Vector Database Management Techniques and Systems

James Jie Pan\textsuperscript{1} \quad Jianguo Wang\textsuperscript{2} \quad Guoliang Li\textsuperscript{1}

\textsuperscript{1}Tsinghua University
\textsuperscript{2}Purdue University
Modern DBMSs are designed for data that humans can understand

- Business Operations
- Medical Records
- Financial Timeseries
- Geospatial Vectors/Rasters
- Social Networks

Relational/NewSQL
- PolarDB
- Oracle
- MySQL
- Microsoft SQL Server
- VoltDB

NoSQL/Specialty
- Amazon DynamoDB
- Cassandra
- Apache HBase
- MongoDB
Embeddings: Building Blocks of the Future

More and more applications rely on deep-learning embedding vectors that can only be understood by machines.
Store embeddings in a database and retrieve desired embeddings for whatever downstream task

- Capabilities: similarity-based top-k/range retrieval, hybrid attribute-vector retrieval, multi-modal (multi-vector) retrieval

- Characteristics: read/write latency/throughput, retrieval accuracy, scalability, availability, consistency, fault tolerance, privacy & security, elasticity
Example: LLMs + VDBMS

- Lack of domain-specific knowledge
- Data freshness
- Monetary/time cost

Chatbot Service:
- LLM
- RAG Retriever
- Semantic Cache

VDBMS:
- Domain-Specific Knowledge
- Data Streams
- Past Queries/Responses

Training Corpus
Some of Today’s Commercial Applications

LLM Retrieval-Augmented Generation (RAG)

E-Commerce & Recommendation Systems

Writing Assistant

News Classification

Photo/Video Search & Deduplication

Threat Detection
Why is Building a VDBMS Hard?

Embeddings are...

- Huge (1024 x float64) → costly to move, clog storage
- Hard to retrieve without ambiguity
- Hard to index
- Costly to compare
- Hard to index together with attributes

Figure: Will Koehrsen
Part 1: VDBMS Techniques

Query Processor
- Data Manipulation
- Similarity Search
- Predicated Search
- Multi-Vector Search

Indexing Module
- Table-Based
- Tree-Based
- Graph-Based
- Multi-Index

Query Optimizer / Executor
- Pre-Filter
- Post-Filter
  - Single-Stage
  - Cache / SIMD / GPU

Storage Manager
- WAL
- Write Buffers
  - Index/ Data
  - Index/ Data
  - Index/ Data
- Physical Storage (SSD / S3 / HDFS)
Overview of Query Processing

Query Processor
- Data Manipulation
- Similarity Search
- Predicated Search
- Multi-Vector Search

Query Definition
- Similarity Score
  - Metrical Scores
  - Non-Metrical Scores
- Query Type
  - Data Manipulation
  - Range Search
  - (c,k)-Search
  - Variants
- Query Interface
  - API, SQL

Operators & Algorithms
- Vector Operators
  - In/Up/Del
  - Object Embedding
  - Vector Math
  - Vector Projection
- Search Operators
  - Table Scan, Top-K
  - Index-Based Operators
- Search Algorithms
  - Brute-Force Search
  - Index-Based Search

Query Processor
- High dimensionality
- Large data volume
- Low latency
- High accuracy
Query Definition: Similarity Scores

- **Non-Metrical Scores**
  - Inner Product, Cosine Similarity

- **Metrical Scores**
  - Metric axioms (id., positiv., sym., triangle ineq.)
  - Hamming, Minkowski (Manhattan, Euclidean)

- A function \( f : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R} \) indicating degree of similarity

- Similarity calculations are expensive
  - Sq. Euclidean (D=1024 floats) takes 62us on my machine (Intel i5 @ 2.3 GHz), about the same as SSD random seek
Query Types: Data Manipulation

• VDBMS interacts with embedding model via plugin/add-on/extension
• More user friendly

• User is responsible for producing embeddings
• More controllable (e.g. custom embedding model)
Query Types: Vector Search Queries

Query Variants: predicated, batched, multi-vector
Query Interfaces

**API-Based**

- Chroma API
  - count, add, get, peek, query, modify, update, upsert, delete

  - ✔️ Less impedance
  - ✗ Not portable

---

**SQL-Based**

- pgvector

  - CREATE TABLE items (... embedding vector(128));
  - UPDATE items SET embedding = ' [1,2,3]' WHERE id = 1;
  - DELETE FROM items WHERE id = 1;
  - SELECT * FROM items WHERE cat_id = 123 ORDER BY embedding <-> ' [3,1,2]' LIMIT 5;
  - CREATE INDEX ON items USING hnsw...;

