I/O Optimizations for Graph-Based Disk-Resident Approximate Nearest Neighbor Search: A Design Space Exploration

Liang Li[1] Shufeng Gong[2] Yanan Yang[1] Yiduo Wang[1] Jie Wu[1,3]

[1]China Telecom Cloud Computing Research Institute
[2]Northeastern University
[3]Temple University

VLDB 2026

Outline

Background: Approximate Nearest Neighbor Search

ANN is fundamental to modern applications:

- Information retrieval & search engines
- Recommendation systems
- Retrieval-Augmented Generation (RAG)
- Vector databases

Graph-based methods are SOTA:

- HNSW, NSG, SSG, Vamana (DiskANN)
- High recall + low latency

The Memory Problem

Billion-scale datasets

HNSW memory: $\sim$800 GB

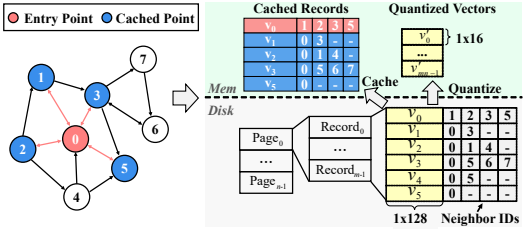
Typical workstation RAM: 64–256 GB

$\Rightarrow$ **Infeasible!**

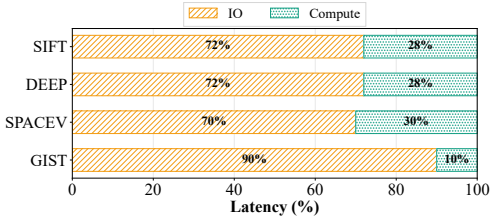
Disk-Based ANN: The I/O Bottleneck

DiskANN: De facto industrial standard

- Used by Azure AI Search, GaussDB, Milvus
- Three-layer architecture:
  1. **Disk layer**: Full vectors + graph
  2. **Memory layer**: PQ compressed codes
  3. **Cache layer**: Hot vertices



DiskANN data layout



Latency breakdown across datasets

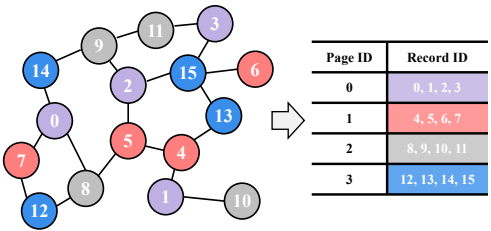
Key Finding

I/O accounts for **70–90%** of query latency!

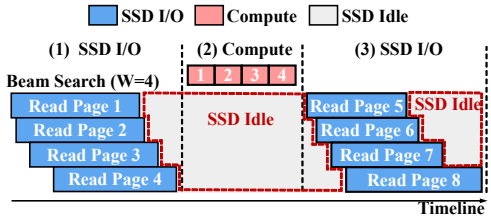
Four Root Causes of I/O Inefficiency

1. **Poor Data Locality**
   Scattered vertices ⇒ massive I/O waste
2. **Low Bandwidth Utilization**
   Sequential I/O–compute execution leaves SSD idle
3. **Long Search Paths**
   Distant entry points ⇒ excessive disk round-trips
4. **Frequent Cache Misses**
   Unpredictable random access patterns

Numerous techniques have been proposed to address these four root causes of I/O inefficiency

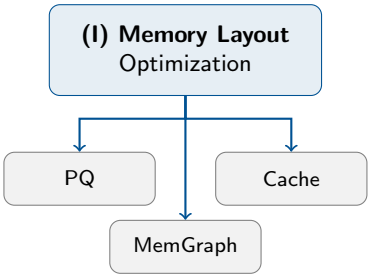


Scattered vertices across pages



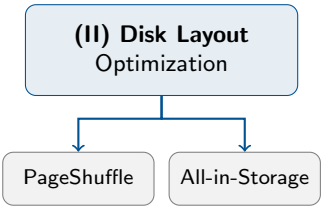
Sequential I/O–compute execution

Systematic Three-Dimensional Taxonomy



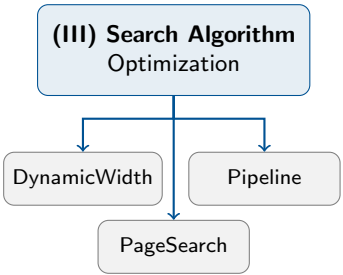
Memory Layout

Reduce disk I/O frequency via in-memory structures



Disk Layout

Optimize physical data placement on storage



Search Algorithm

Improve traversal execution efficiency

Memory Layout: Product Quantization (PQ)

