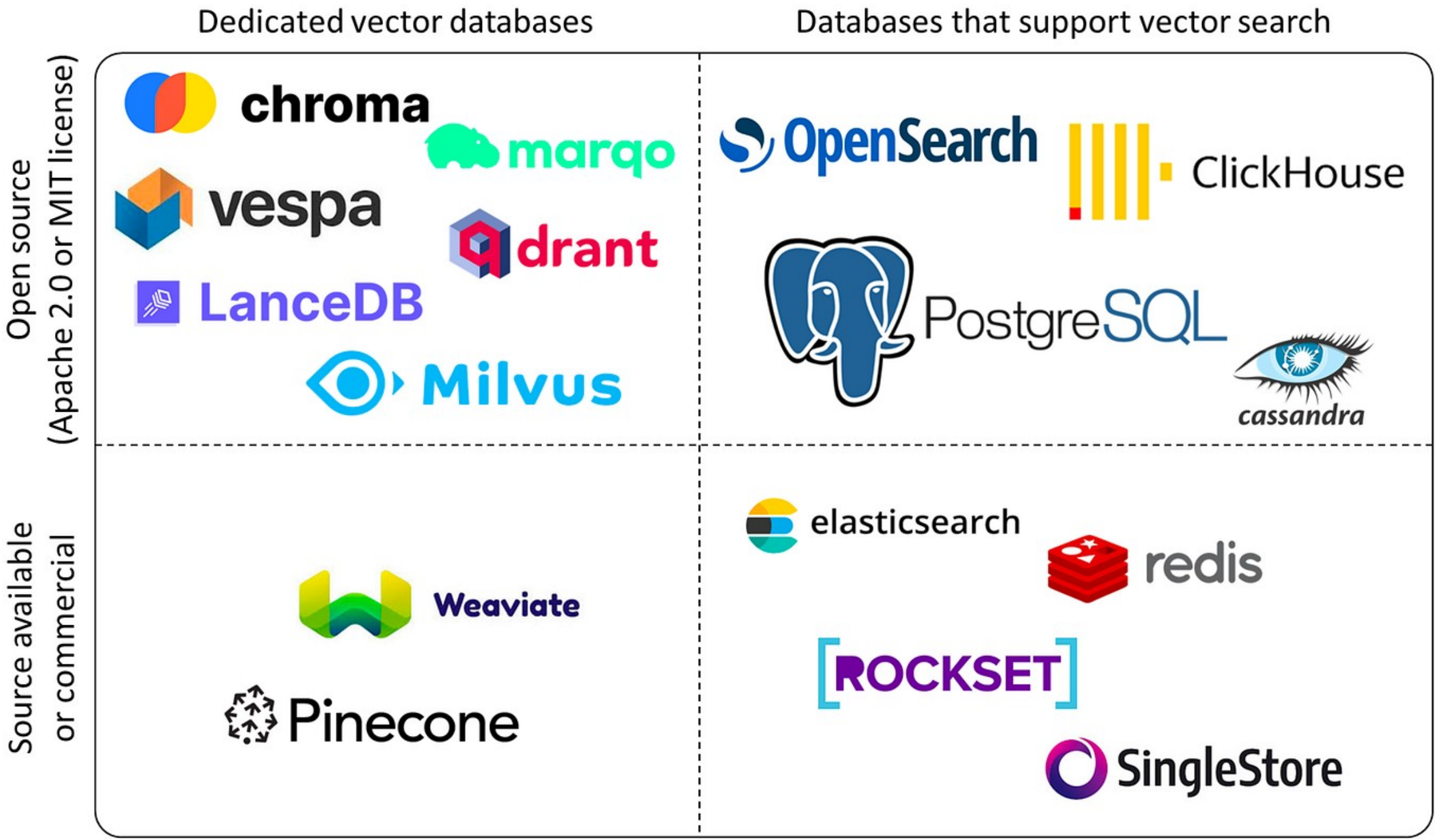
CONTENTS

CUBRID

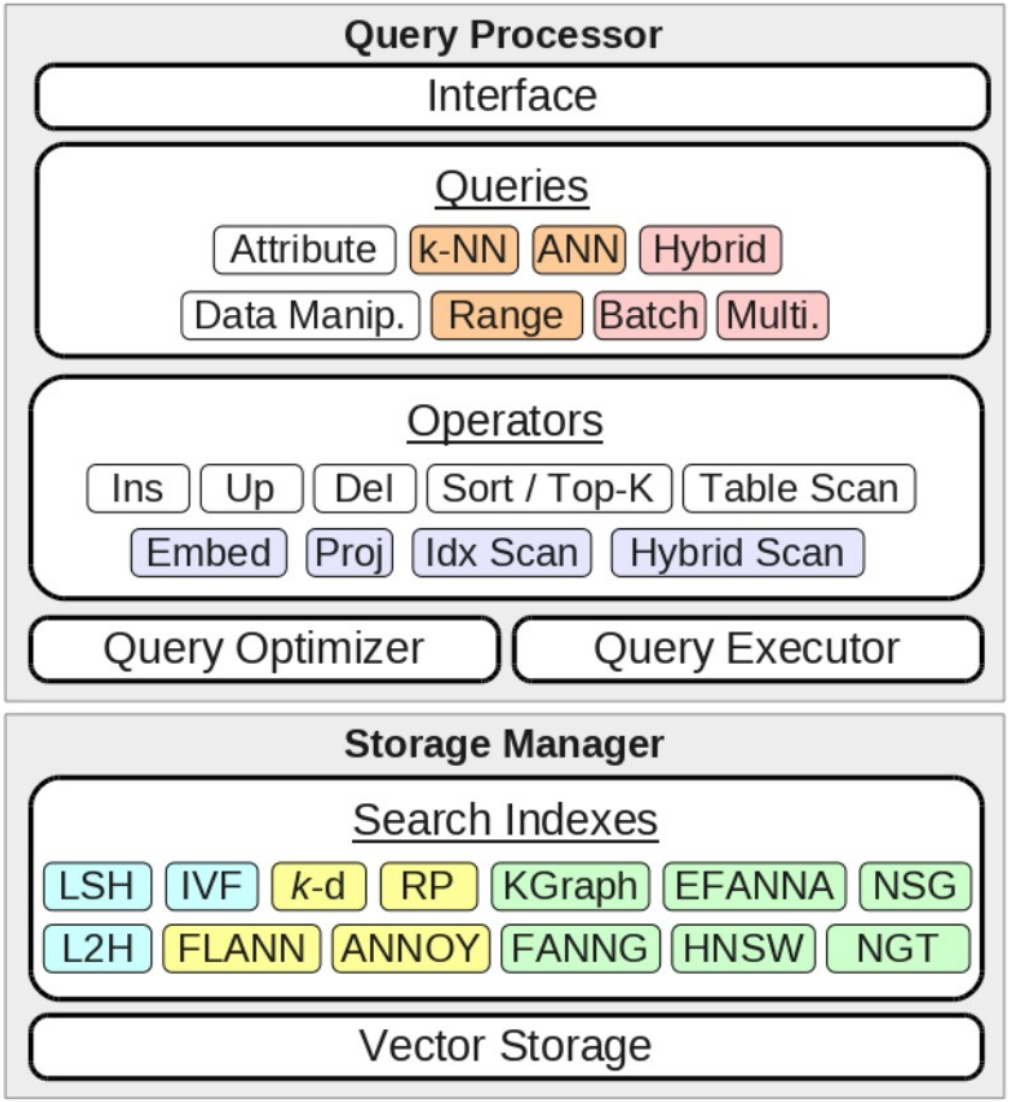
Introduction

- High-dimensional feature vectors: LLMs, e-commerce, recommendation, document retrieval ...
- Requirements: billions of vectors and millisecond query latency
- Traditional DBMS (NoSQL, RDBMS) are not designed for these datasets and workloads
  1. Vector queries rely on the concept of **similarity**
  2. Similarity **computation** is more expensive than other types of comparisons
  3. **The cost of retrieval** is more expensive compared to simple attributes (memory, disk)
  4. Lack obvious properties that can be used **for indexing** (sortable or ordinal)
  5. **Hybrid queries** require accessing both attributes and vectors together

CUBRID


Dedicated vector databases
Databases that support vector search
Open source
(Apache 2.0 or MIT license)
Source available
or commercial
chroma
marqo
vespa
drant
LanceDB
Milvus
OpenSearch
ClickHouse
PostgreSQL
cassandra
elasticsearch
redis
ROCKSET
SingleStore
Weaviate
Pinecone

Architecture of a VDBMS



CUBRID

Current Systems

- Native: aim at providing dedicated vector capabilities.
  - Mostly-Vector: limited support for attribute-related capabilities (Vald, EuclidesDB, Pinecone, Chroma)
  - Mostly-Mixed
    - support a wider variety of queries and query plans including attribute-only
    - storage models, query optimization
    - Milvus, Qdrant, Manu, Weaviate ..
- Extended
  - NoSQL
  - Relational
- Libraries and Other systems
  - Search Engines
  - SPTAG

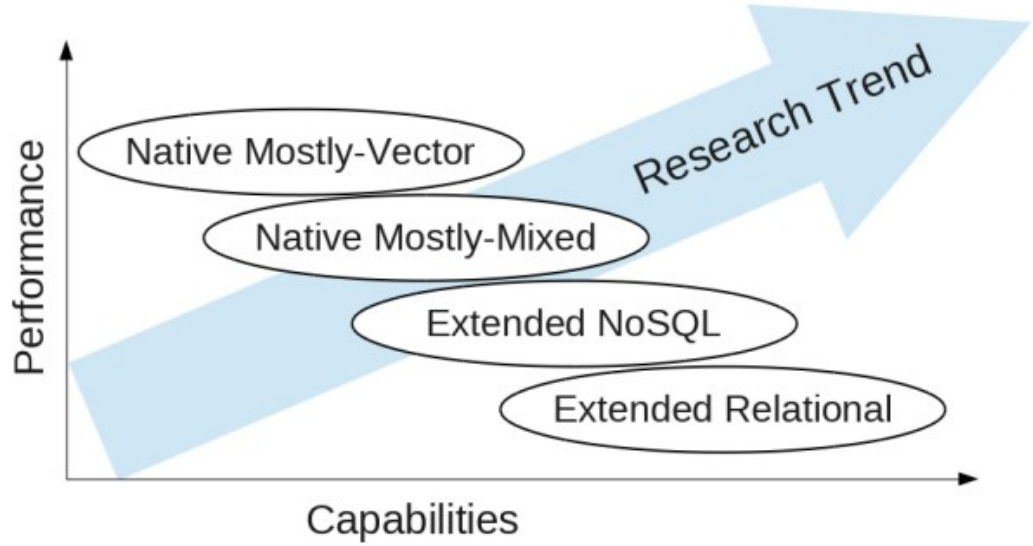


Fig. 14 High-level characteristics of VDBMSs.

Query Processing in VDBMS

1. Similarity Scores
2. Queries and Operators
3. Query Interfaces

CUBRID

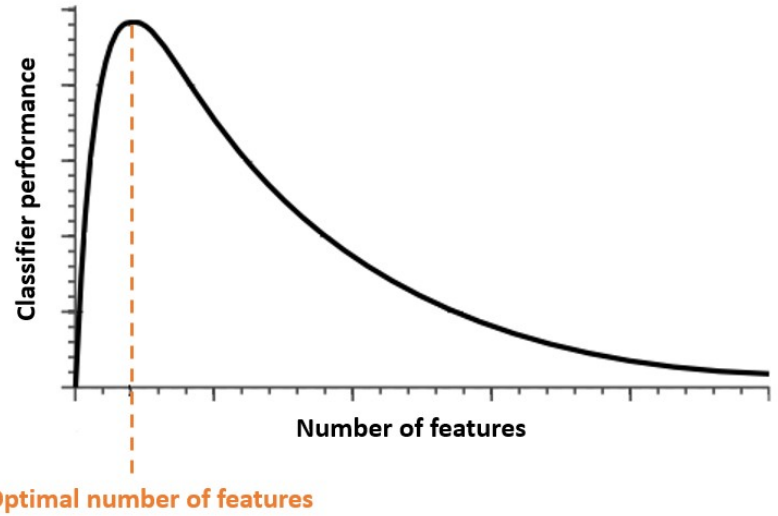
Similarity Scores

For two D-dimensional vectors

$$f : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}$$

A similarity score can be used to quantify the degree of similarity between two feature vectors