  - ✔️ Potentially more portable
  - ✗ Impedance mismatch
**Operators & Algorithms**

**Project** a vector onto similarity score with respect to query vector
• $O(D)$ for $D$-dimensions

**Bruteforce Search**
- Scan Direction
- Vector Dataset
- Project
- Project
- Project
- Results Buffer

**Index-Based Search**
- Index Scan
- Vector Index

**Distrib./Parallel Search**
- Scatter
- Gather
- Shard 1
- ... Shard n

**Hardware-Accelerated Search**
- CPU / SIMD
- Main Memory
- GPU
- GPU Memory
Characteristics of Search Algorithms

**Performance**

- Amount of visited vectors, similarity comparisons

**Accuracy**

- Recall: \( \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \)
- \textbf{Recall@K}
  - Recall when \( k=K \) for k-NN, ANN (Li et al. 2020)
  - Proportion of queries where 1-NN is ranked in first \( k \) results (Jegou et al. 2011)
  - Proportion of true nearest-neighbors within the first \( K \) results of a k-NN or ANN query (\( K \leq k \)) (RecSys)
- Precision: \( \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \)
Challenges to Query Processing

**Query Semantics**
(e.g. “Amazon”)

**Curse of Dimensionality**

**Score Selection**

**Score Design**
(e.g. Metric Learning)

---

**Figure: Moll et al 2024**

**Figure: Cornell CS4780/CS5780 Lecture 2 (Fall 2018)**
Overview of Storage & Indexing

Indexing Module
- Table-Based
- Tree-Based
- Graph-Based
- Multi-Index

Construction / Search / Maint.

Construction
- Randomization
- Learned Partitions
- Navigable Partitions

Search
- Bucket Scan
- Defeatist Search
- Best-First Search

Maintenance
- Rebalancing

Logical / Physical Storage

Logical Structure
- Tables
- Trees
- Graphs

Physical Structure
- Quantization
- Disk-Resident Indexes

- High dimensionality
- Large data volume
- Low latency
- High accuracy
- **Construction cost**
- **Storage cost**
- **Maintenance cost**
Table-Based Indexes

**Locality-Sensitive Hashing**
- Random Hyperplanes (E2LSH)
- Random Bits (Faiss IndexLSH)
- Random Balls (FALCONN)

**Centroid-Based (Quantization)**
- Nearest Centroid (SPANN, Faiss IVF*)
- Nearest Centroid Product (Faiss PQ)

**Learned Hashing**

- Construction: $O(DN)$
- Search: $O(DN^c)$, $0 < c < 1$

**Learn from data distribution**

**Rely on probability amplification**

**Data Drift**

Original data

Real concept drift

$p(y|X)$ changes

Figure: Gama et al, 2013
Locality-Sensitive Hashing (LSH)

Figure: Andoni & Indyk, 2008

Vec. $g = (h_1, h_2)$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>b</td>
<td>21</td>
</tr>
<tr>
<td>c</td>
<td>23</td>
</tr>
<tr>
<td>d</td>
<td>44</td>
</tr>
<tr>
<td>e</td>
<td>42</td>
</tr>
<tr>
<td>f</td>
<td>42</td>
</tr>
</tbody>
</table>

e.g. E2LSH

$L=2$, $K=4$

More Projections
(Increasing $K$)

More Hash Tables
(Increasing $L$)

Collision Probability
of some $g_j$

Distance

Distance

SIGMOD'24 Tutorial

Figure: Andoni & Indyk, 2008
LSH Families

• “Hash Family”: For any $r_1$, $r_2$, $x \in S$, and $q$:
  - if $d(x, q) \leq r_1$, then collision prob. $\geq p_1$  Large is Better
  - if $d(x, q) \geq r_2$, then collision prob. $\leq p_2$  Small is Better
• Typically storage $\sim O(DN^{1+c})$, search $\sim O(DN^c)$ where
  $$\rho = \frac{\log(1/p_1)}{\log(1/p_2)}$$
• Interesting families:
  - Hamming (Faiss IndexLSH) $\rho = 1/c$
  - Random hyperplans (E2LSH) $\rho = 1/c$
  - Spherical LSH (FALCONN) $\rho = 1/(2c^2-1)$
Learning to Hash (L2H)