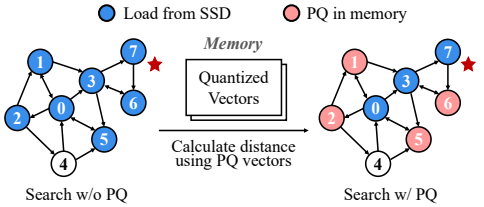
Core Idea: Two-stage search

1. ❶ **Approximate filtering**
   Use memory-resident PQ codes for ranking
2. ❷ **Precise refinement**
   Read full vectors only for top candidates

I/O Complexity Reduction:

$$O\left(\frac{\bar{R} \cdot H}{OR \cdot n_p}\right) \longrightarrow O\left(\frac{H}{OR \cdot n_p}\right)$$

Eliminates the $\bar{R}$ factor!



Quantization reduces disk reads

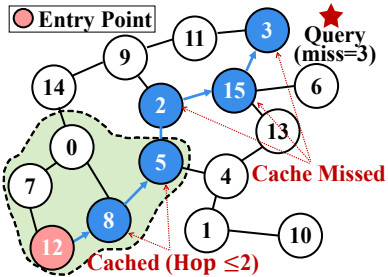
Memory Layout: Cache & MemGraph

Cache Management

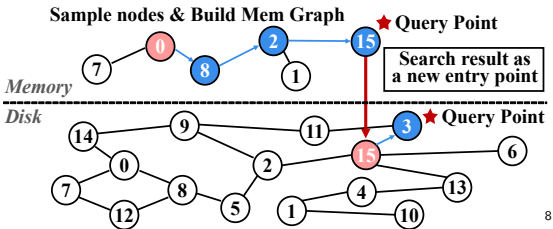
- SSSP-based caching from entry points
- Reduces early-hop I/O
- Effective when graph quality is high

Hierarchical Graph (MemGraph)

- In-memory coarse navigation graph
- Provides high-quality entry points
- Key insight: Entry point quality determines search path length!



Cache miss illustration



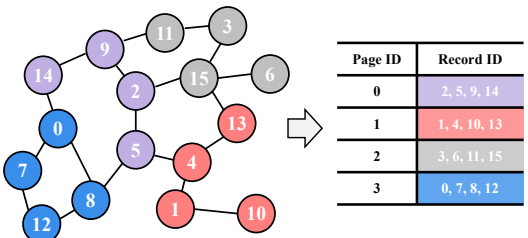
Disk Layout: Page Shuffle

Problem: Vertex-ID sequential storage scatters neighbors across pages

- Low overlap ratio (∼6.25% on SIFT1B)
- Each I/O serves few useful neighbors

Solution: Page Shuffle

- Reorganize physical data placement
- Group neighboring vertices on same page
- Dramatically improves overlap ratio



Page shuffle illustration

Disk Layout: All-in-Storage (AiSAQ)

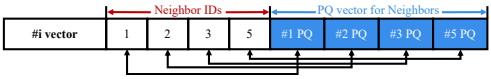
Core Idea: Move everything onto SSD

- Both full-precision vectors *and* PQ codes reside on SSD
- Co-locate vector, neighbor IDs, and neighbors' PQ codes within the same storage unit
- One read per hop fetches all needed data

Trade-offs:

- \+ Minimal memory footprint, fast cold-start
- – Substantial on-disk expansion
- – Read amplification

Excluded from our evaluation: targets memory-constrained scenarios whose goal (memory minimization) conflicts with our focus on I/O efficiency under realistic memory budgets



