Vector Search for the Future: From Memory-Resident, Static Heterogeneous Storage, to Cloud-Native Architectures

Yitong Song[1] Xuanhe Zhou[2] Christian S Jensen[3] Jianliang Xu[1]

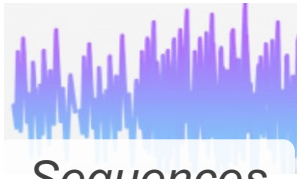
[1]Hong Kong Baptist University
[2]Shanghai Jiao Tong University
[3]Aalborg University

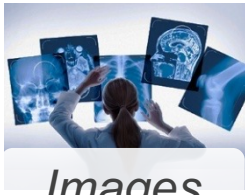
2026/6/2

Vector Data

Unstructured Data


Text


Sequences


Images


Video


Audio

TS2Vec

ResNet-50

CLIP

Embedding Models

Cohere Embed

BERT

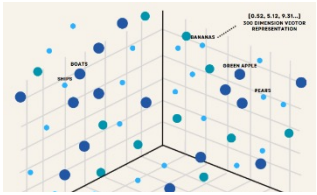
Embedding Vectors

[0.95, 0.49, 0.67, 0.75, 0.43, 0.00, 0.60, 0.13, 0.72, ...],
[0.73, 0.83, 0.80, 0.48, 0.55, 0.83, 0.86, 0.85, 0.06, ...],
[0.03, 0.71, 0.62, 0.07, 0.74, 0.30, 0.80, 0.08, 0.63, ...],
[0.49, 0.48, 0.38, 0.51, 0.45, 0.05, 0.10, 0.92, 0.29, ...],
[0.84, 0.38, 0.81, 0.00, 0.48, 0.61, 0.41, 0.85, 0.93, ...],
[0.24, 0.90, 0.69, 0.57, 0.43, 0.60, 0.37, 0.63, 0.28, ...],
[0.72, 0.33, 0.36, 0.98, 0.88, 0.05, 0.64, 0.91, 0.15, ...],
[0.89, 0.17, 0.44, 0.78, 0.33, 0.59, 0.29, 0.23, 0.22, ...],
[0.88, 0.75, 0.38, 0.77, 0.31, 0.98, 0.84, 0.79, 0.25, ...],
[0.11, 0.77, 0.30, 0.56, 0.62, 0.14, 0.89, 0.19, 0.62, ...],
[0.63, 0.41, 0.97, 0.73, 0.64, 0.41, 0.96, 0.36, 0.87, ...],
[0.51, 0.16, 0.26, 0.94, 0.53, 0.38, 0.39, 0.90, 0.55, ...],
[0.91, 0.99, 0.10, 0.87, 0.24, 0.91, 0.94, 0.36, 0.69, ...],
[0.89, 0.64, 0.14, 0.94, 0.51, 0.24, 0.61, 0.99, 0.11, ...],
[0.77, 0.65, 0.48, 0.94, 0.49, 1.00, 0.24, 0.10, 0.97, ...],
[0.58, 0.47, 0.01, 0.56, 0.59, 0.32, 0.79, 0.95, 0.40, ...]

Dense Float-Point Data

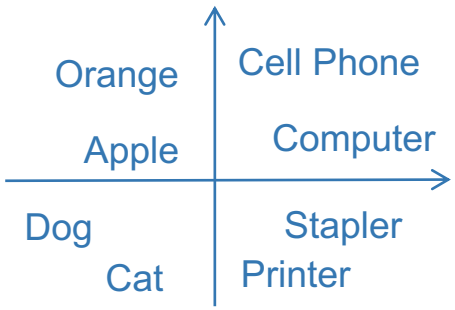
- ***High Dimensionality***

  Objects are represented using a large number of features (i.e., dimensions)