Pairwise Similarity-Preserving

Multiwise Similarity-Preserving

Implicit Similarity-Preserving

Reconstruction Error-Minimizing


kNN Hashing, Ding et al 2015

Spherical Hashing, Heo et al 2012

Quantization, Wang et al 2018
Quantization

Level Quantization
- Uniform (e.g. Faiss SQ)
- Non-Uniform

Learned Quantization
- k-Means (e.g. Faiss IVFSQ)
- Product (e.g. Faiss PQ)

Quantization Levels:
- 2 bits
- 4 bits
- 6 bits

4-Dimensional Vectors
- Dims [1, 2]: m=2, k=4
- Dims [3, 4]

E.g.: $c_{1,1}, c_{1,2}, c_{1,3}, c_{1,4}, c_{2,1}, c_{2,2}, c_{2,3}, c_{2,4}$

$U = U_1 \times U_2$
Product Quantization

- Preserves dimensions, e.g. $\mathbb{R}^4 = \mathbb{R}^2 \times \mathbb{R}^2$
- Faster training
  - 4 centroids per subspace = 16 total codes
  - $k$-means $O(DN*k) : 2(2N*4) = 16N$ vs $4N*16 = 64N$

Figure: Jegou et al 2011

U = U_1 \times U_2

Diminishing Returns

Square Distortion vs Codeword Bits
“IVFADC” Asymmetric Distance Comp.

Coarse Quantization

Buckets

16 bits

16 bits

2 bits

2 bits

K-MEANS

Index

Search

Vector Centering

Vector Residuals

Sample + PQ

Compressed Entries

ADC Lookup Table

\[
\begin{align*}
\overbrace{d(q_1, c_1^m)}^{U_m} & \cdots d(q_1, c_K^m) \\
\cdots & \cdots \\
\overbrace{d(q_m, c_1^m)} & \cdots d(q_m, c_K^m)
\end{align*}
\]

• Bucketing for fast search
• Lookup cache
• Sampling from residuals for faster training
Summary of Table-Based Indexes

Randomization-Based

- E.g. Faiss IndexLSH, E2LSH, FALCONN
- ✓ Theoretical guarantees
- ✓ No rebalancing
- ✗ High storage costs

Learning-Based

- E.g. L2H, SQ, PQ, IVFADC
- ✓ Low storage costs
- ✓ Low latency
- ✗ Susceptible to data drift
Table-Based Indexes: Discussion

**Advantages**
- Disk-friendly, E.g. LSH, SPANN
- Readily supports in-distribution insert/update/delete
- Easier to derive error bounds

**Disadvantages**
- Hard to deal with queries near borders/corners
  - *How many buckets are adjacent to a corner in a D-dimensional space?*

![Diagram showing number of adjacent buckets in 1D, 2D, and 3D spaces]
Tree-Based Indexes

Axis-Aligned
- k-d Tree
- PKD-tree (Principal Components)
- FLANN

Randomized
- RPTree
- ANNOY

Fixed
- Medians
- Overlapping

Construction: $O(DN \log N)$
Search: $O(D \log N)$

Recursive space partitioning
Rely on probability amplification
Balanced partitioning

Discernability
k-d Tree

- $O(DN^{1-1/D})$ search, $O(DN \log N)$ construction
- Tends toward $O(DN)$ as $D$ grows

Figure: Silpa-Anan and Hartley, 2008
Principal Component Trees

- More discernability by aligning to principal components
- No huge gains from multiple trees

Figure: Silpa-Anan and Hartley, 2008 (bottom)
Random Projection Trees

Random Projections

Random Rotations

Random Splits

Overlapping Splits

ANNOY (200 random trees)

FLANN (Principal Components)

Ball Tree

Queries per Second

Recall

Source: ann-benchmarks.com

Summary of Tree-Based Indexes

Axis-Aligned

- E.g. k-d Tree, PKD-Tree, FLANN
- √ High recall for low dims.
- X Inflexible

Randomized

- E.g. ANNOY, RPTree
- √ High recall for high dims.
- X High storage (forests)
Tree-Based Indexes: Discussion

**Advantages**

- ✓ Disk-friendly in principle (store together by leaf)
- ✓ $O(D \log N)$ defeatist search
- ✓ Supports in-distribution insert/update/delete

**Disadvantages**

- ✗ Hard to keep balanced following data drift
- ✗ Low recall for queries near leaf borders / corners
Graph-Based Indexes

Navigable Partitioning

**k-Nearest Neighbor Graphs (kNNGs)**
- KGraph
- EFANNA

**Monotonic Search Networks (MSNs)**
- FANNG
- NSG
- Vamana

**Small-World Graphs**
- NSW
- HNSW

- Directly index the nearest neighbors
- Support greedy depth-first search
- Exploit navigational small-world properties