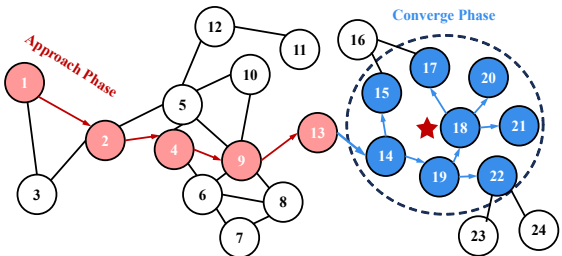
AiSAQ layout: each record stores the node vector, neighbor IDs, plus PQ codes of neighbors

Search Algorithm: Dynamic Width & Pipeline

Dynamic Width (DW)

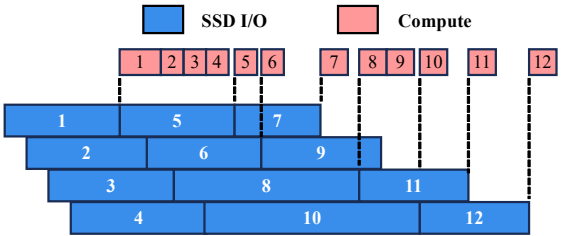
- Two-phase search behavior:
  1. *Approach phase*: small width
  2. *Converge phase*: large width
- Suppresses wasted reads early
- Improves I/O utilization:

$$U_{io} = \frac{N_{eff}}{N_{read}}$$



Pipeline Search

- Overlap I/O and computation
- Continuous I/O pipelining
- Maximizes bandwidth utilization



Pipelined I/O processing

Techniques vs. I/O Challenges

Method	I/O Waste	BW Util.	Long Path	Cache Miss
PQ	✓	—	—	—
Cache	—	—	—	✓
MemGraph	—	—	✓	—
PageShuffle	✓	—	—	—
All-in-Storage	×	—	—	—
DynamicWidth	✓	—	✓	—
Pipeline	×	✓	—	—
PageSearch	✓	×	—	—

Insight

Different techniques address different bottlenecks ⇒ potential for **synergistic combinations**

Algorithmic Landscape

	Memory			Disk		Search		
	PQ	Cache	MemG	PS	AiS	DW	Pipe	PSe
DiskANN	●	●	×	×	×	×	×	×
Starling	●	×	●	●	×	×	×	●
PipeANN	●	×	●	×	×	●	●	×
AiSAQ	●	×	×	×	●	×	×	×

Observation

Existing systems optimize along **single dimensions** ⇒ room for multi-dimensional synergy!

Experimental Setup

Hardware

- CPU: Intel Xeon w7-3455 (24 cores)
- RAM: 128 GB DDR4
- Storage: 4 TB NVMe SSD
- 4KB random read: **819K IOPS**

Datasets

Dataset	Dim	#Vectors	Type
SIFT	128	100M	uint8
DEEP	96	100M	float
SPACEV	100	100M	int8
GIST	960	1M	float

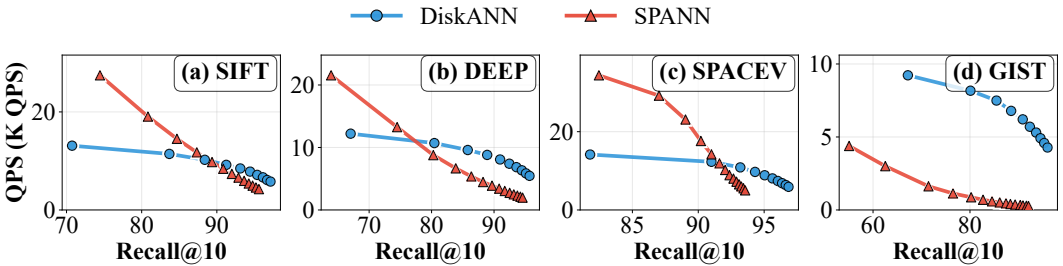
Evaluation Metrics

- Throughput (QPS)
- Latency (ms/query)
- Accuracy (Recall@10)
- I/O operations per query
- IOPS & Bandwidth utilization

Baseline Configuration

- $R = 64$, $L = 125$, $\alpha = 1.2$
- Beam width: $\omega = 8 \rightarrow 32$
- Page size: 4 KB (8/16 KB for GIST)