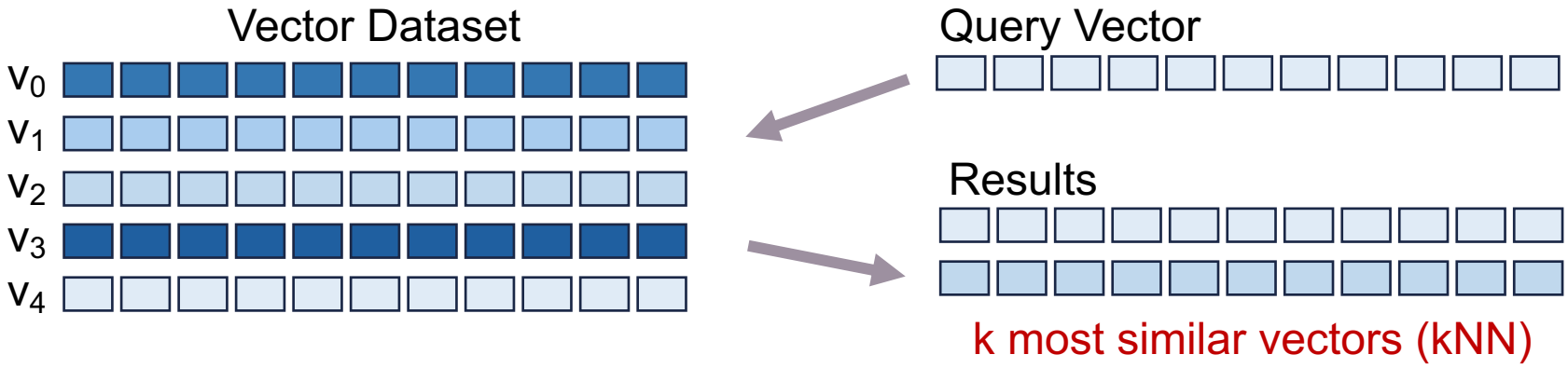
- ***Measured by Distances***

Similarity between objects ⟷ Distances between vectors

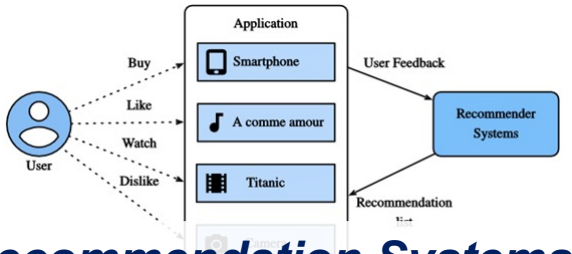


Vector Search

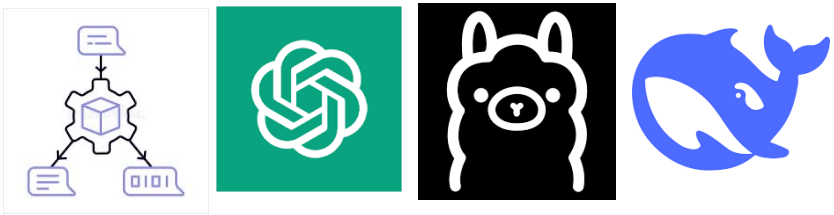
Vector Search (VS)



Applications



Recommendation Systems



RAG / LLM Retrieval

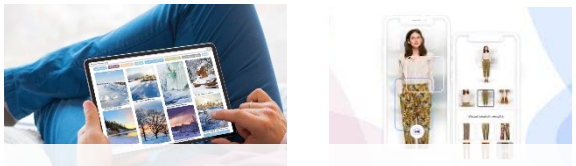


Image-to-Image Search

Music Recognition

TRAE

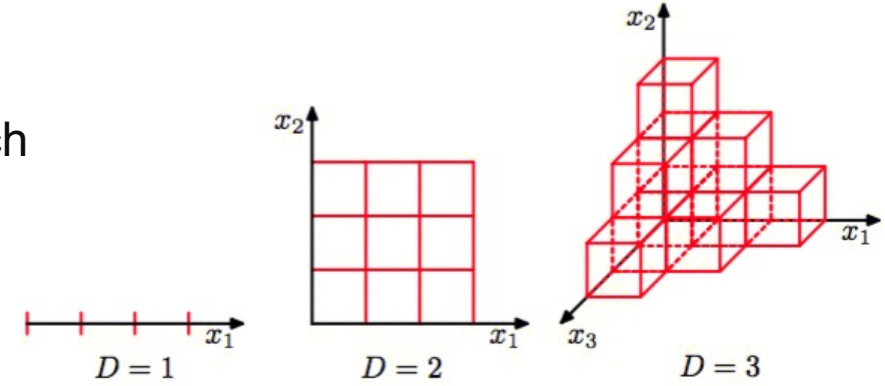
CURSOR

Code Assistants

Challenges of VS

Curse of Dimensionality

As the dimensionality increases, the search space grows exponentially



- Exact Search → Approximate Search
  (kNN → ANN)
- Accuracy is measured by recall

The percentage of correct answers in the query results

Groundtruth  Query results 

Recall = $\frac{4}{5} = 80\%$

Computational and Storage Intensive

Floating-point numerical vectors: [0.12, -0.53, 0.91, ...]

- L2 Distance Computation Cost: O(N × d)

$$d(q,x) = \sum_{i=1}^{d} (q_i - x_i)^2$$

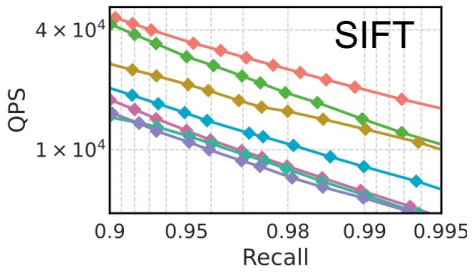
- Storage Footprint: 768 dims × 4 bytes ≈ 3 KB per vector

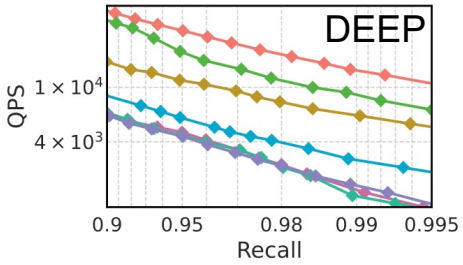
1B vectors → ~3 TB

Indexing and Compressing

Index Type	Representative Methods
Tree-based [IEEE TC75, KDD19, etc.]	KD-Tree, R-Tree (*Inefficiency in high-dimensional spaces*)
Hash-based [VLDB15, TKDE23, etc.]	LSH (Local-Sensitive Hash)
IVF-based [Faiss library]	IVF-Flat, IVF-PQ
Quantization-based [TPAMI11, SIGMOD24, etc.]	SQ, PQ, RaBitQ (also used for vector compression)
Graph-based [TPAMI18, VLDB19, etc.]	HNSW, NSG, Vamana