- Construction: ranges from $O(DN \log N)$ to $O(DN^{1+c})$
- Search: $O(D \log N)$ or $O(DN^c)$

Increasing Randomness
k-Nearest Neighbor Graphs (kNNGs)

A. Sampled point inside query ball → check its neighbors
B. Point ball intersects query ball → check neighbors in the overlap
C. Point ball outside query ball → prune near neighbors of p

- **O(1) search** for queries in dataset, else $O(DN^{1+c})$ via sample and prune (see above)

**Construction**

- **Exact:** $O(DN^2)$
KGraph (NNDescent)

“A neighbor of a neighbor is likely to also be a neighbor”

Fast convergence

<table>
<thead>
<tr>
<th>Dataset &amp; Measure</th>
<th>Empirical Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corel/l₂</td>
<td>$O(n^{1.11})$</td>
</tr>
<tr>
<td>Audio/l₂</td>
<td>$O(n^{1.14})$</td>
</tr>
<tr>
<td>Shape/l₂</td>
<td>$O(n^{1.11})$</td>
</tr>
<tr>
<td>DBLP/cos</td>
<td>$O(n^{1.11})$</td>
</tr>
<tr>
<td>Flickr/EMD</td>
<td>$O(n^{1.14})$</td>
</tr>
</tbody>
</table>

Construction

• Initialize random kNNG
• For each node, change any 2-hop neighbor into a 1-hop neighbor if it is a new k-nearest neighbor
• Repeat until convergence

Figure: Dong et al, 2011
Monotonic Search Networks (MSNs)

“Greedy search is all you need”

Figures: Delaunay triangulation (Wikipedia)

- Probe the graph by conducting searches from a random entry point to a random query point in the dataset, e.g. FANNG
- Designate a point as the sole entry point for all search trials, e.g. NSG, Vamana
Fast ANN Graph (FANNG)

Construction

• $O(N)$ trials, each trial $O(N^c)$ to yield $O(N^{1+c})$
• Occlusion rule prunes redundant edges to limit out-degrees

Search Latency after 50N Random Search Trials

$O(N^c), 0 < c < 1$

Dataset Size (Log Scale)

Distance Calculations (Log Scale)

Figure: Harwood & Drummond, 2016
Navigating Spreading-Out Graph (NSG)

- Single source makes it easier to establish monotonic search paths from this node to all other nodes.
- Spanning tree ensures connectivity.

**Construction**

- $O(N^{1+c} \log N^c)$

**Search**

- $\sim O(\log n)$ due to higher quality neighborhoods.

Figure: Fu et al, 2019
Vamana/DiskANN

VamanaIndexScan

- Similar to NSG
- On-disk neighborhoods
- Edge traversal performed in memory using compressed vectors

SSD Storage

<table>
<thead>
<tr>
<th></th>
<th>R Neighbor IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-Precision Vector 1</td>
<td>1 ... R</td>
</tr>
<tr>
<td>Full-Precision Vector 2</td>
<td>1 ... R</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

SIGMOD'24 Tutorial
Small-World Graphs

- Small characteristic path lengths (short shortest-paths)
- High clustering (friend of a friend is also my friend)

Figure: Watts and Strogatz, Nature 1998
Navigable Small-World Graphs

Random edge probability $\alpha$

Randomly add an edge from $u$ to $v$ with probability $|u,v|^{-\alpha}$

$O(N^\beta)$ Greedy Search Complexity Parameter $\beta$

- Not all small-world graphs permit $O(\log N)$ greedy search
- (In Kleinberg’s graph, only $\alpha = 2$ yields a navigable graph)
Hierarchical Navigable Small-World Graphs (HNSW)

- Simply **inserting vectors one at a time**, connecting it to its k nearest neighbors already in the graph found via search trial, is **Navigable and small-world**
- Hierarchical levels **mitigates high out-degrees**

Figures: Malkov et al 2014, Malkov & Yashunin 2020
Summary of Graph-Based Indexes

**Nearest-Neighbor Graphs**

- O(1) offline search
- Fast approx. construction
- Slow for online queries

E.g. KGraph (NNDescent)

**Monotonic Search Networks / Small Worlds**

- ~O(log N) online search
- Slow construction

E.g. FANNG, NSG, Vamana, HNSW
Graph-Based Indexes: Discussion

**Advantages**

√ Empirically state-of-art throughput/recall

**Disadvantages**

✗ Hard to adapt to disk

✗ Hard to support updates for many graphs
  • For HNSW: accuracy degradation issue