Graph-Based vs. Inverted Indices (Finding 1)



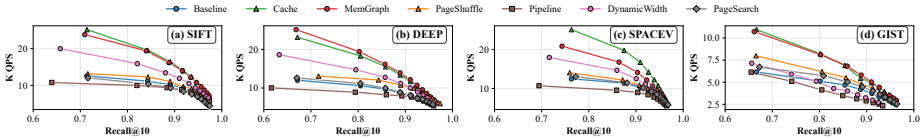
SPANN vs. DiskANN across four datasets

Finding 1: Graph Indices Win for ANN

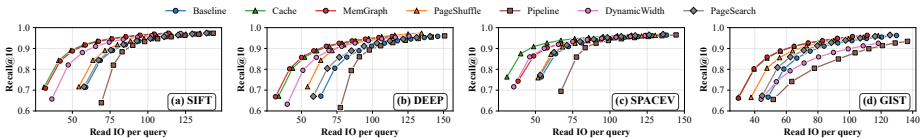
- SPANN wins only on **low-dimensional data at low recall**; it degrades fast as recall rises
- High-dimensional data: SPANN suffers the **curse of dimensionality**
- SPANN index size: 1.5×–3.4× larger than DiskANN

⇒ We focus on **graph-based** disk-resident ANN

Individual Optimization Results



Recall–QPS trade-off



I/O operations per query

Finding 2: I/O Dominates Latency

Latency-recall and I/O-per-query curves align closely

Key Findings: Single-Factor Optimizations

Finding 3: Strong Standalone Gains

Best single-factor optimizations:

- `MemGraph`: -45% I/O, $+90\%$ QPS
- `Cache`: Strong on SPACEV
- `DynamicWidth`: -25% I/O, $+50\%$ QPS

Target root causes: poor entry points & over-expansion

Finding 4 & 5: Weak or Counterproductive

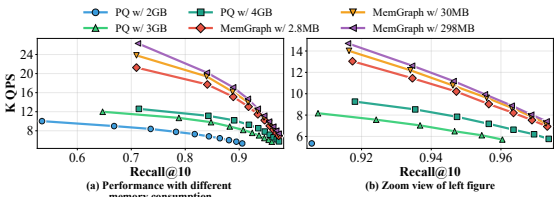
Limited standalone impact:

- `PageShuffle`: $\sim$9% I/O reduction
- `PageSearch`: May degrade performance

Counterproductive:

- `Pipeline`: Speculative reads hurt!

Memory Budget Allocation



Performance vs. memory budget allocation

Finding 6: Budget Allocation Guideline

When memory headroom exists:

1. First: Allocate to `MemGraph`
   ⇒ Higher throughput
2. Then: Increase `PQ` dimensions
   ⇒ Better accuracy

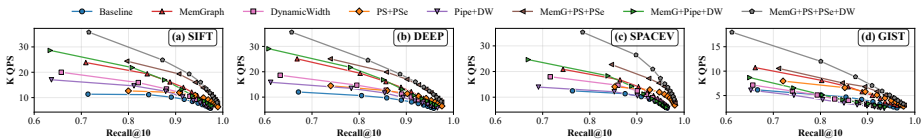
`MemGraph` overhead: ∼30–50 MB

`PQ` overhead: several GB

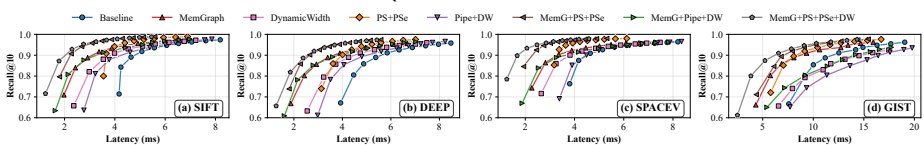
Five Progressive Combinations:

- **C1** (PS + PSe): PageShuffle + PageSearch
- **C2** (Pipe + DW): Pipeline + DynamicWidth
- **C3** (MemG + PS + PSe): Add MemGraph to C1
- **C4** (MemG + Pipe + DW): Add MemGraph to C2
- **C5** (MemG + PS + PSe + DW): **OctopusANN**