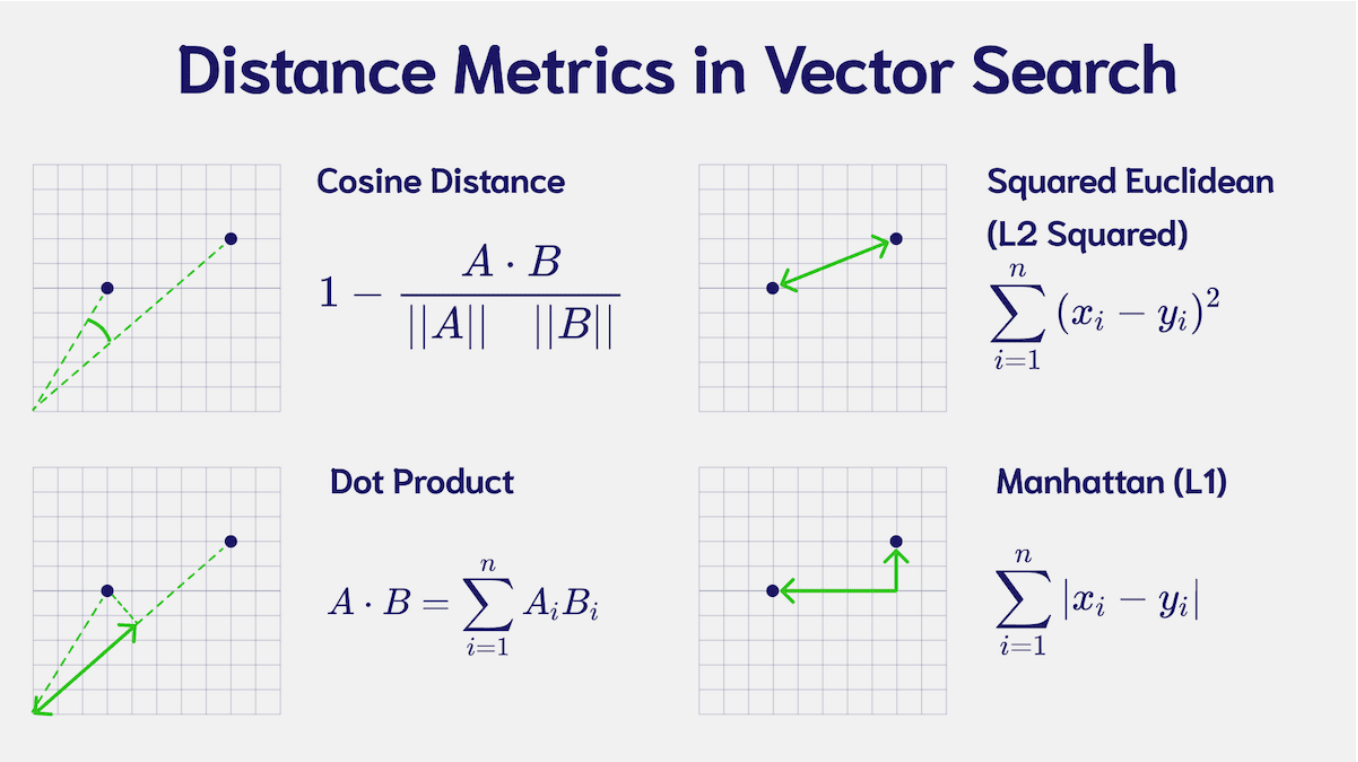
- Score Design
  - basic scores - Hamming, inner, cosine, Minkowski, Mahalanobis
  - aggregate scores - mean, weighted sum, other combine multiple scores
  - learned scores
- Score Selection
  - semantic similarity
  - curse of dimensionality

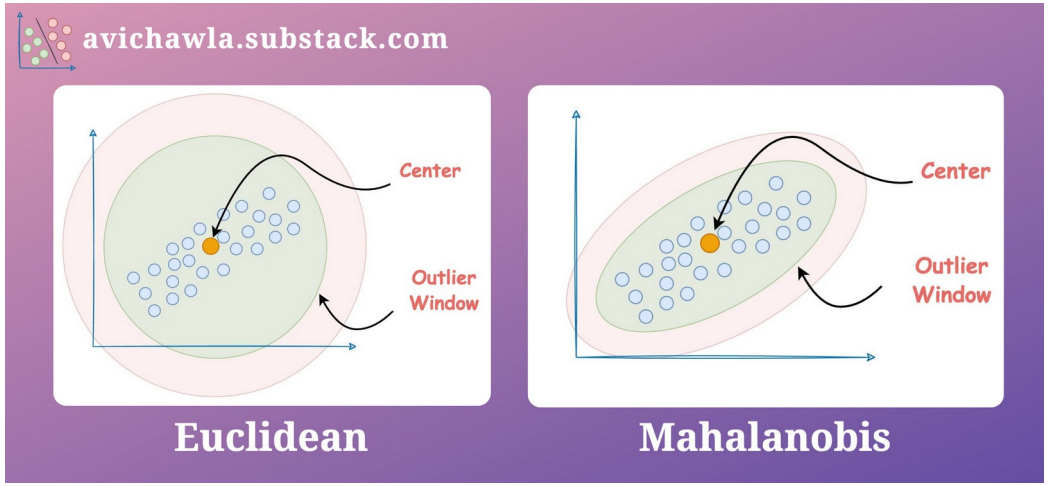


CUBRID

Basic Scores

- Hamming
- Inner Product
- Cosine Similarity
- Minkowski
- Mahalanobis

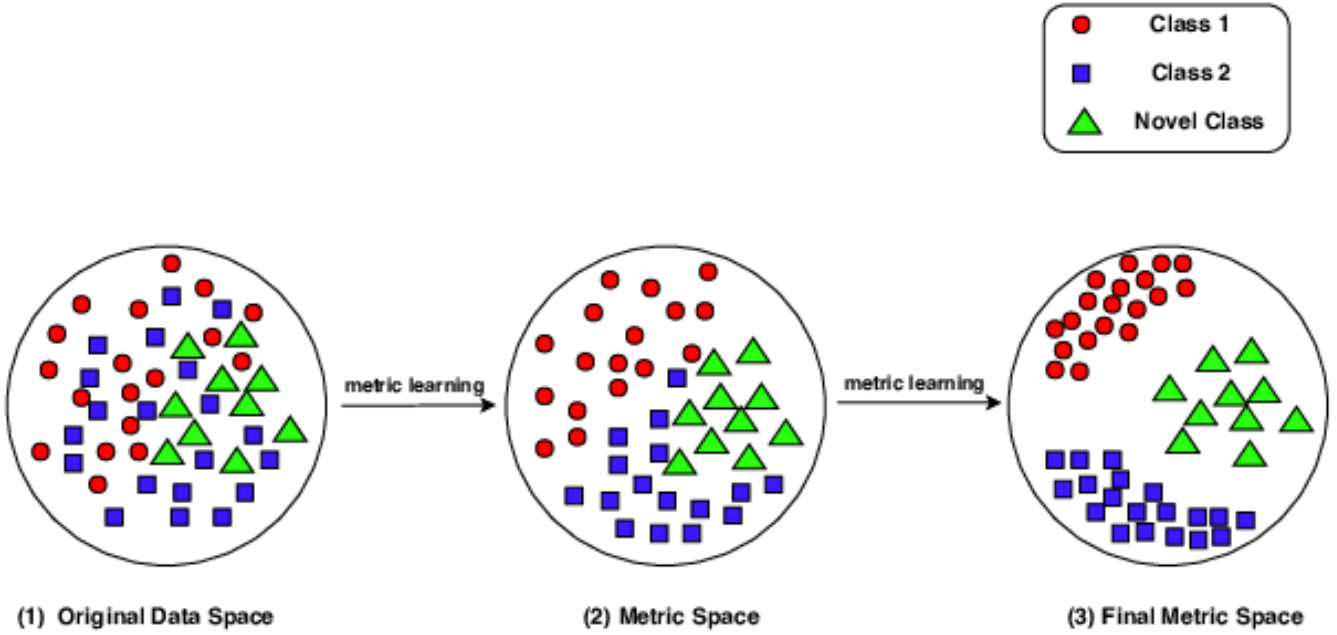




CUBRID

Learned Scores

- Metric learning: find a transformation **M**
  - informative input samples
  - structure of the network model
  - metric loss function



CUBRID

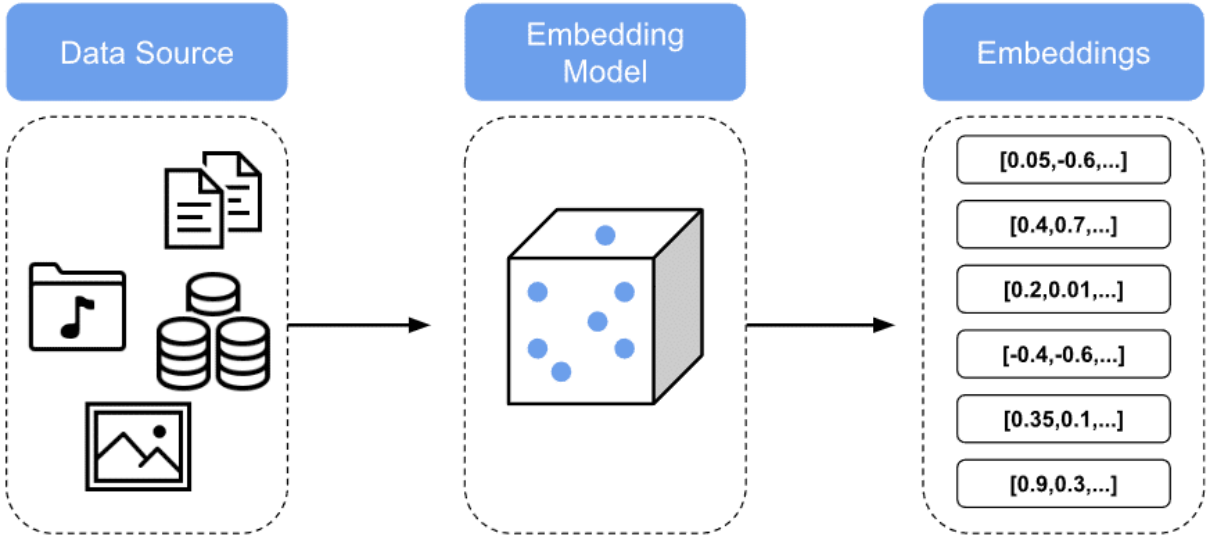
Queries and Operators

- Data Manipulation Queries
- Basic Search Queries
- Query Variants
- Basic Operators

CUBRID

Data Manipulation Queries

- DM queries provides insert, update and delete
- An embedding model maps real-world entities to feature vectors



CUBRID

Data Manipulation Queries

- The embedding model can live inside or outside the VDBMS
- Direct manipulation: users freely manipulate the values of the vectors
- Indirect manipulation: vectors are **hidden** from users

Vald

- UDF 에서 사용자의 임베딩 함수 등록
- 사용자가 " 텍스트 " 로 검색
- 내부적으로 Vald 가 벡터로 변환 후 검색 수행

Pinecone

- 사전 학습된 모델을 외부 서비스에서 사용
  - e.g.) OpenAI, HuggingFace ..
- 텍스트 입력
  -> API 통해 임베딩
  -> 벡터 저장과 검색

CUBRID

Basic Search Queries

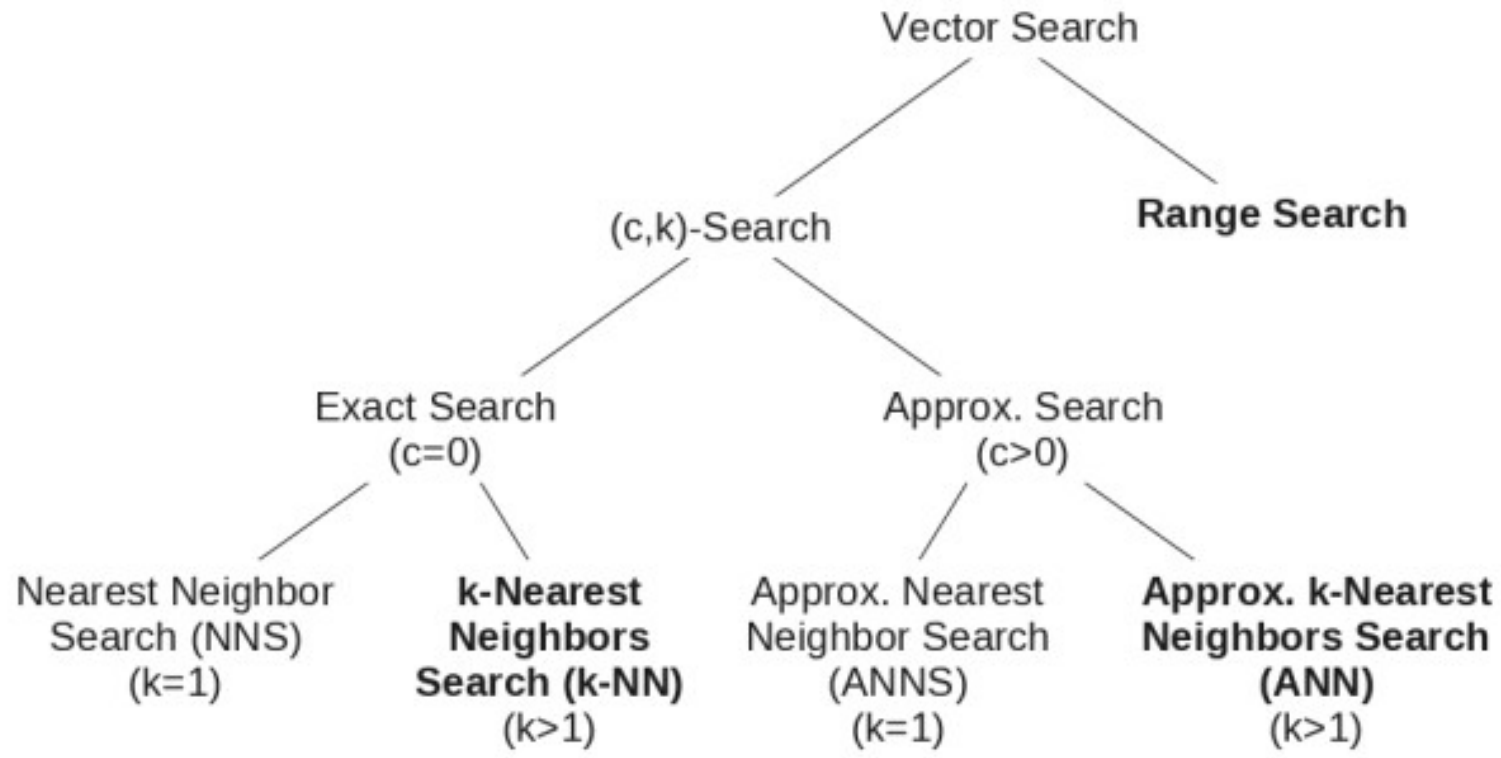


Fig. 2 Basic search queries.