✗ Long construction times for graphs based on search trials (incl. HNSW)
## Summary of Indexes

<table>
<thead>
<tr>
<th>Index Type</th>
<th>Search Efficiency</th>
<th>Search Accuracy</th>
<th>Write Friendliness</th>
<th>Disk Friendliness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table-Based</td>
<td>![Up Arrow]</td>
<td>![Up Arrow]</td>
<td>![Up Arrow]</td>
<td>![Up Arrow]</td>
</tr>
<tr>
<td>Tree-Based</td>
<td>![Up Arrow]</td>
<td>![Up Arrow]</td>
<td>![Up Arrow]</td>
<td>![Up Arrow]</td>
</tr>
<tr>
<td>Graph-Based</td>
<td>![Up Arrow]</td>
<td>![Up Arrow]</td>
<td>![Up Arrow]</td>
<td>![Down Arrow]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index Type</th>
<th>Construction Efficiency</th>
<th>Storage Efficiency</th>
<th>Ease of Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table-Based</td>
<td></td>
<td></td>
<td>![Up Arrow]</td>
</tr>
<tr>
<td>Tree-Based</td>
<td></td>
<td></td>
<td>![Up Arrow]</td>
</tr>
<tr>
<td>Graph-Based</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Challenges to Storage & Indexing

Index Variety

- Index Variety
- Capabilities

- Index Selection
- Index Design

E.g. How to handle workload shift?

Examples:
- Disk-Resident Indexes
- Concurrent Indexes
- Distributed Indexes
- Indexes for Predicated Queries
- Etc.
Overview of Query Optimization

Predicated Vector Search Query
e.g. select * from items where price < $10 order by simTo(query) limit k

Plan Types
- Naive
- Pre-Filtering
- Post-Filtering
- Single-Stage Filtering

Cost-Based
- Cost Model
- Operator Costs

Rule-Based
- Rule Design
Plan Types for Predicated Queries

**Brute-Force**

- **Top-K**
- **Project**
- **Table Scan**

**Single-Stage**

- **Vector Index Filtered Scan**
- **“Whitelist”**
- **Query**
- **AND/OR**

**Pre-Filtering**

- **AND/OR**
- **Attr.Index Scan**
- **Clause 1**
- **...**
- **Clause n**

**Post-Filtering**

- **Intersect**
- **AND/OR**
- **Vector Index Scan**
- **Attr.Index Scan**
- **Clause 1**
- **...**
- **Clause n**
## Summary of Plan Types

<table>
<thead>
<tr>
<th>Plan Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute-Force</td>
<td>✓ Exact (100% Recall)</td>
<td>✗ High latency for weak filters</td>
</tr>
<tr>
<td>Pre-Filtering</td>
<td>✓ Exact (100% Recall), efficient for strong filters</td>
<td>✗ High latency for weak filters</td>
</tr>
<tr>
<td>Post-Filtering</td>
<td>✓ Efficient (Native vector search speed &amp; native attribute filter speed)</td>
<td>✗ Low accuracy risk (e.g. empty intersection). Mitigation: collect $(\alpha k)$ similar vectors, not just k. E.g. ADBV</td>
</tr>
<tr>
<td>Single-Stage Filtering</td>
<td>✓ No loss of recall, often more efficient than pre-filtering</td>
<td>✗ Possibly high latency for strong filters. Mitigation for graph-based indexes: Encourage visiting satisfying vectors, e.g. FilteredDiskANN, HQANN, NHQ; Increase reachability, e.g. ACORN; Decrease failures via partitioning, e.g. Milvus</td>
</tr>
</tbody>
</table>
Plan Enumeration

**Predefined, e.g. Weaviate, Milvus, ADBV**

- Use a **single predefined plan** for all predicated queries, e.g. Weaviate, Pinecone
- Predefine **multiple plans** and select which plan to use at query time, e.g. ADBV, Milvus

**Automatic, e.g. PASE, pgvector (PostgreSQL)**

- Let the optimizer automatically enumerate plans
  - **Post-filter low-accuracy risk is real!**

---

pashkinelfe commented on Sep 21, 2023 • edited

I suppose the case when post-filtering depletes all (or most) of ann-found tuples is completely legit. Though considering how often are the related complaints I'd suggest it to be mentioned in readme/manual explicitly. Maybe both filtering by other attribute or "filtering" due to dead heap tuples could be mentioned both.
Plan Selection