Combination Results



Recall–QPS: combinations C1–C5



Latency–Recall: combinations C1–C5

Synergy Findings

Finding 8: PS + PSe Synergy

- `Page Shuffle`: Raises data locality
- `Page Search`: Raises per-page utility
- **Complementary goals!**
- +10% recall gain, +27% QPS

Finding 9: Pipeline + DynamicWidth

- Complementary but not coequal
- `DynamicWidth` dominates
- Pipeline: only for low concurrency

Finding 10: OctopusANN (C5)

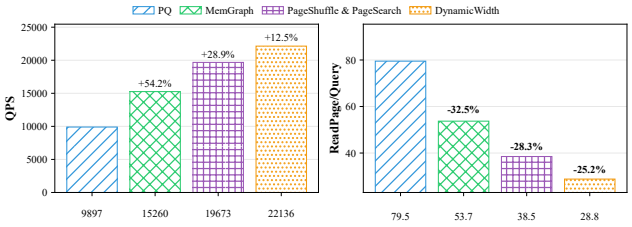
Best overall combination:

- PQ + MemGraph + PS + PSe + DW
- Up to **+133%** QPS on SPACEV
- Modest overhead cost

Sources of gains:

- Higher page utilization (PS+PSe)
- Faster convergence (MemG+DW)

OctopusANN Breakdown Analysis



Incremental contribution of each technique (SIFT)

QPS Improvement:

- `MemGraph`: **+54.2%**
- `PS+PSe`: **+28.9%**
- `DynamicWidth`: **+12.5%**

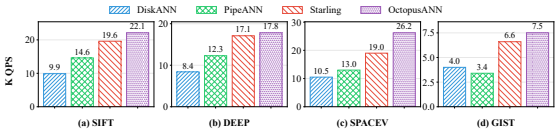
I/O Reduction:

- `MemGraph`: **−32.5%**
- `PS+PSe`: **−28.3%**
- `DynamicWidth`: **−25.2%**

⇒ Gains come from I/O optimization!

State-of-the-Art Comparison

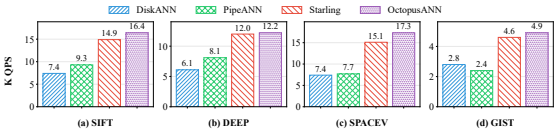
Recall@10 = 90%



OctopusANN vs. Starling & DiskANN

- vs. Starling: **+4.1–37.9%**
- vs. DiskANN: **+87.5–149.5%**

Recall@10 = 95%



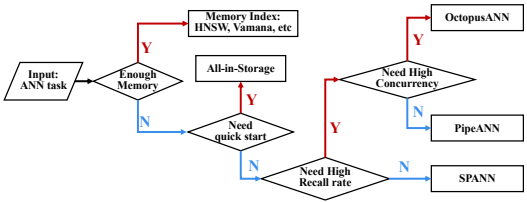
At higher accuracy target

- vs. Starling: **+1.6–14.6%**
- Gains diminish at high recall

Finding 11

Multi-dimensional optimization most effective at **moderate accuracy** (90% recall)

Practical Guidelines



Decision guide for system selection

Selection Workflow:

1. Memory sufficient?
   → In-memory (HNSW/Vamana)
2. Need fast cold-start?
   → All-in-Storage
3. Low-dim + relaxed accuracy?
   → SPANN (inverted index)
4. **High concurrency?**
   - High recall → **OctopusANN**
   - Moderate recall → MemGraph
5. **Low concurrency?**
   → PipeANN

Summary & Contributions

Three Key Contributions

1. **Systematic Taxonomy**
   Three-dimensional framework unifying I/O optimization techniques
2. **Controlled Evaluation**
   Apples-to-apples comparison across 4 datasets, 7 techniques
3. **OctopusANN & Guidelines**
   Principled multi-dimensional combination with actionable insights

Take-Home Message

Systematic composition $>$ **isolated tweaks**
when pushing the performance frontier of disk-based ANN

Thank You!

Questions?

Code: `https://github.com/LeonLee666/IObench4DiskANN`