CUBRID

Query Variants

- Predicated Search Queries

Each vector is associated with a set of attribute values

A hybrid k-NN query written in SQL:

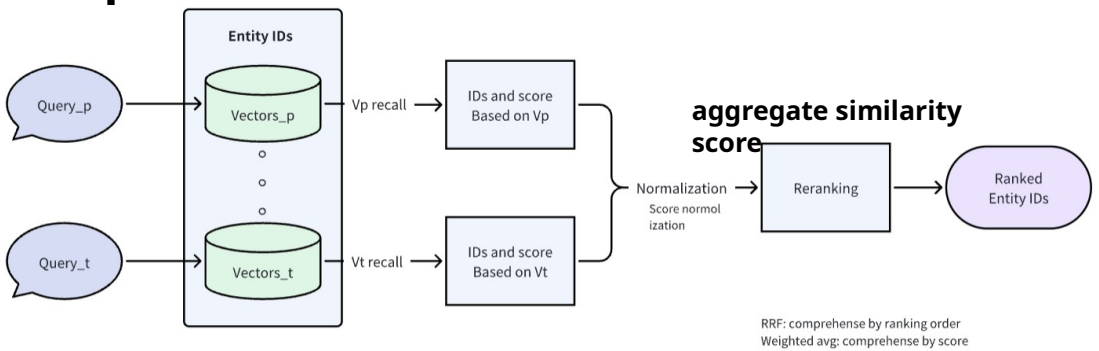
```
select * from S where attr < c
order by d(q) limit k;
```

- Batched Queries

- A number of queries are revealed to the system **at the same time**
- The VDBMS can answer them **in any order**

- Multi-Vector Queries

Example: Multi-vector search in Milvus



CUBRID

Batched Queries

- -- single query
  SELECT * FROM items ORDER BY l2_distance(vec, '[1,2,3]') LIMIT 3;

- -- batch query
  SELECT * FROM items ORDER BY l2_distance(vec, q.vec)
  FROM (VALUES ('[1,2,3]'), ('[4,5,6]'), ('[7,8,9]')) AS q(vec);

How to improve batch queries?

- Computation sharing: 같거나 유사한 벡터와의 중복 계산 개선
- Index sharing: 한번의 인덱스 검색으로 여러 질의에 함께 사용
- Candidate reuse: 공통 후보 벡터 집합을 만들고 , 각 질의에 대해 재검색

CUBRID

Query Interfaces

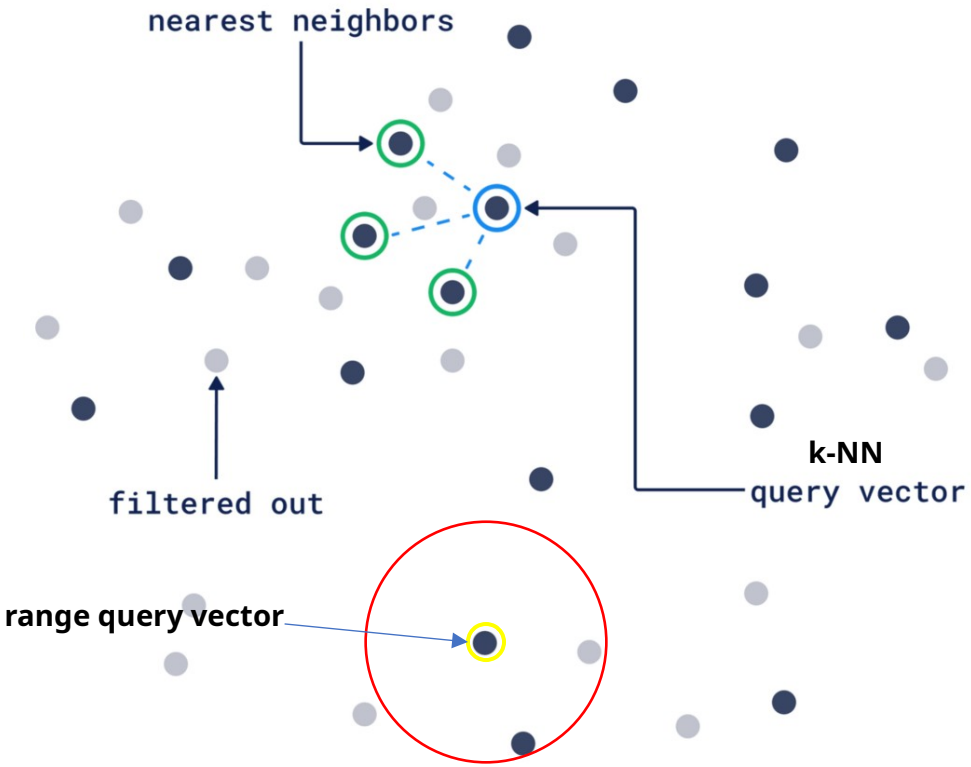
- Native and NoSQL VDBMSs - Small APIs (REST API, Python..)
- SQL extended VDBMSs - **SQL**

a k-NN query

```
select * from items order by
embedding <-> [3,1,2] limit 5;
```

a range query

```
select * from items where
embedding <-> [3,1,2] < 5;
```



CUBRID

Storage and Indexing

- How to organize and store the vector collection to support efficient and accurate search
- Partitioning Techniques
  - Randomization
  - Learned Partitioning
  - Navigable Partitioning
- Storage Techniques
  - Quantization
  - Disk-Resident Designs

CUBRID

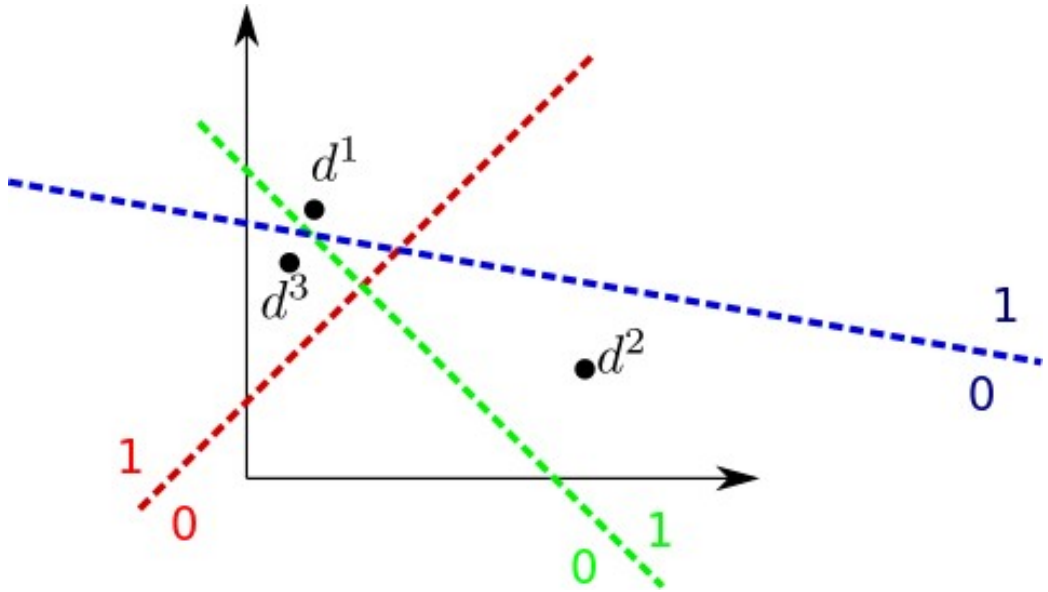
Vector search indexes

- table-based: easy to update
- tree-based: to provide logarithmic search
- graph-based: empirically well but with less theoretical understanding

CUBRID

Tables - Locality Sensitive Hashing

- Random Projection: 벡터 간의 거리를 (최대한) 보존하면서 저차원으로 변환하는 linear mapper
- LSH: RP 를 통해 저차원으로 변환한 벡터로 같은 버킷에 해시하여 k-ANN 을 수행



CUBRID

Tables - Locality Sensitive Hashing

- 예시
  - datasets
    - v1 [1, 2, 1]
    - v2 [2, 0, 1]
    - v3 [0, 1, 2]
    - v4 [3, 1, 0]
    - v5 [1, 0, 3]
  - parameters
    - k = 3 (한 테이블당 사용하는 hyperplane 수)
    - L = 2 (테이블의 갯수)
  - randomized hyperplane
    - TABLE1: r1 = [1, -1, 0], r2= [0, 1, -1], r3= [-1, 0, 1]
    - TABLE2: r4 = [1, 1, 1], r5 = [-1, 1, 0], r6 = [0, -1, 1]
  - hash function
    - $h_i(v) = 1$ if $\langle r_i, v \rangle \geq 0$ else 0

벡터	$g_1(v)$	$g_2(v)$
v_1	011	110
v_2	100	101
v_3	001	111
v_4	110	100
v_5	101	101

쿼리 예시

q = [1, 2, 0]

- $g_1(q) = \langle r_1,q\rangle,\langle r_2,q\rangle,\langle r_3,q\rangle = (-1, 2, -1) \rightarrow (0, 1, 0)$
- $g_2(q) = \langle r_4,q\rangle,\langle r_5,q\rangle,\langle r_6,q\rangle = (3, 1, -2) \rightarrow (1, 1, 0)$

테이블	코드	후보 벡터
1	010	(없음)
2	110	{v_1}

→ 후보집합 = {v_1}

→ 최종 거리 계산 후 K=1 반환: $\mathbf{v_1}$

벡터 ID	벡터 v	‖q‖	‖v‖	q · v	코사인 유사도 $\frac{q \cdot v}{\|q\|\|v\|}$	코사인 거리 1−유사도
v_1	[1, 2, 1]	√5 ≈ 2.236	√6 ≈ 2.449	5	5 / (2.236·2.449) ≈ 0.9129	0.0871
v_2	[2, 0, 1]	√5 ≈ 2.236	√5 ≈ 2.236	2	2 / (2.236·2.236) = 0.4000	0.6000
v_3	[0, 1, 2]	√5 ≈ 2.236	√5 ≈ 2.236	2	0.4000	0.6000
v_4	[3, 1, 0]	√5 ≈ 2.236	√10≈ 3.162	5	5 / (2.236·3.162) ≈ 0.7071	0.2929
v_5	[1, 0, 3]	√5 ≈ 2.236	√10≈ 3.162	1	1 / (2.236·3.162) ≈ 0.1414	0.8586

CUBRID

Learning to Hash

- **SKIP**

CUBRID

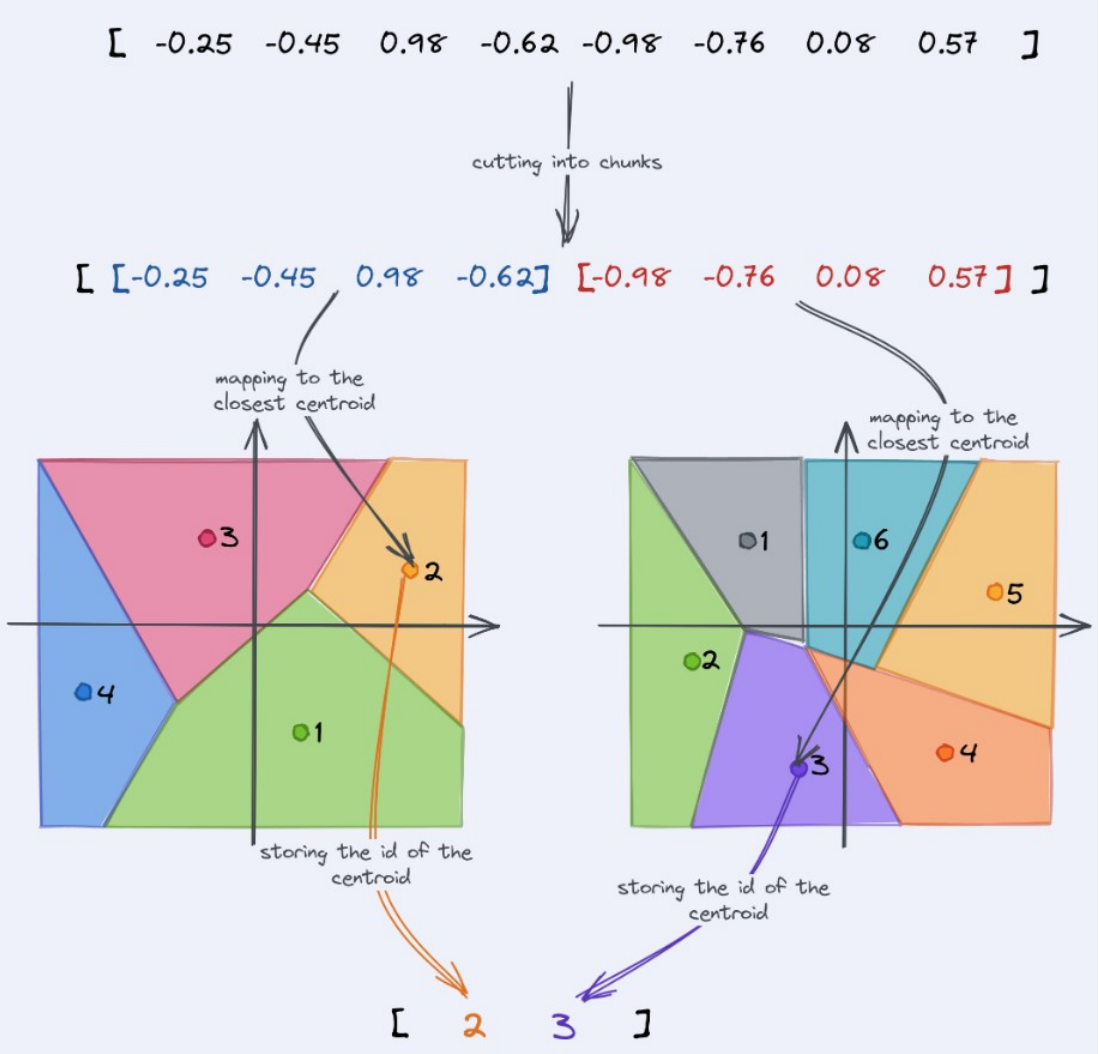
Tables - Quantization

- LSH shows high storage cost due to the use of multiple hash tables
- k-means
- product quantization
- scalar quantization