**Rule-Based, e.g. Qdrant, Vespa**
- Simple to implement
- Depends on accurate selectivity estimates

**Cost-Based, e.g. ADBV, Milvus**
- Select based on a cost model
- Generally more flexible
- Depends on accurate operator cost estimates
Rule-Based Plan Selection

Example: Qdrant and Vespa
Cost-Based Plan Selection in ADBV

Table 1: Notations used by hybrid query optimization

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>the total number of tuples in database</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>the ratio between $n$ and the number of records satisfying structured predicate</td>
</tr>
<tr>
<td>$\beta$</td>
<td>the visited subcells ratio during VGPQ index searching process in VGPQ Knn Bitmap Scan</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>the visited subcells ratio during VGPQ index searching process in VGPQ Knn Scan</td>
</tr>
<tr>
<td>$\sigma_{{B,C,D}}$</td>
<td>amplification factors of ANNS Scan operators in Plan{$B, C, D$}</td>
</tr>
<tr>
<td>$c_1$</td>
<td>the total time cost to fetch a vector and compute pairwise distance</td>
</tr>
<tr>
<td>$c_2$</td>
<td>the total time cost to fetch a PQ code and run ADC</td>
</tr>
</tbody>
</table>

- **Pre-Filter k-NN Scan**
  \[
  cost_A = T_0 + \alpha \times n \times c_1
  \]

- **Single-Stage PQ Filtered Scan**
  \[
  cost_B = T_0 + \alpha \times n \times c_2 + \sigma_B \times k \times c_1
  \]

- **Single-Stage VGPQ Filtered Scan**
  \[
  cost_C = T_0 + \beta \times n \times \alpha \times c_2 + \sigma_C \times k \times c_1
  \]

- **Post-Filter VGPQ Scan**
  \[
  cost_D = \gamma \times n \times c_2 + \sigma_D \times k \times c_1
  \]
Challenges to Query Optimization

**Cost Estimation**

**Single-Stage**
- May be difficult to estimate cost of Vector Index Filtered Scan (i.e. backtracking)

**Post-Filtering**
- Hard to take into account low-accuracy risk

I suppose the case when post-filtering depletes all (or most) of ann-found tuples is completely legit. Though considering how often are the related complaints I'd suggest it to be mentioned in readme/manual explicitly.

Source: pgvector Issue #263
Overview of Query Execution

**Hardware Acceleration**
- CPU Cache
- SIMD/GPU

**Data Manipulation**
- Execution Mode
  - Immediate
  - Deferred

**Distributed Query Processing**
- Partitioning
  - Random/Uniform
  - Attribute-Based
  - Learned
- Consistency
  - Strong
  - Eventual

**Data Transfers**
- HW Parallelism
  - CPU Cache
  - SIMD/GPU
Hardware Acceleration

Data Transfers

• CPU Cache: “Query blocks” keep query vectors cache-resident while assigning threads to data vectors keeps data vectors cache-resident, e.g. Milvus

Data/Task Parallelism

• SIMD/GPU for IVFADC, e.g. Faiss
  • Parallelize lookups by keeping the lookup table inside the SIMD register and simulate lookups via SIMD shuffle (also avoids memory retrieval)
  • Parallelize summations over registers
Data Manipulation

Streaming Updates

• Some indexes support in/up/del, e.g. HNSW
• Vearch: use tombstone deletes to avoid disconnecting the graph + periodic garbage collection

Batched Updates

• Perform in/up/del inside a fast-writeable slow-readable structure which also participates in search
• Reconcile into the slow-writeable fast-readable structure at a convenient time
Fast Slices with Slow Segments

Manu Vector Database

Growing Slice (not indexed)

Full Slices

Fast TTI Index (e.g. IVFFlat)

Fast TTI Index (e.g. IVFFlat)

"Growing" Segment

"Sealed" Segments

• HNSW is built over the growing segment once full, and then the temporary slices are discarded

SIGMOD'24 Tutorial
Log-Structured Merge (LSM) Tree

Milvus Vector Database

- HNSW built during segment compaction
- Tombstones are reconciled during compaction
• Designed for massive TB+ datasets
• HNSW serves as the fast-writeable structure while disk-resident VGPQ is the slow-writeable structure
Distributed Query Processing

Partitioning

• By attribute if available, e.g. ADBV
• By k-means cluster, e.g. ADBV
• By memory availability, e.g. Vald
• By uniform hashing

Consistency

• Eventual consistency
  • By quorum, e.g. Weaviate
  • By timestamp delta, e.g. Manu
• Strong consistency
  • Concurrent HNSW via internal locks, e.g. Vearch
Scatter-Gather Search