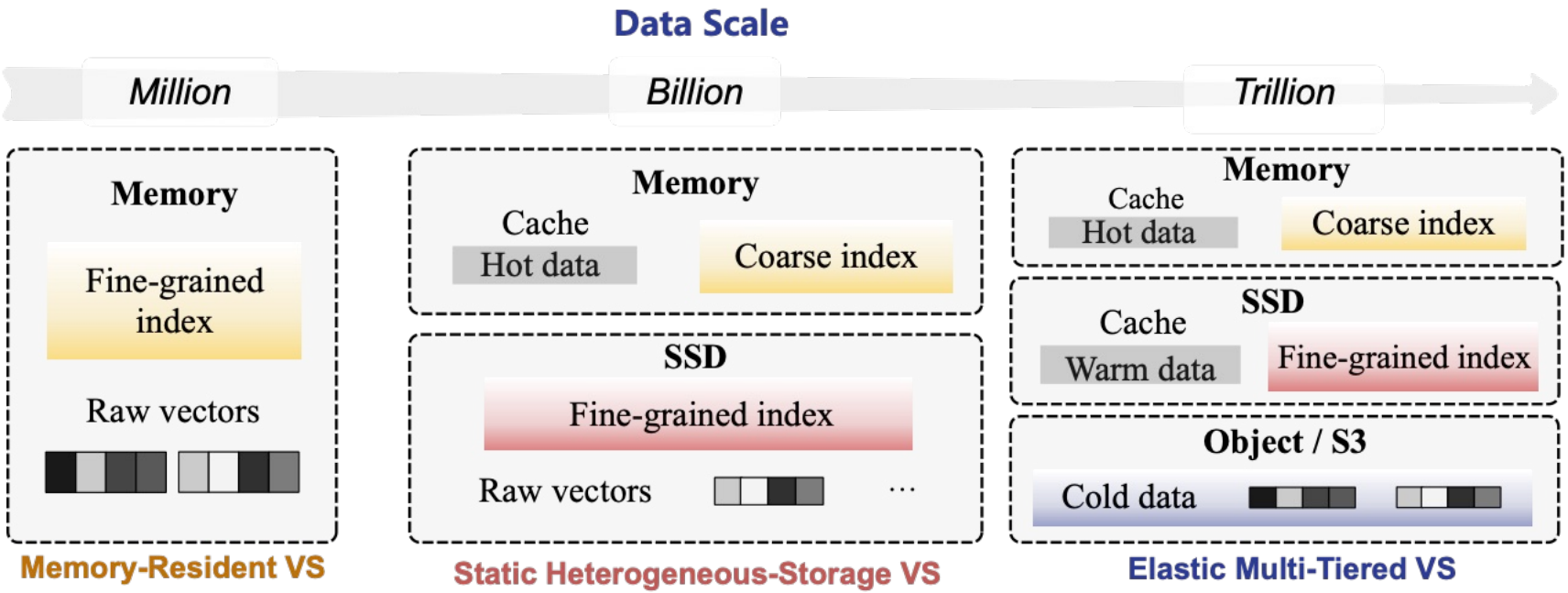
Focusing on high efficiency (QPS) and recall trade-offs





Mainly designed for memory-resident vectors

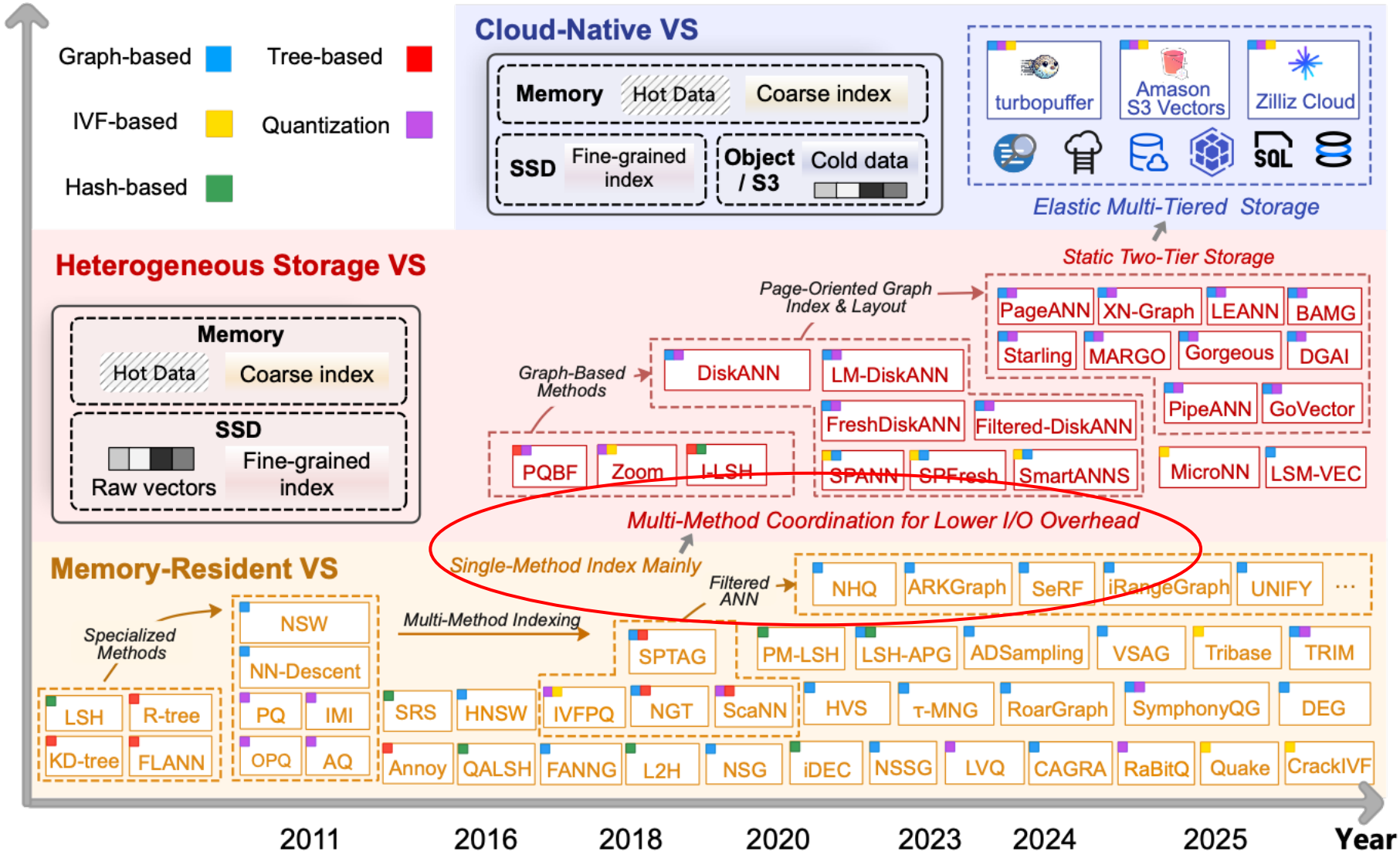
Storage Architecture Evolution with Data Scale