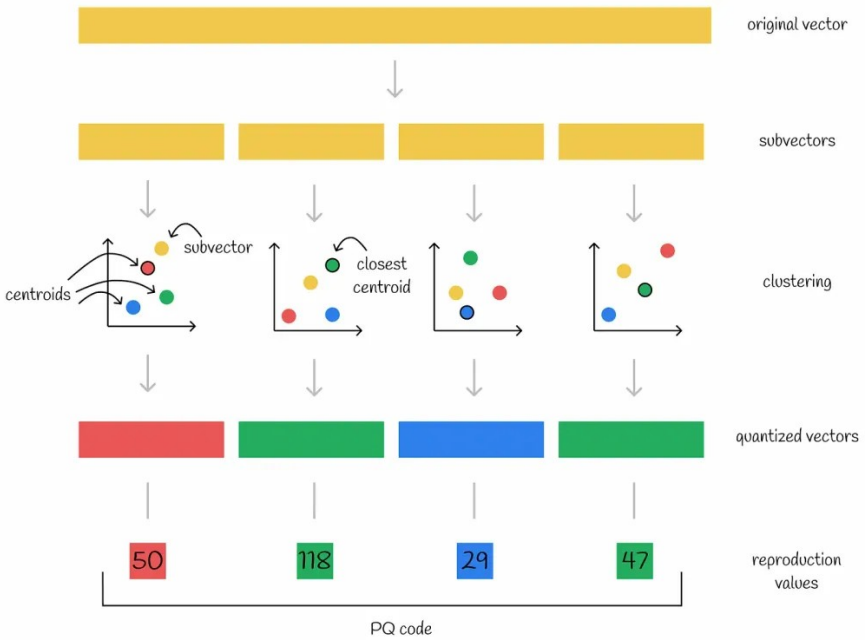
CUBRID

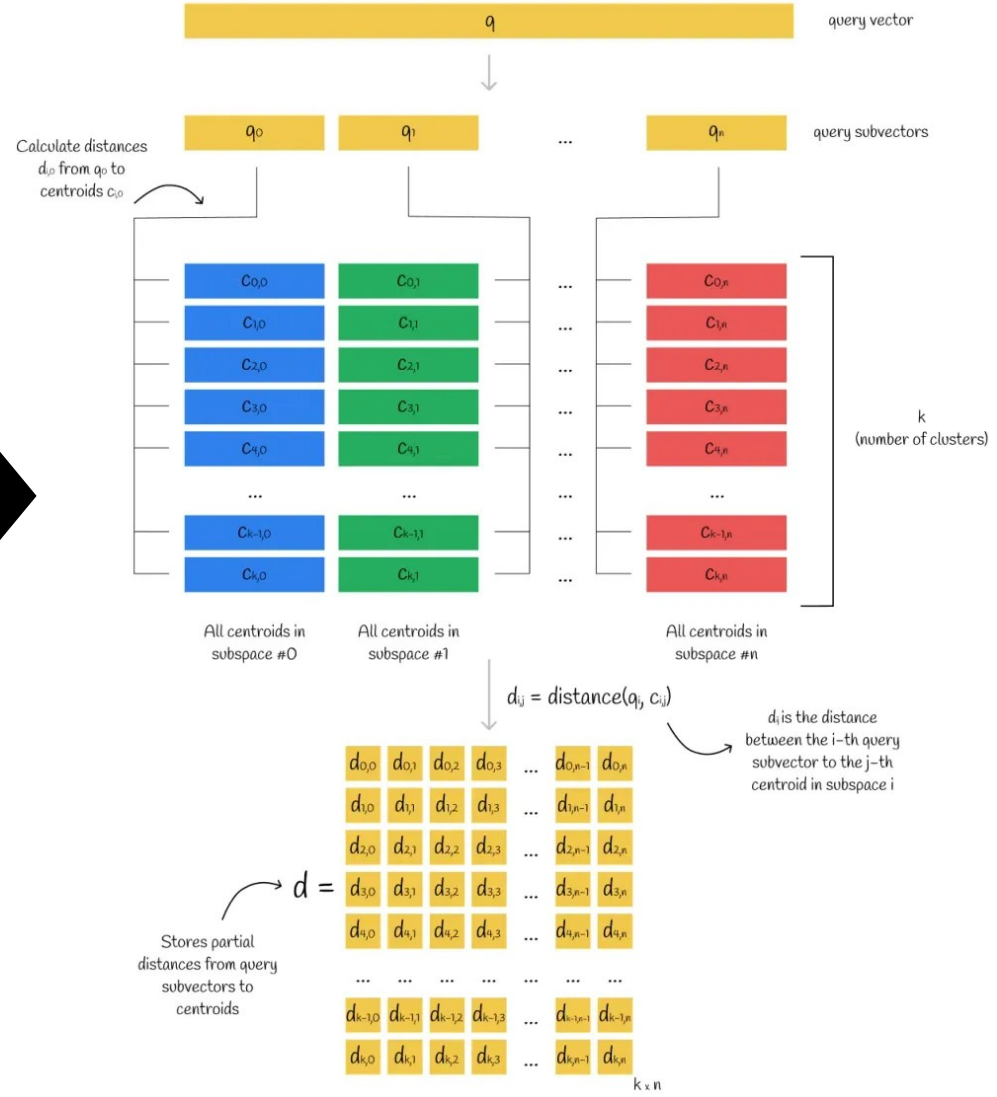
Tables - Product Quantization

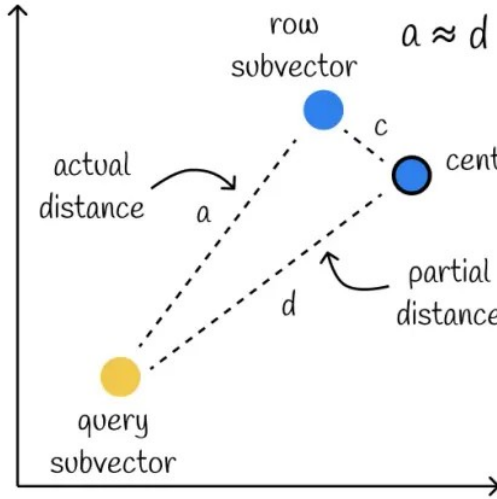
- lossy-compression
- set the k # of centroids (2^m)