- Hard to know beforehand which workers to select
- k-means partitioning can reduce searched partitions but needs rebalancing
Part 2: Commercial VDBMSs

**Native**
- VEARCH
- Pinecone
- zilliz
- EuclidesDB
- milvus
- marqo
- Chroma
- Weaviate
- nucliaDB
- Vald

**Extended**
- ClickHouse
- AnalyticDB
- SingleStore
- vespa
- Apache Lucene
- Elasticsearch
- OpenSearch
- Solr
- Microsoft
VDBMS Types and Capabilities

**Native Systems**
- Mostly-Vector
  - Limited query/index types, mainly read-heavy, limited or no predicated queries
- Mostly-Mixed
  - Multiple query/index types, mixed workloads, predicated + attribute-only queries

**Extended Systems**
- NoSQL
  - Basic vector search capability, similar to Mostly-Vector systems
- Relational
  - Comprehensive capabilities, similar to Mostly-Mixed systems

**Search Engines & Libraries**
- Search Engines
- Libraries
  - Embedded at application-level
  - Offers specific functionality, e.g. a single vector index
VDBMS Performance

- Native systems “tend to” outperform extended systems

- This view is already being challenged. Zhang et al ICDE’24: “...there is no fundamental limitation in using a relational database (e.g., PostgreSQL) to support efficient vector data management”
Design Considerations

### Database Management

(Distributed QP, Failure Recovery, Storage Management, Query Optimizer)

### Vector Search Capability

(Operators, Indexes, Plans, Interface)

<table>
<thead>
<tr>
<th>System Type</th>
<th>Examples</th>
<th>Features</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Mostly-Vector</td>
<td>Pinecone, Vearch</td>
<td>Distributed QP, Failure Recovery</td>
<td>Shards/reps, shared-storage persistence</td>
</tr>
<tr>
<td>Native Mostly-Mixed</td>
<td>Milvus, Weaviate</td>
<td>Distributed QP, Failure Recovery, Storage Management</td>
<td>Shards/reps, WAL, LSM-Tree</td>
</tr>
<tr>
<td>Extended NoSQL</td>
<td>Redis, Vespa</td>
<td>Single Query/Index Types</td>
<td>Bolt on HNSW</td>
</tr>
<tr>
<td>Extended Relational</td>
<td>PASE, ADBV</td>
<td>Multi. Query/Index Types, Operators</td>
<td>Tight Integration</td>
</tr>
</tbody>
</table>
Native Mostly Vector

**Example:** Pinecone

API serves as query interface
Stream Processor triggers vector indexing
Blob Storage holds persistent replicas for backup/recovery

**Advantages**
√ Low-latency searches, high search throughput

**Limitations**
× Systems with graph-based index may struggle with writes
× Systems with table-based index may struggle with latency/accuracy
Other Mostly Vector Systems

• **Vald**: Architecturally similar to Pinecone, except uses NGT graph index

• **Vearch**
  • Li et al Middleware 2018: Architecturally similar, except uses table-based index and supports predicated search via post-filtering
  • Latest version: Adds support for attribute-only indexes/queries, pre-filtering, multiple index types (HNSW, IVFPQ)

• **EuclidesDB/Chroma**: On-premise centralized
Native Mostly Mixed

Example: milvus

Advantages
✓ Many supported query types, can be configured for both read-heavy and write-heavy workloads

Limitations
✗ Can be resource-intensive due to more sophisticated storage/recovery

More query capabilities and index types

More sophisticated storage to support fast writes in addition to reads

Figure: Wang et al “Milvus: A purpose-built vector data management system”. SIGMOD 2021
Advantages
✓ Many supported query types, can be configured for both read-heavy and write-heavy workloads

Limitations
X Can be resource-intensive due to more sophisticated storage/recovery

Figure: Wang et al “Milvus: A purpose-built vector data management system”. SIGMOD 2021
Other Mostly Mixed Systems

- **Weaviate**
  - Targeted at documents over a graph model; supports both vector search and traditional graph queries via GraphQL
  - HNSW + LSM-Tree, used for raw records + inverted index over keywords and attributes
  - Pre-filtering for predicated search queries

- **Qdrant**: rule-based optimizer for predicated queries

- **NucliaDB/Marqo**
Extended NoSQL

Example: RedisVL

Advantages
√ High-performance vector search, similar to Native Mostly-Vector
√ Combined vector search + non-vector capabilities

Limitations
Χ As with Mostly-Vector, performance is tied to specific workload
Other NoSQL Systems

• **Vespa**
  • Document model
  • SQL-like query language for complex big data analytics
  • Rule-based optimizer for predicated queries