Storage architectural evolution reshapes the VS techniques

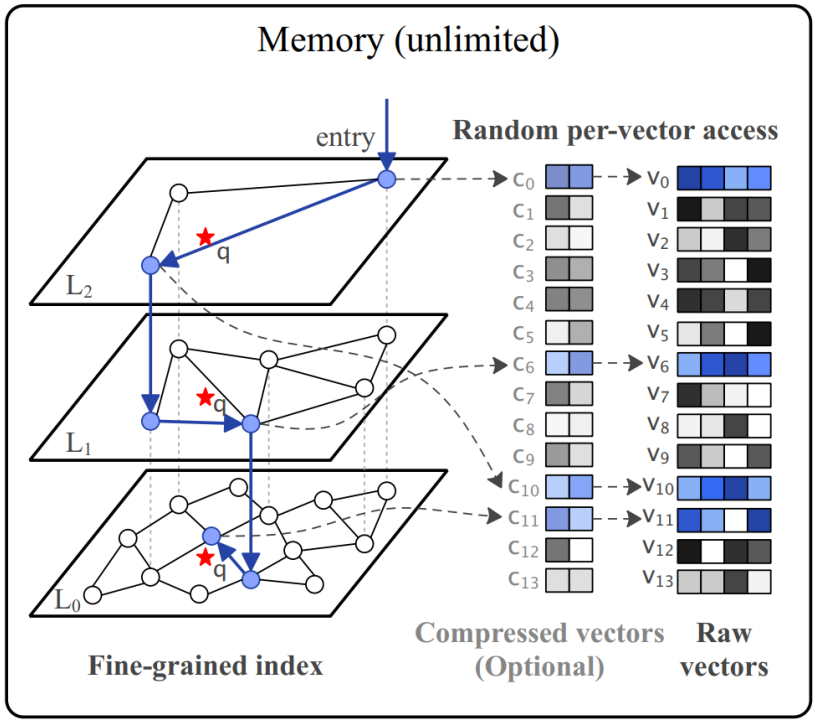
- **Memory-Resident VS**: Store both raw vectors and indexes entirely in memory
- **Static Heterogeneous-Storage VS**: Space-intensive raw vectors and fine-grained index structures are placed on SSDs, while compressed vectors or coarse navigational structures are kept in memory
- **Elastic Multi-Tiered VS**: Dynamically place vectors and indexes across tiers according to access patterns and cost-performance considerations