Tables - Product Quantization

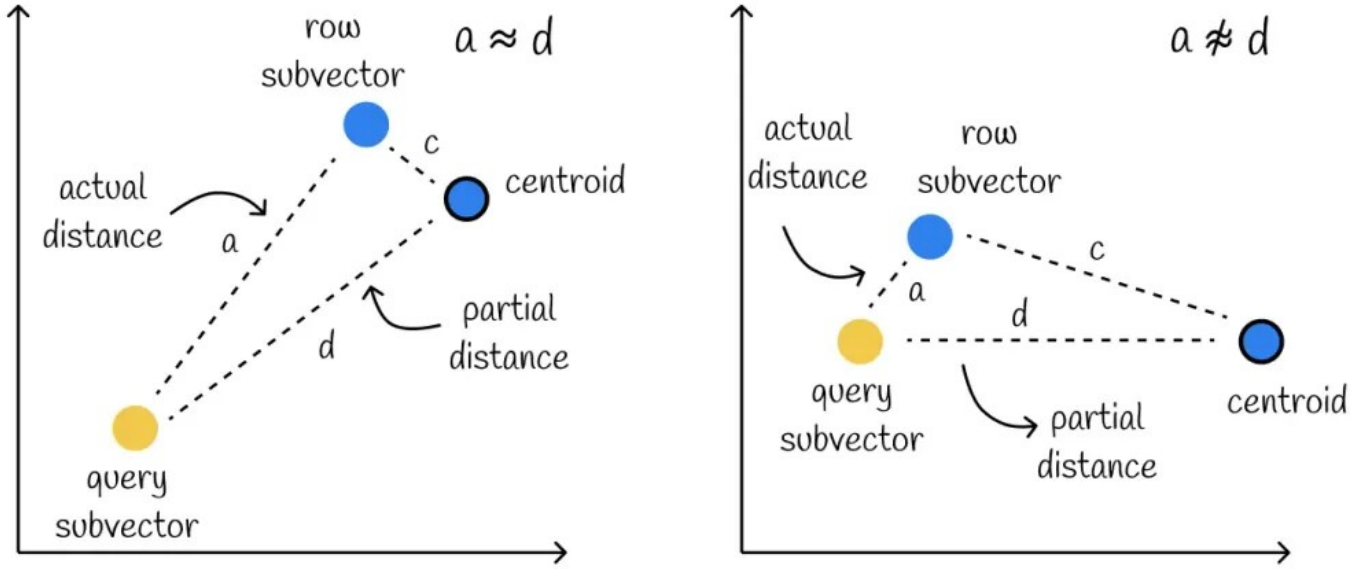






CUBRID

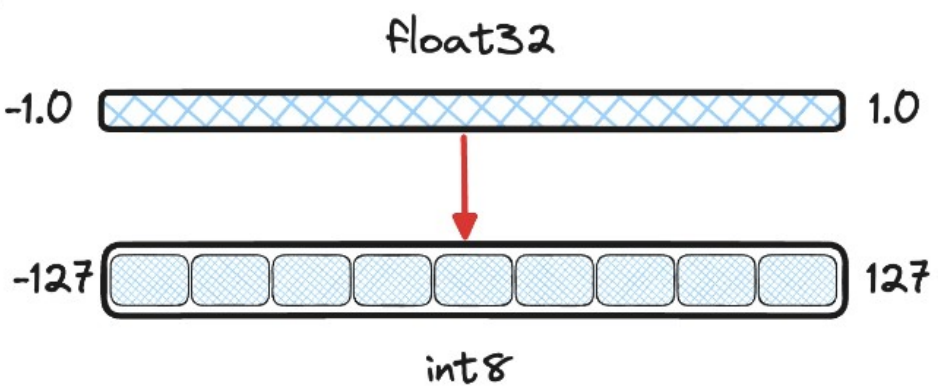
Tables - Product Quantization

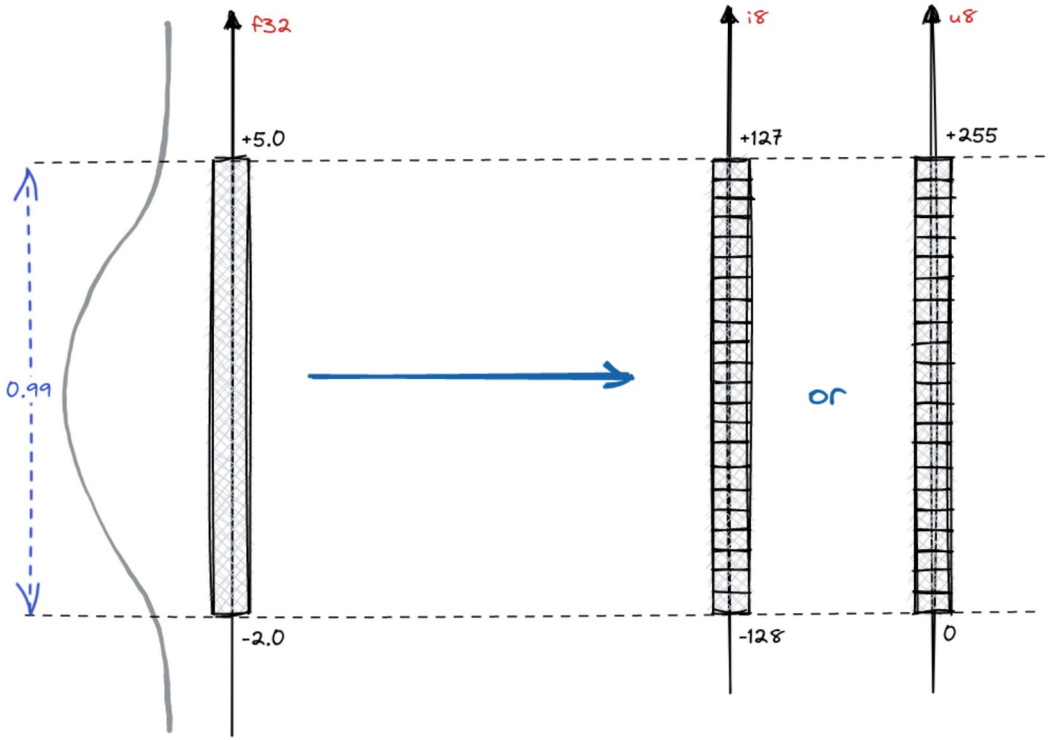


CUBRID

Tables - Scalar Quantization

- float 형식의 벡터 요소들을 integer 로 변환
- 데이터 분포에 따라 적절하게 변환하기도 함

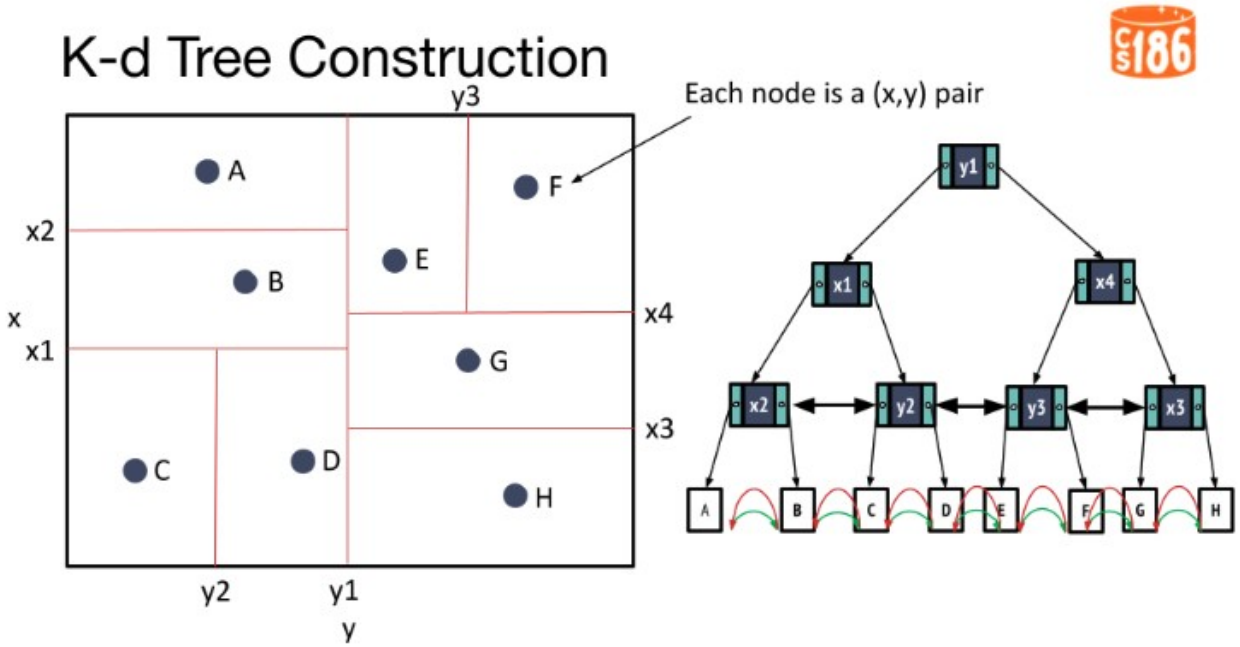




CUBRID

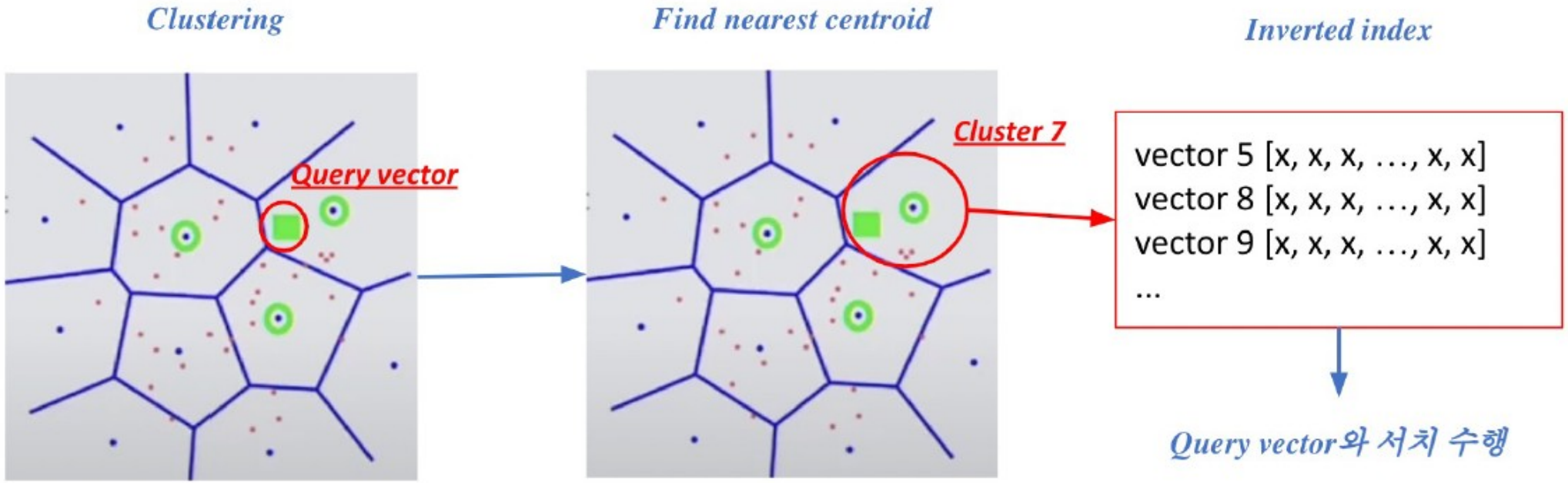
Trees

- splitting strategy?
- k-d TREE, ANNOY (spotify), FLANN
- **SKIP**



Inverted File

- 벡터 공간을 n 개의 클러스터 공간으로 나눔 (k-means)
- query vector 가 속하는 해당 클러스터의 centroid id 로부터 거리를 구한 후, 해당 공간에 속한 벡터에 대해서 탐색 수행

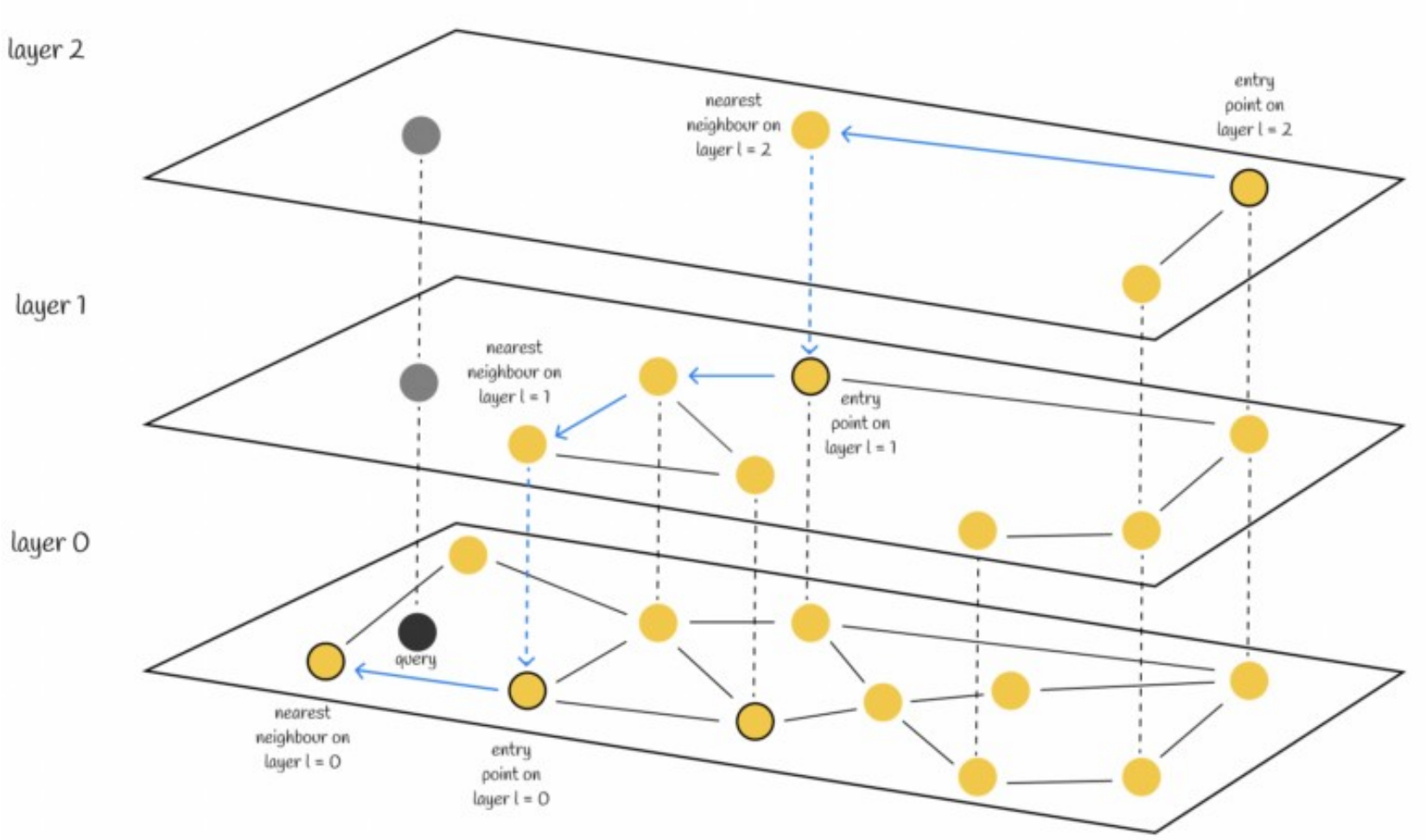


CUBRID

HNSW (Microsoft, 2019)

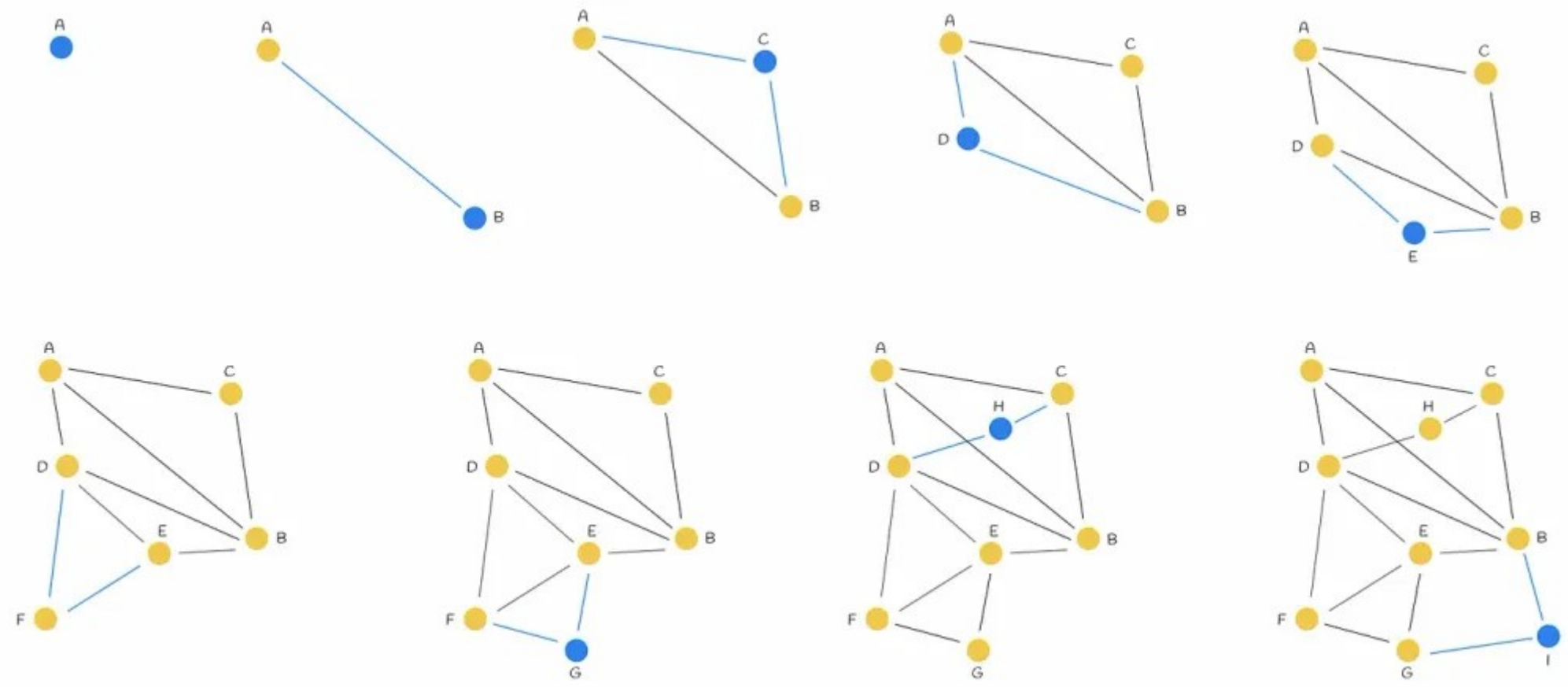
CUBRID

HNSW (Microsoft, 2019)