• **Cosmos DB**: proprietary vector index

• **MongoDB**: HNSW bolt-on

• **Neo4j**: HNSW bolt-on

• **Cassandra**: HNSW bolt-on
Extended Relational

Example: PASE

**Advantages**
- ✔ Adaptable to different types of workloads
- ✔ As with Ext NoSQL systems, offers diverse capabilities

**Limitations**
- ✗ May suffer performance overhead (e.g. page indirection)

IndexAmRoutine

Index Interface

- **PASE Index module**
  - build
  - scan
  - common utils

Index Storage

- **PASE Index**
  - IVFFlat Index
  - HNSW Index

Raw Data Tables

Figure from Yang et al. “PASE...”. SIGMOD 2020
Other Relational Systems

- **pgvector**: similar to PASE
- **AnalyticDB+V**
  - Relational OLAP DBMS over disaggregated compute-storage
  - Adds indexing and fast-slow write structures for supporting real-time read/writes over slow-updateable vector indexes, plus accuracy-aware cost estimation model for ANN
- **SingleStoreDB**: Adds sim. scores to enable brute-force vector search
- **ClickHouse, MyScale**
Search Engines and Libraries

Lin et al. “Vector Search with OpenAI Embeddings: Lucene Is All You Need” (2023) arXiv:2308.14963

Search Engines

- Elasticsearch
- OpenSearch
- Solr

Libraries:
- Meta Faiss
- hnswlib
- ANNOY
- Microsoft SPTAG
- KGraph
- E2LSH
- FALCONN
- etc.
Benchmarks

• Surprisingly few benchmarks

• **ann-benchmarks.com**
  • Real implementations, highly implementation-dependent

• **Li et al TKDE 2020**
  • Idealized implementations

Summary of Vector Database Systems

- Latency
- Throughput, scalability

- Retrieval methods
- Storage methods
- Recovery methods
- Elasticity, availability, consistency, security

Performance

Capabilities

Native Mostly-Vector
Native Mostly-Mixed
Extended NoSQL
Extended Relational

Research Trend
Part 3: Challenges and Open Problems
A particular score may not return maximally relevant results, even under high recall:

“The total prod [negative user feedback] rate are 7.3%, 32.6%, and 43.1% at top 1, 3, and 5... roughly 70% are generated by the EBR node during the retrieval stage”

Table 1: Distribution of EBR failures

<table>
<thead>
<tr>
<th>Failure reasons</th>
<th>Percentage</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>irrelevant result (fuzzy text match)</td>
<td>53%</td>
<td>junkiness</td>
</tr>
<tr>
<td>Location mismatch</td>
<td>18%</td>
<td>junkiness</td>
</tr>
<tr>
<td>Language mismatch</td>
<td>4%</td>
<td>junkiness</td>
</tr>
<tr>
<td>Misinformation</td>
<td>10%</td>
<td>integrity</td>
</tr>
<tr>
<td>Untrustworthy results</td>
<td>10%</td>
<td>integrity</td>
</tr>
<tr>
<td>Offensive results</td>
<td>5%</td>
<td>integrity</td>
</tr>
</tbody>
</table>

Wang et al “Integrity and junkiness failure handling for embedding-based retrieval: A case study in social network search”. SIGIR 2023
Disk-Friendly/Distributed Indexes

Graphs are slow for disk-resident datasets

Entry Point Selection  Graph Traversal  Termination
Start  Neighborhood 1  Neighborhood 2  Finish

Disk Layout

Meanwhile, trees/tables are disk-friendly but have worse QPS/recall

Source: ann-benchmarks.com
Update-Friendly Graphs

Accuracy degradation following series of updates

Effect of Repeated Delete-Reinsert Cycles

Singh et al “FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search” 2021. arXiv 2105.09613
Easy-to-Build Graphs

• O(DN log N) still too high for huge N (billions)

• ANN_SIFT1B (128 dimensions):
  • Vamana single: 2 days @ 1100 GB peak memory
  • Vamana merged: 5 days @ 64 GB peak memory

• 200M subset of ANN_SIFT1B:
  • HNSW, ef=500: 5.6 hours @ 64 GB peak memory

Sources:
• Subramanya et al “DiskANN: Fast accurate billion-point nearest neighbor search on a single node”. NeurIPS 2019
New Capabilities

- Multi-Vector Search
  - NRA only works with bounded scores (e.g. cosine)
- Incremental k-NN
- Secure k-NN
- VDBMS Benchmark
Thanks!

Q and A