Evolution of VS Techniques

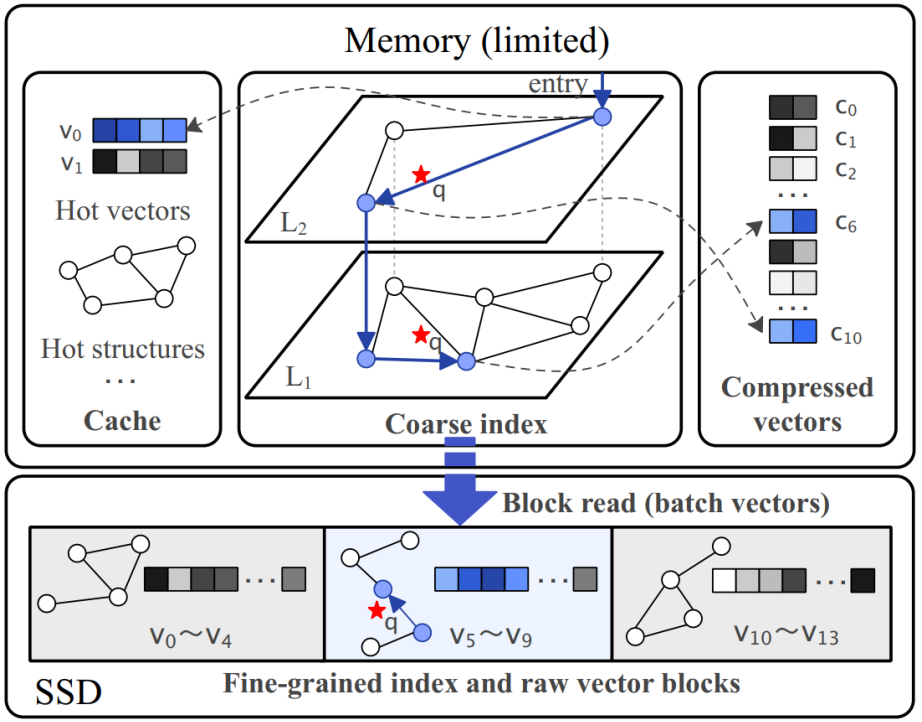


Computational bottleneck ⟶ IO bottleneck

Memory-Resident vs. Memory-SSD VS



(a) Memory-Resident VS

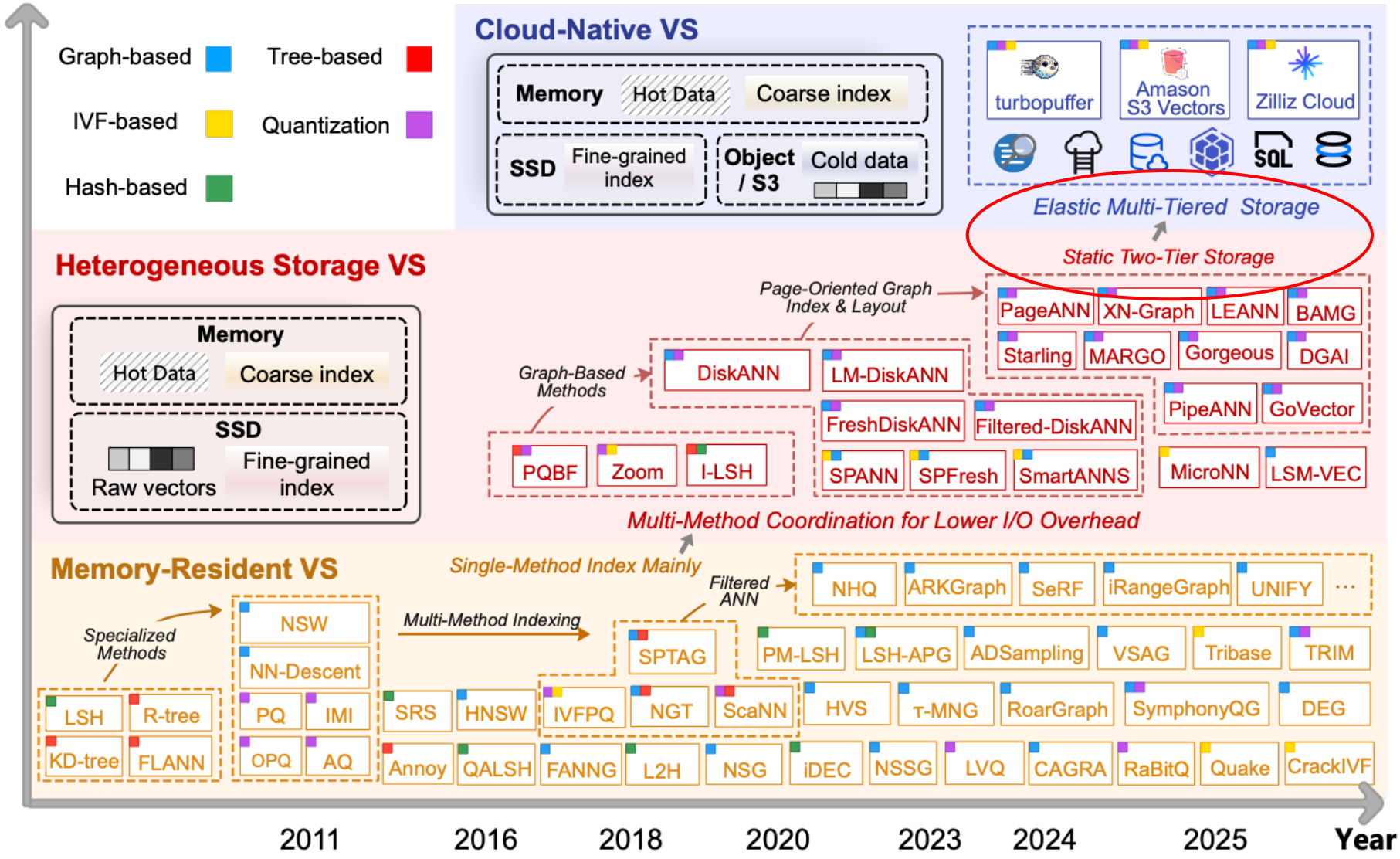


(b) Memory-SSD VS

(illustrated with graph-based methods)

- Random per-vector access → Block-level multi-vector access
- Additional trade-offs between query efficiency and memory usage

Evolution of VS Techniques



Cost–Performance Trade-Offs & Network Bottlenecks

Tutorial Overview

Part 1: Memory-Resident VS

Part 2: Heterogeneous-Storage VS

Part 3: Elastic Multi-Tiered VS

Part 4: Challenges and Open Problems

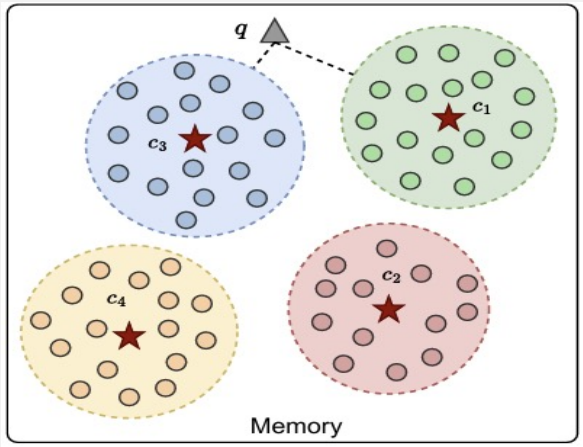
Part 1: Memory-Resident VS

- IVF
- Quantization
- Graph

IVF-Based Methods

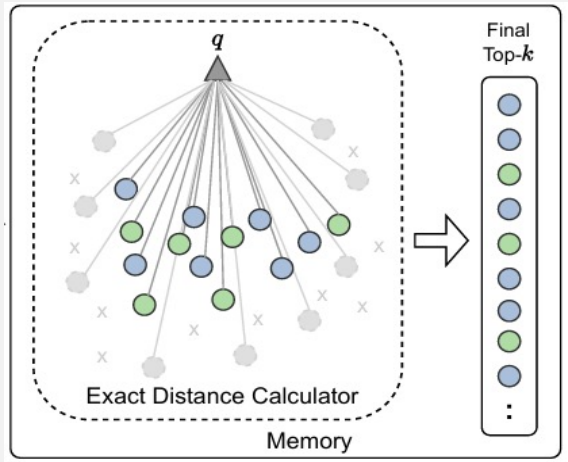
Organize vectors into clustered partitions and restrict the search to the nearest clusters, thereby reducing computational cost

Index Construction



- Using k-means to divide the dataset into clusters, each with a Centroid
- Vectors of the same centroid are stored in one inverted list

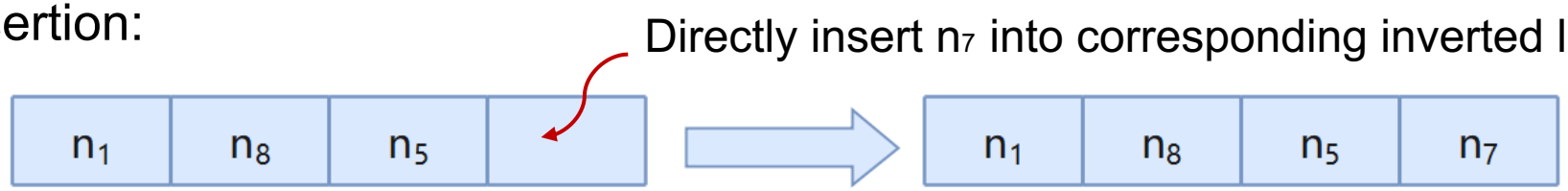
Two-Phrase Search



- Finding *top-nprobe* closest clusters
- Computing distances against all vectors within those clusters to get final top-k results

IVF-Based Methods

Insertion:



Pros.	• Outstanding QPS for batch queries (leveraging parallel computing) • Extremely fast vector insertions
Cons.	• High recall requires accessing numerous clusters • Updates can introduce cluster imbalance, degrading efficiency

Best suited for:

- ☐ Scenarios maximizing QPS via heavy batch queries
- ☐ Read-heavy, static or append-mostly workloads

Less suited for:

- ☐ Latency-critical single-query (low-batch) scenarios
- ☐ Highly skewed data distribution

Quantization-based -- SQ (Scalar Quantization)

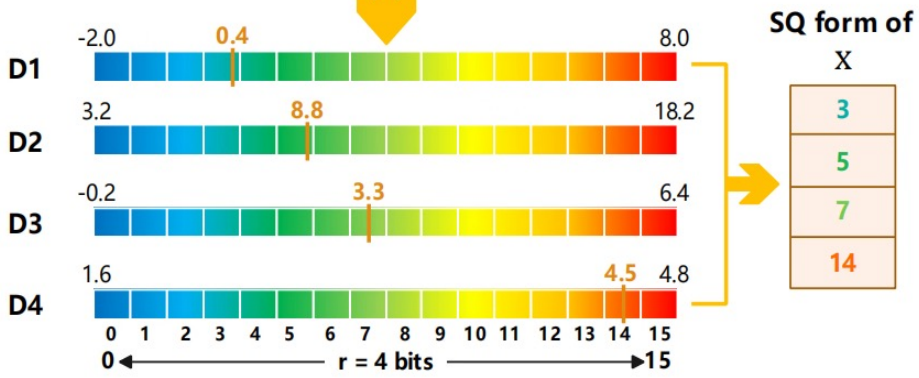
SQ maps float values of each dimension to low-bit integers to reduce storage cost and accelerate computations

Uniform SQ (equal-width bins)

$$\Delta_j = \frac{b_j - a_j}{2^r}$$

$$q_j = \left\lfloor \frac{x_j - a_j}{\Delta_j} \right\rfloor$$

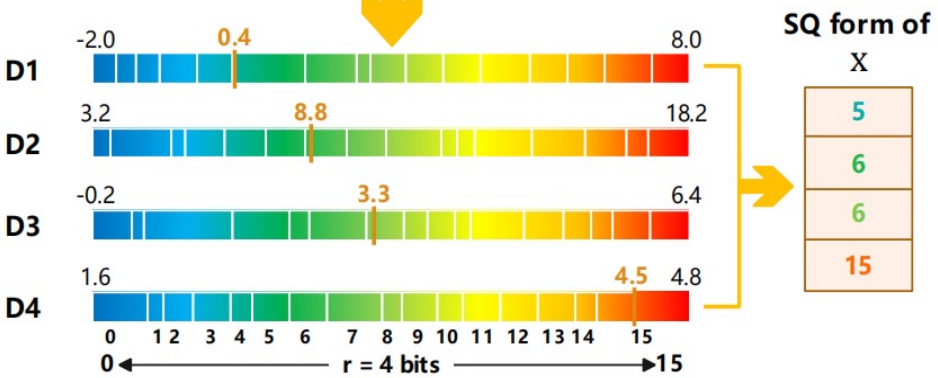
x = [0.4 , 8.8 , 3.3 , 4.5]



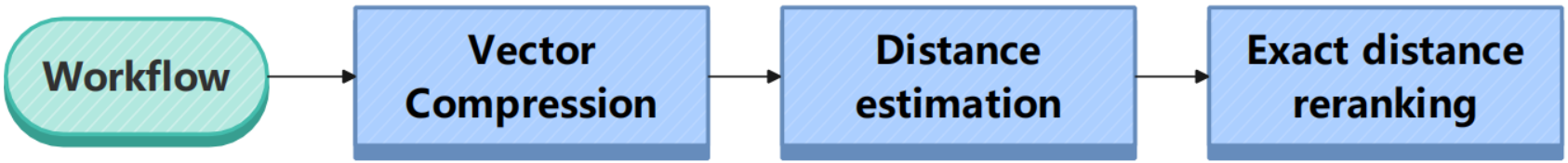
Simple, Fast, Low-Cost

Non-uniform SQ (density-aware / learned bins)

x = [0.4 , 8.8 , 3.3 , 4.5]