CUBRID

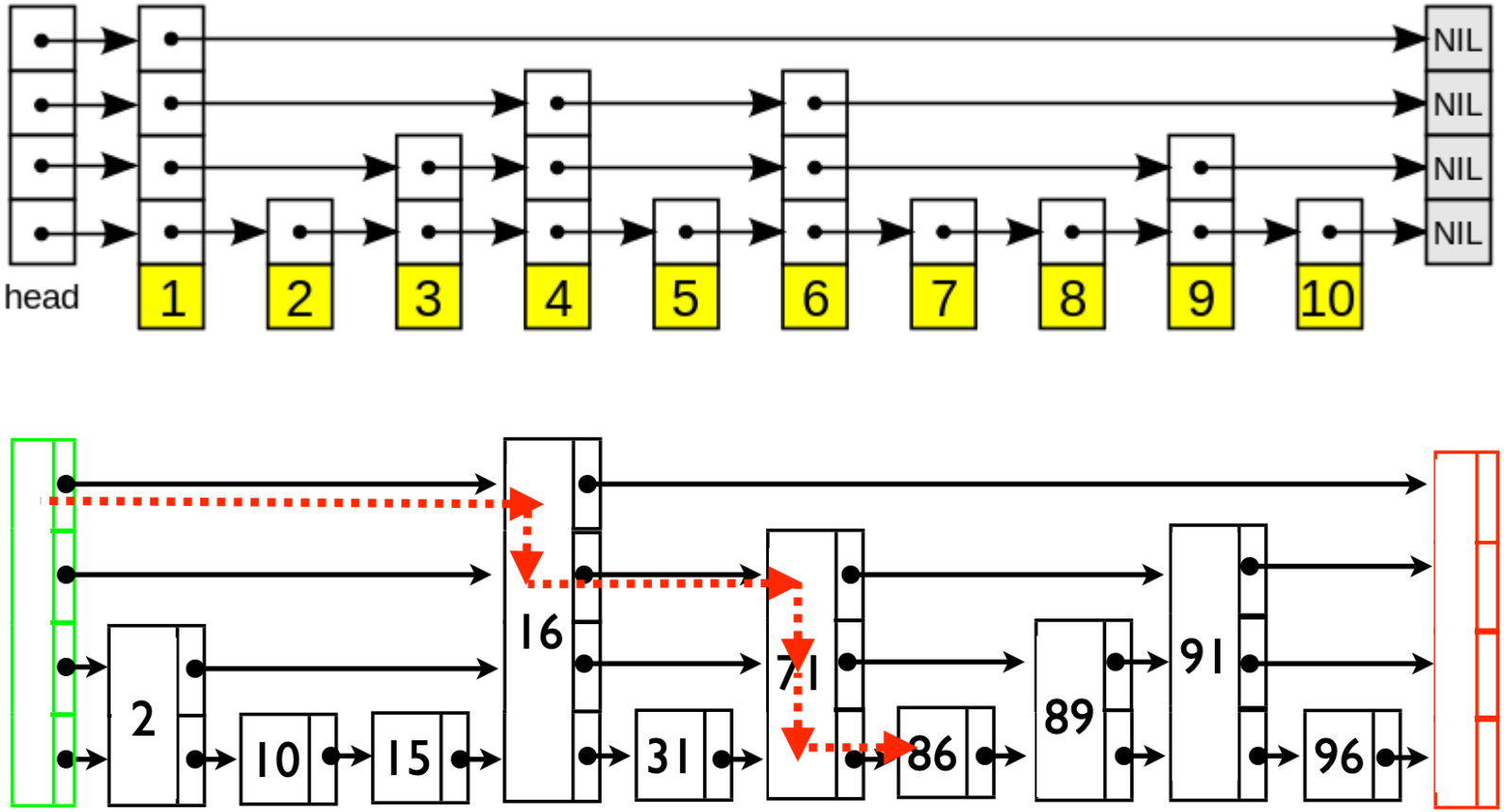
HNSW (Microsoft, 2019)



새로운 벡터가 입력할때마다 M 개의 가까운 벡터와 연결

CUBRID

HNSW (Microsoft, 2019)

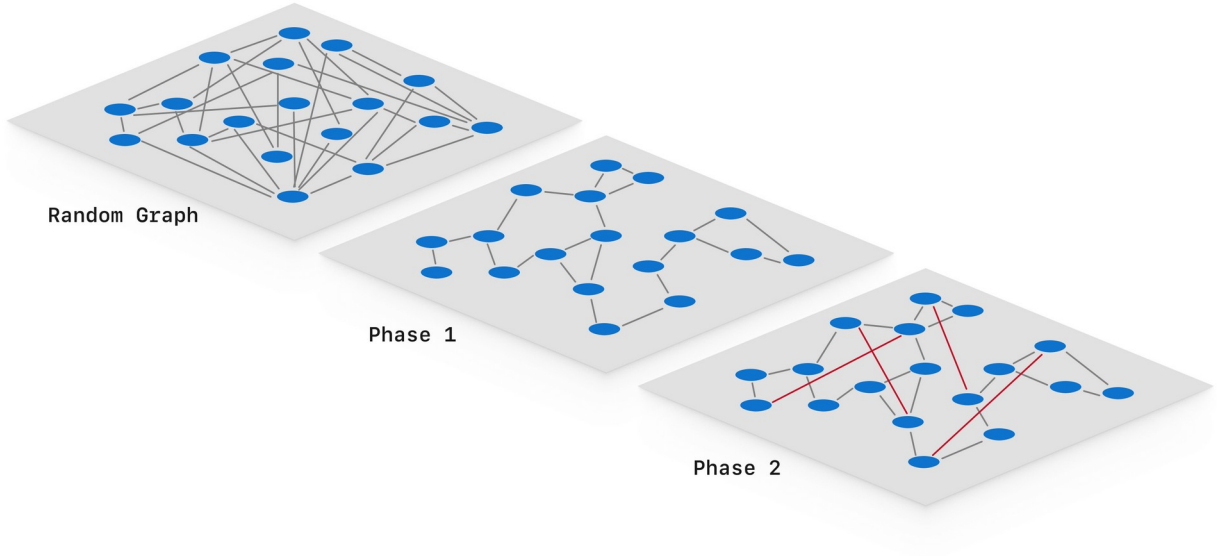


CUBRID

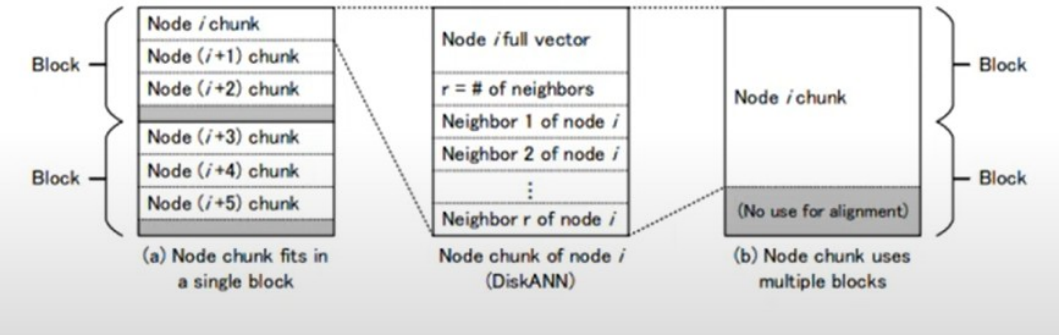
DiskANN (Microsoft, 2019)

- 인메모리 인덱스의 한계 (HNSW...)
- SSD.

Index construction



Index Layout in disk



CUBRID

DiskANN (Microsoft, 2019)

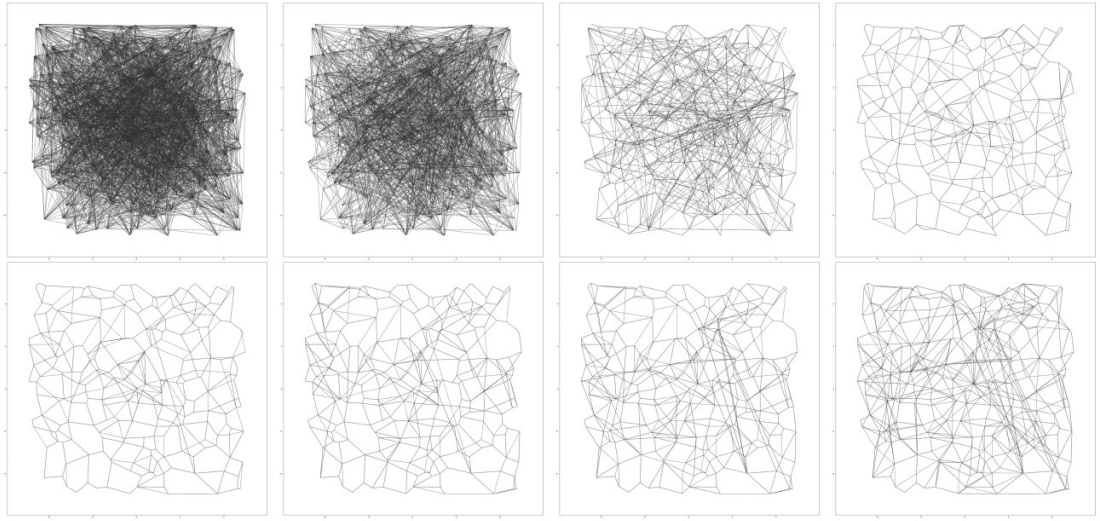
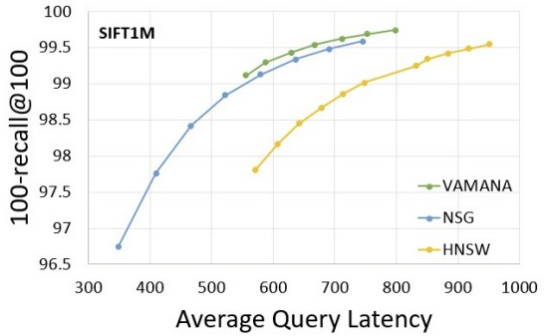
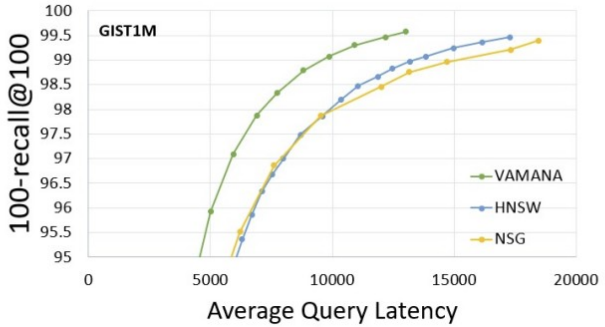
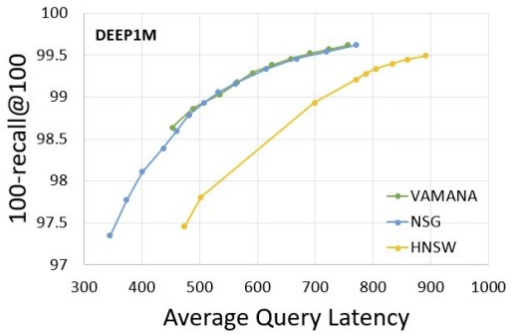


Figure 1: Progression of the graph generated by the Vamana indexing algorithm described in Algorithm 3 on a database with 200 points in 2 dimensions. Notice that the algorithm goes through the first pass with $\alpha = 1$, followed by the second pass where it introduces long range edges.







Query Optimization and Execution

```
SELECT User, Feature
FROM items WHERE User =
'Jack'
ORDER BY embedding <->
'[3,1,2]'
LIMIT 3;
```

- **A given query** -> multiple ways to execute

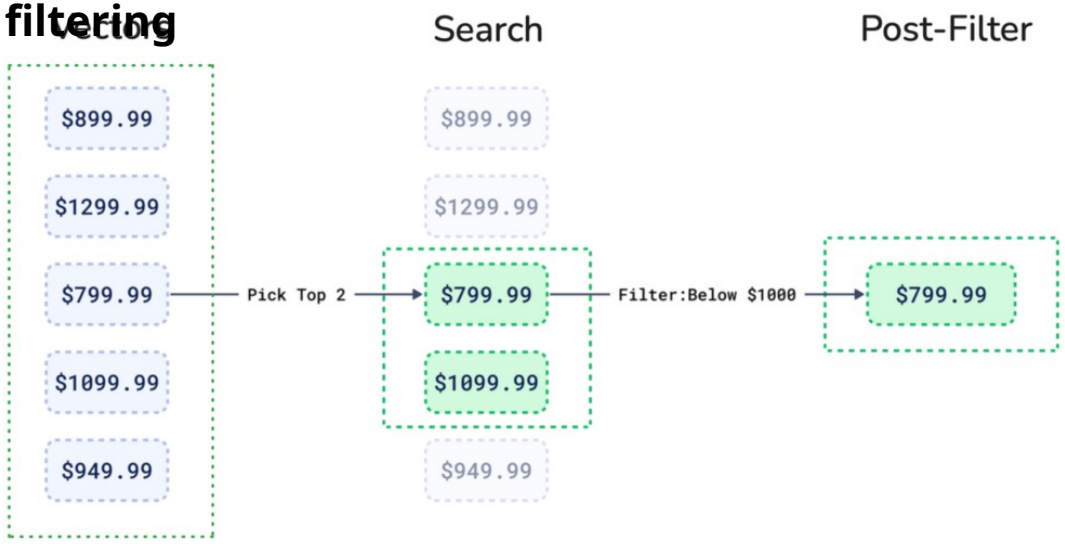
- Query optimizer: select the **optimal** query plan
  - To achieve this **goal**: plan enumeration -> plan selection -> query execution

CUBRID

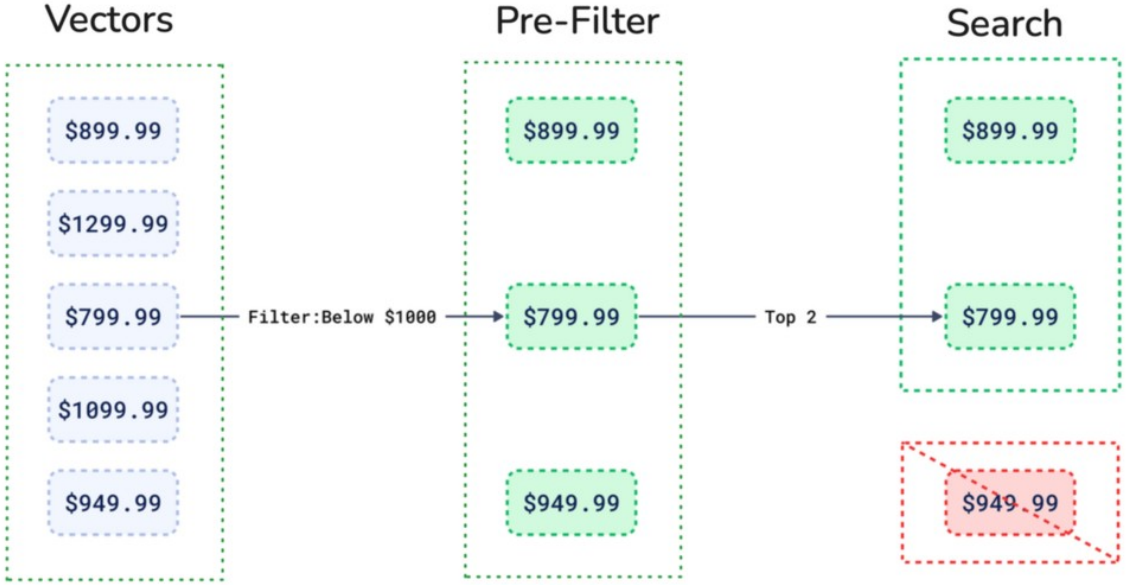
[QO] Hybrid Operators

- Predicated query
  - post-filtering (aka single-stage filtering): **Visit First!**
  - pre-filtering: **Block First!**

Post-filtering



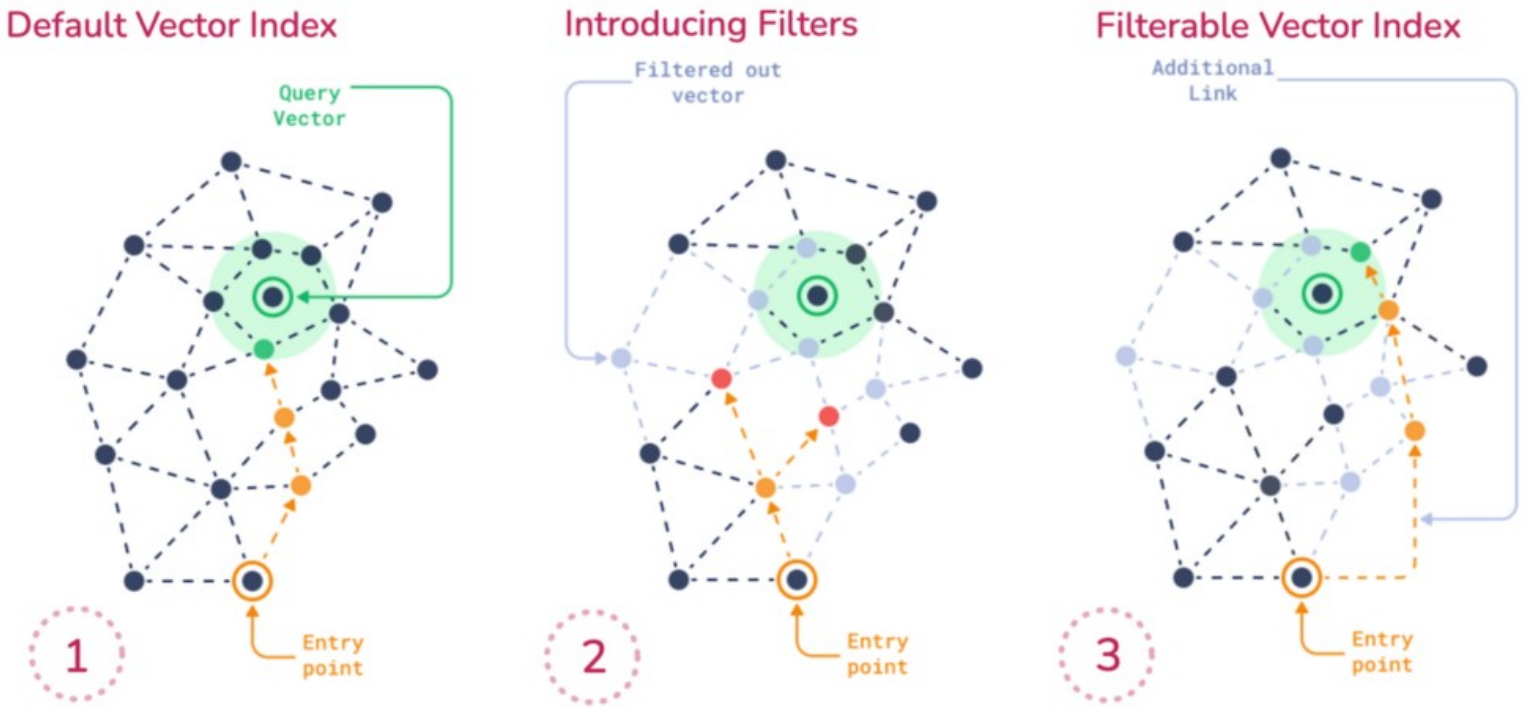
Pre-



CUBRID

[QO] Hybrid Operators: Block-First Scan

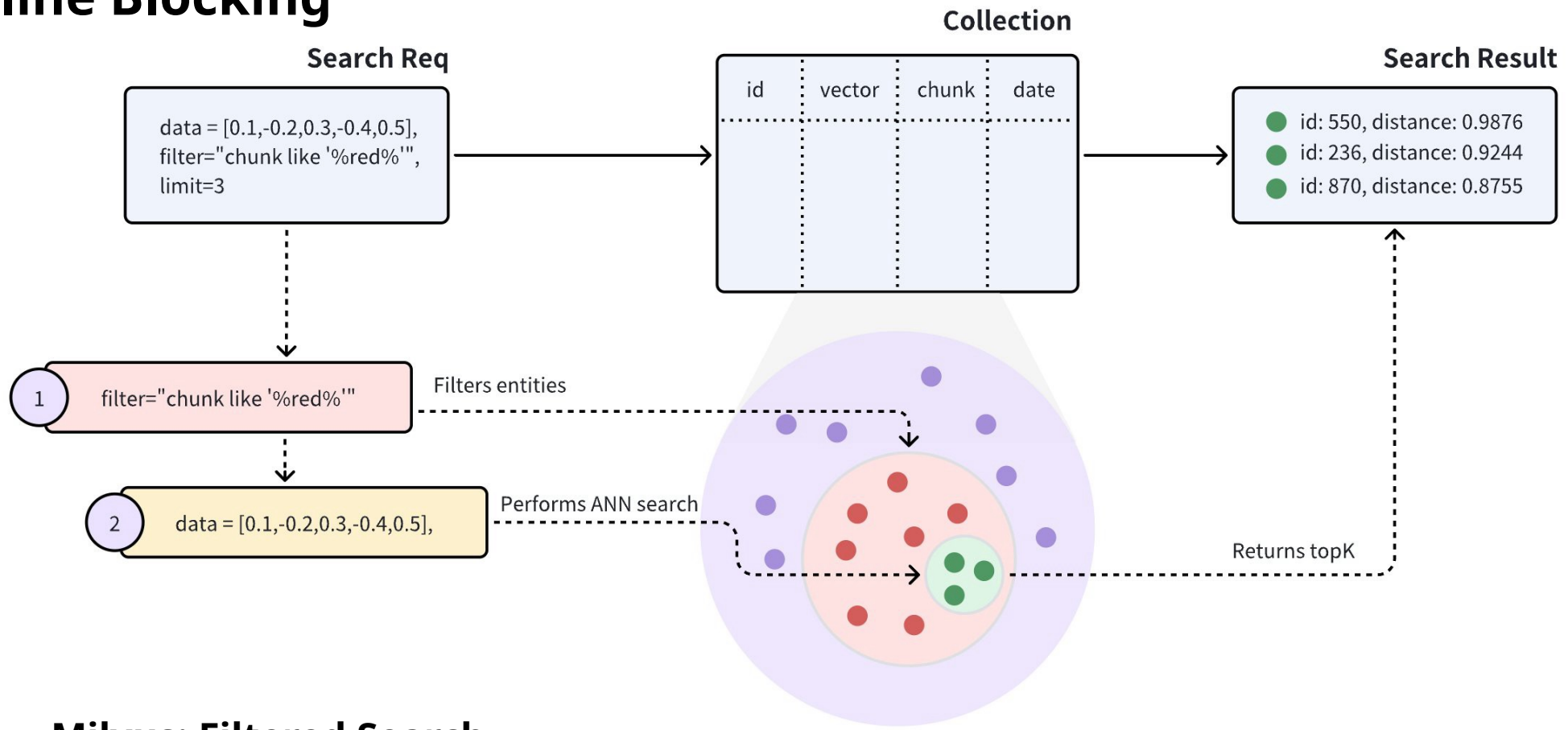
- Online Blocking
  - During index scan, a vector is quickly checked to determine whether it is blocked
- Offline Blocking
  - For graph-based indexes, blocking can cause the graph to become disconnected



CUBRID

[QO] Hybrid Operators: Block-First Scan

Online Blocking



Milvus: Filtered Search

CUBRID

[QO] Plan Enumeration

- Predefined
- Automatic: by optimizer
- **SKIP!!**

CUBRID

[QO] Plan Selection

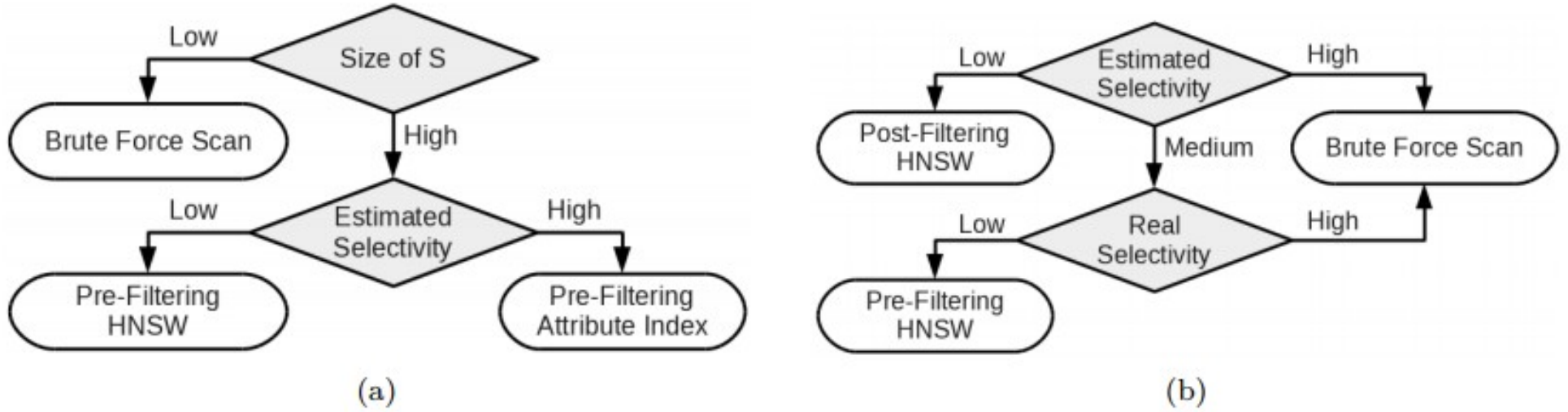


Fig. 10 Plan selection rules in (a) Qdrant and (b) Vespa.

CUBRID

[QE] Query Execution

- Hardware Acceleration
  - CPU Cache
  - SIMD
  - GPUs
- Distributed Search
  - distributed clusters
  - scatter-gather pattern
- Out-of-Place Updates
  - Replicas
  - LSM Tree
  - Bulk update

CUBRID

Conclusion

Part I

챕터 입구 overview

CUBRID

① Introduction — overview

원본 p3. 본 챕터는 "왜 VDBMS 가 별도 분야로 부상했는가" 를 다섯 가지 이유로 정리한다.