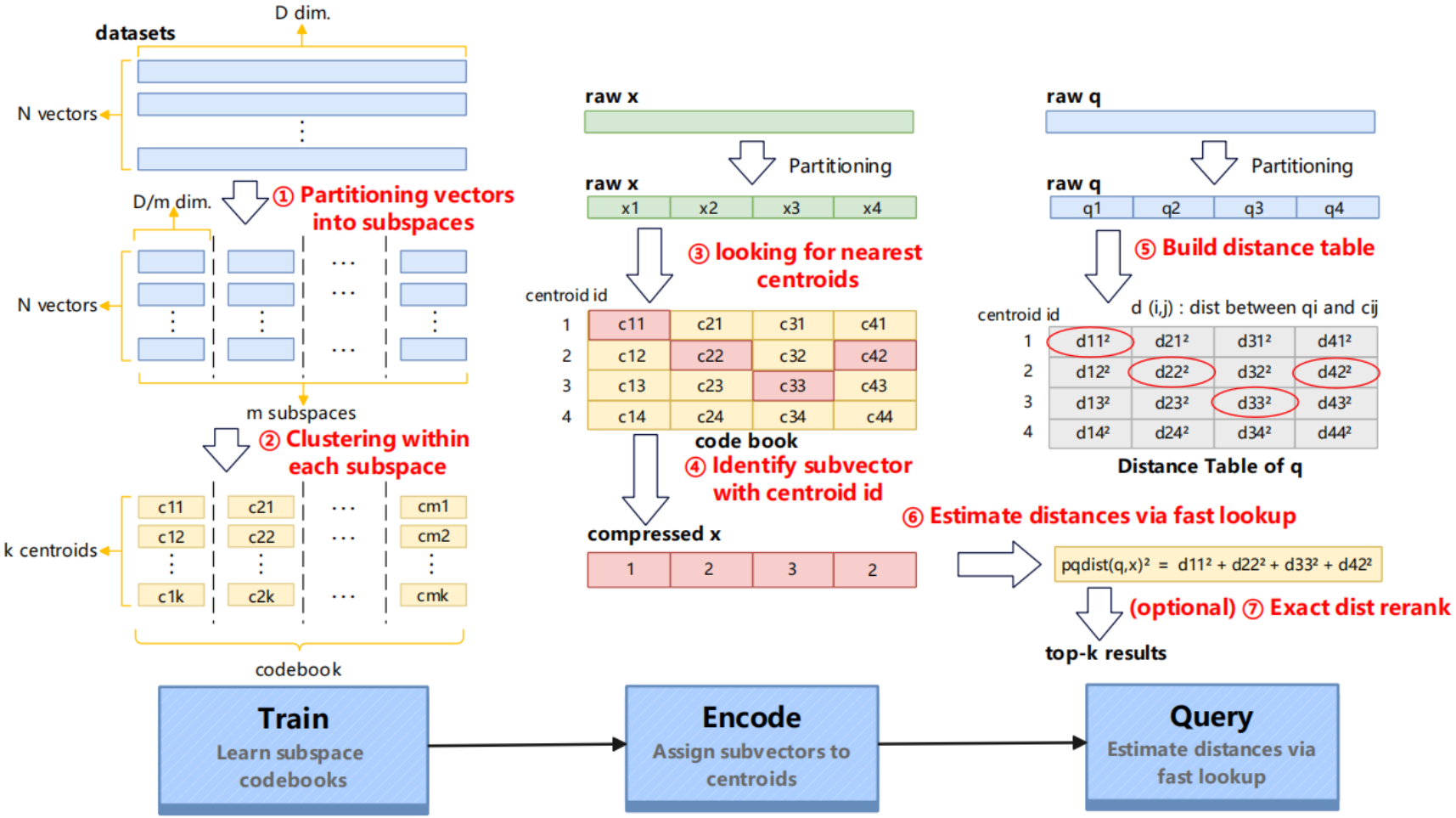


Better fit to data distribution



Quantization-based -- PQ (Product Quantization)

PQ splits a vector into sub-vectors, then maps each sub-vector to its nearest centroid and stores only the centroid index



Quantization-based -- RaBitQ

RaBitQ encodes vector directions into binary codes for compact storage and efficient distance estimation

Step 1: Normalization
Project vectors onto unitball

Step 2: Random Codebook:
Uniforminze quantized points

c: centroid

o_r,q_r: raw vector

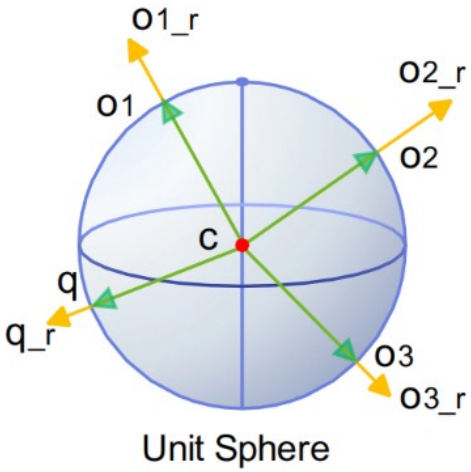
o,q: normalized vector

$$o=\frac{o_r-c}{\|o_r-c\|},\ q=\frac{q_r-c}{\|q_r-c\|}$$

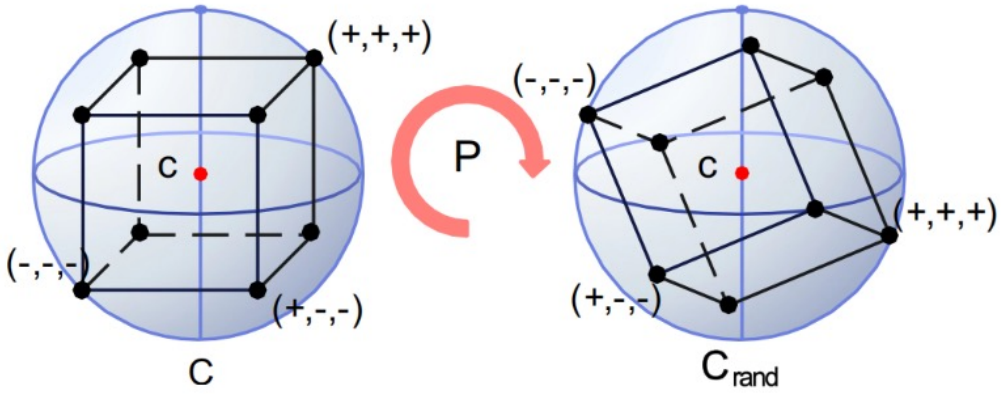
Codebook $C:=\left\{+\frac{1}{\sqrt{D}},-\frac{1}{\sqrt{D}}\right\}^D$, $\|C\|=2^D$