1. Vector queries 는 **similarity** 라는 개념에 기반
2. Similarity 연산은 기존 비교 연산보다 **비쌈**
3. 검색 비용 자체가 단순 attribute 보다 큼 (memory · disk)
4. **Indexing 에 쓸 자명한 속성** 이 없음 (정렬 가능·서수 부재)
5. **Hybrid query** — attribute + vector 동시 접근

> Take-away: VDBMS 가 일반 DBMS 위에 layer 가 아니라 **새 카테고리** 로 정의되는 이유는 1·4·5 이 함께 작용하기 때문이다.

② Query Processing — overview

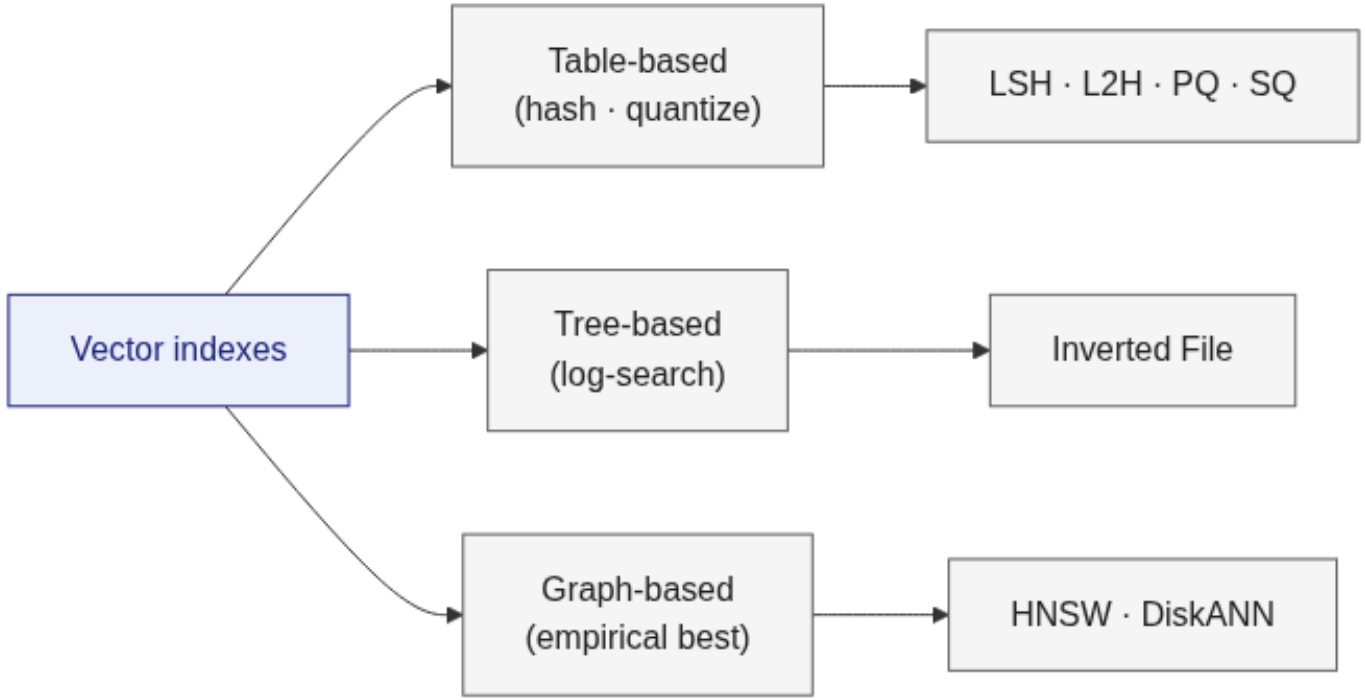
원본 p7–17. 세 개의 subtopic 으로 구성된다.

Subtopic	원본 페이지	핵심
Similarity Scores	p8–10	Basic (Hamming · Inner · Cosine · Minkowski · Mahalanobis) · Learned (metric learning)
Queries and Operators	p11–16	Data manipulation · Basic search · Variants · Batched
Query Interfaces	p17	SDK · SQL-like · 외부 노출 형태

- 발표 시 "Similarity score 선택은 검색 품질의 윗천장을 정한다" 를 강조.
- p12–13 의 embedding pipeline 그림은 contrast 가 약하니, 발표 중 별도 설명 한 줄 ("Data Source → Embedding Model → Embeddings") 을 보충.

③ Indexing — overview

원본 p18–35 (제목은 "Storage and Indexing"). 가장 큰 챕터.
표 = **갱신 용이**, 트리 = **log search**, 그래프 = **경험적 최고**.



④ Query Optimization and Execution — overview

원본 p36–42. 다섯 개의 subtopic.

Subtopic	원본 페이지	무엇
Hybrid Operators	p37	attribute + vector 동시 처리
Hybrid: Block-First	p38–39	attribute 로 selectivity 우선 차단
Plan Enumeration	p40	predefined · optimizer-driven
Plan Selection	p41	cost model 의 선택 기준
Query Execution	p42	execution engine 측면

> CUBRID 관점: hybrid 의 **Block-First** 가 가장 가깝다 — vector 인덱스 호출 비용이 크니 attribute 술어로 선차단하는 전략.

⑤ Current Systems — overview (재배치)

원본은 이 챕터를 **p6 단 한 장** 으로 압축해 둠. 본 deck 에서 분류 트리를 복원.

분류	정의	대표 시스템
Native — Mostly-Vector	attribute 지원 제한	Vald · EuclidesDB · Pinecone · Chroma
Native — Mostly-Mixed	attribute + vector, query plan 다양	Milvus · Qdrant · Manu · Weaviate
Extended — NoSQL	기존 NoSQL 에 vector 확장	Cassandra · Elasticsearch · Redis
Extended — Relational	기존 RDBMS 에 vector 확장	pgvector · SingleStore
Libraries / Other	DB 가 아닌 search SDK / 엔진	FAISS · ScaNN · SPTAG

> CUBRID 11.6 의 `vector(N)` + HNSW 는 **Extended — Relational** 위치.

Part II

Benchmarks

측정 대상 — 7 개의 축

VDBMS 벤치마크는 **recall × latency** 두 차원만으로는 부족하다. paper 는 실무에서 함께 봐야 할 7 가지 축을 정리한다.

축	단위	왜 보는가
Recall@k	0–1	검색 품질 (ground-truth 대비)
Query latency	ms (p50 · p95 · p99)	사용자 체감
Throughput	QPS	서버 처리량
Build time	s · min	인덱스 갱신 비용
Index size	GB	RAM · disk 한도
Insert / update rate	ops/s	streaming 지속성
Hardware	CPU · GPU · NVMe	비교 공정성의 전제

- 한 축만 좋게 만드는 것은 쉽다. **Pareto 전선** 을 봐야 의미가 있다.

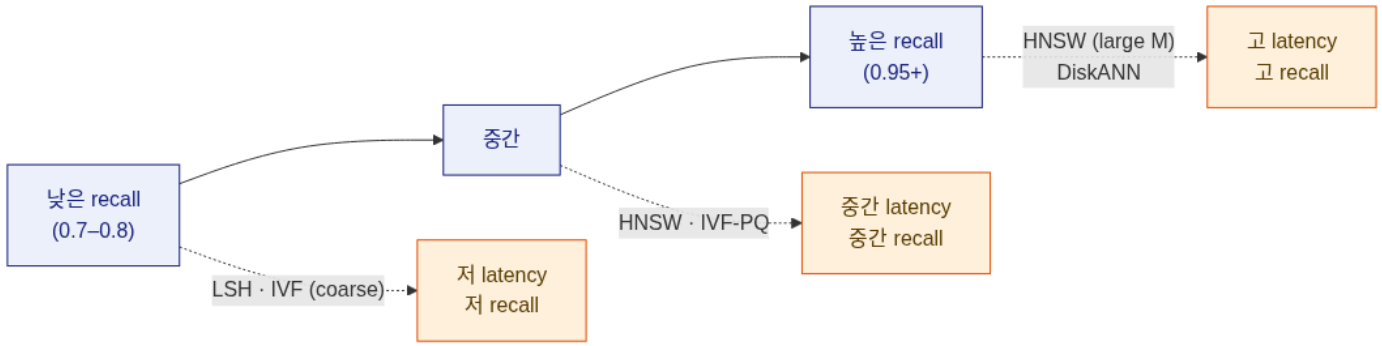
대표 벤치마크 셋

이름	데이터셋	규모	특징
ANN-Benchmarks	SIFT · GloVe · NYTimes · GIST · Deep1M	~1M	가장 널리 인용되는 standard, recall-QPS 곡선
BigANN Challenge	Microsoft SPACEV · Bing · YFCC	1B–10B	십억 단위, **disk-resident** 가 가능한지
MS MARCO	passage retrieval	8.8M	sparse-dense 혼합, 검색 품질 평가
Deep1B	image features	1B	high-dim (96), 그래프 인덱스 비교

- **No single index dominates.** 데이터 분포·차원·메모리 예산에 따라 우열이 뒤집힌다.
- 따라서 paper 의 결론은 "**어느 인덱스가 최고?**" 가 아니라 "**어떤 축에서 최적인지 보고 골라라**".

CUBRID

recall–latency 전선 읽는 법



- 같은 인덱스도 **파라미터 (M · efConstruction · efSearch)** 로 곡선의 어디에 앉을지 정한다.
- 발표 시 "**우리 워크로드의 SLA 가 어느 영역인지 먼저 정해야 한다**" 로 닫는다.

Part III

Challenges and Open Problems

CUBRID

① Freshness · streaming updates

VDBMS 가 학술 prototype 에서 production 으로 옮겨갈 때 가장 먼저 부딪히는 벽.

- **삽입·삭제·갱신이 끊임없이 들어오는 상황에서 recall 을 유지** 할 수 있는가.
- 정적 인덱스 (FAISS · HNSW raw) 는 batch rebuild 가 필요 → 가용성 손실.
- 해법 후보:
  - **Out-of-place + merge** (DiskANN · FreshDiskANN): 새 버전을 별도 인덱스로, 후에 머지.
  - **Tombstone + vacuum**: 삭제는 표시만, 백그라운드 회수.
  - **LSM-style segments**: 작은 메모리 인덱스 + 큰 디스크 인덱스 계층.

> CUBRID 의 vacuum · MVCC 모델과 자연스럽게 맞물리는 지점.

② Multi-modal · heterogeneous data

단일 vector field 만 다루는 시스템은 production 요구의 일부만 만족.

시나리오	요구
Text + Image 검색	다른 embedding 모델의 vector 를 **같은 인덱스에서** 검색
Filter + similarity	`WHERE category='A' ORDER BY embedding <-> q`
Multi-vector per row	한 row 가 title · body · image embedding 을 동시에 가짐
Sparse + dense	BM25 + 벡터 score 의 결합

- 인덱스 입장에서 어려움: **selectivity 가 미리 보이지 않는다** — attribute filter 가 1% 인지 99% 인지에 따라 최적 전략이 뒤집힘.
- paper 는 이를 **hybrid query** 카테고리로 묶어 별도 연구 영역으로 본다.

③ Cost · resource trade-offs

VDBMS 의 운영비는 **인덱스가 메모리에 들어가느냐** 가 결정한다.

결정	비용 항목	트레이드오프
In-memory HNSW	RAM 크기 (vector 수 × 차원 × 4B + 그래프)	가장 빠르지만 RAM 비싸짐
Disk HNSW / DiskANN	NVMe IOPS · 캐시 hit율	십억 단위 가능, latency 증가
Quantization (PQ / SQ)	인덱스 크기 4–16× 축소	recall 손실 (재정렬로 복구)
GPU index	GPU 메모리 · 호스트 ↔ 디바이스 전송	batch query 에 강함, latency 분산

- "어디서 비용을 받을지" 를 미리 정해야 인덱스 선택이 가능.
- AiSAQ (DRAM-free) 같은 최근 연구는 **RAM 을 거의 안 쓰는** 방향으로 Pareto 전선을 밀고 있음.

④ SQL 통합 · security · privacy

paper 가 마지막으로 짚는 세 갈래의 미해결 영역.

- **SQL 통합** — `ORDER BY embedding <-> q LIMIT k` 같은 자연스러운 표현, optimizer 가 인덱스 호출을 일반 plan 안에서 다루기. pgvector · CUBRID 의 방향.
- **Security · 암호화 검색** — 인덱스 자체가 vector 분포를 노출. 암호화 ANN (HE · MPC) 은 연구 단계, 수십~수백× 오버헤드.
- **Privacy · 멤버십 추론** — embedding 으로부터 원본 복원 (inversion) 공격이 가능. 모델·데이터 거버넌스 문제로 확장.

> CUBRID 11.6 의 우선순위는 **SQL 통합** — security · privacy 는 후속 마일스톤. spec **v0.0.25** 의 user-facing contract 가 첫 단계이며, OOS (CBRD-26357) · `information_schema` (CBRD-25859) 같은 사내 engine 프로젝트와의 timing 이 P0 로 추적되고 있다.

Thank you

Q & A

- 원본 deck: `2025-05-suvery-vdbms` — paper 본문 정리
- 본 deck: 그 위의 chapter overview · Benchmarks · Challenges 보강
- 원논문: Pan · Wang · Li, **Survey of Vector Database Management Systems**, VLDB Journal 2024 / arXiv:2310.14021