Random Orthogonal Matrix: P

Randomized Codebook $C_{rand}:=\{Px \mid x \in C\}$



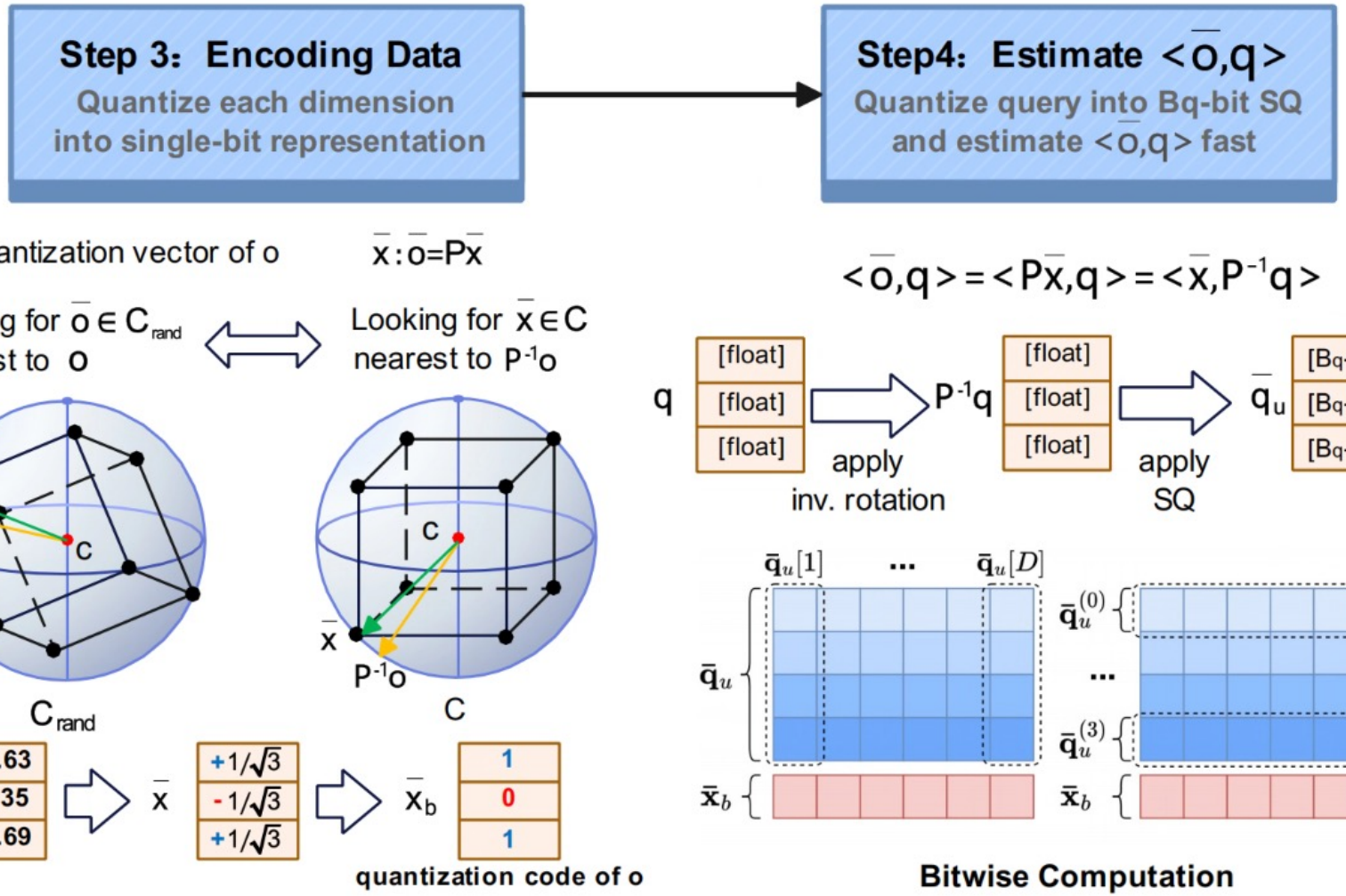
Unit Sphere



Only keeping the direction

Uniformize codewords to remove preference of data vectors

Quantization-based -- RaBitQ



Looking for nearest point in the random rotation space

SIMD-friendly / FastScan-friendly

Quantization-based

Pros.	SQ	☐ Simplest design ☐ Lightweight	☐ Low computation overhead ☐ Low storage overhead ☐ Fast approximate distance evaluation
	PQ	☐ Efficient lookup ☐ Subspace codebooks	
	RaBitQ	☐ Theoretically robust error control ☐ Storage-free codebook ☐ Bit-wise & hardware-friendly	
Cons.	SQ	Disregarding inter-dimensional correlations	☐ Limited recall due to inherent precision loss
	PQ	High training overhead	
	RaBitQ	\	

Best suited for:

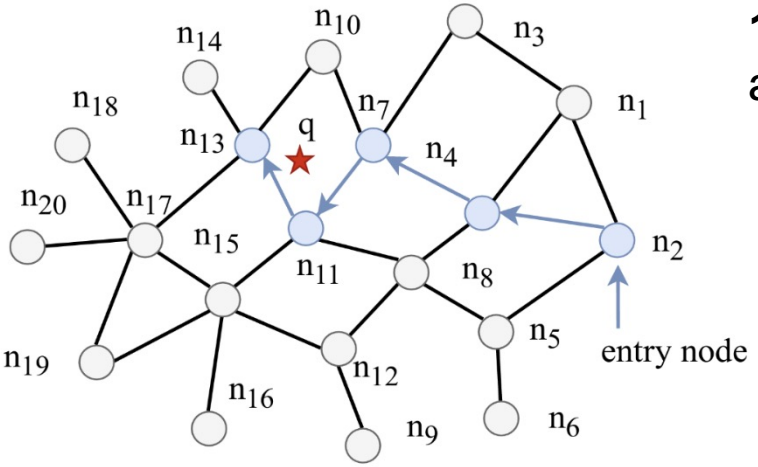
- ☐ Memory-limited scenarios
- ☐ Latency-sensitive scenarios

Less suited for:

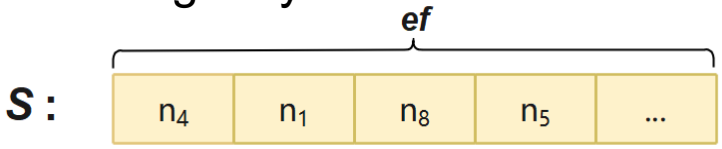
- ☐ Accuracy-critical workloads

Graph-Based Methods -- PG(Proximity Graph)

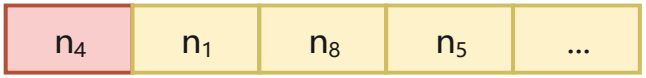
Graph-based methods construct a proximity graph by linking each vector to its approximate nearest neighbors, and perform search via greedy or best-first traversal



1. Initialization: Starting from an Entry Node and maintaining a dynamic candidate set ***S***



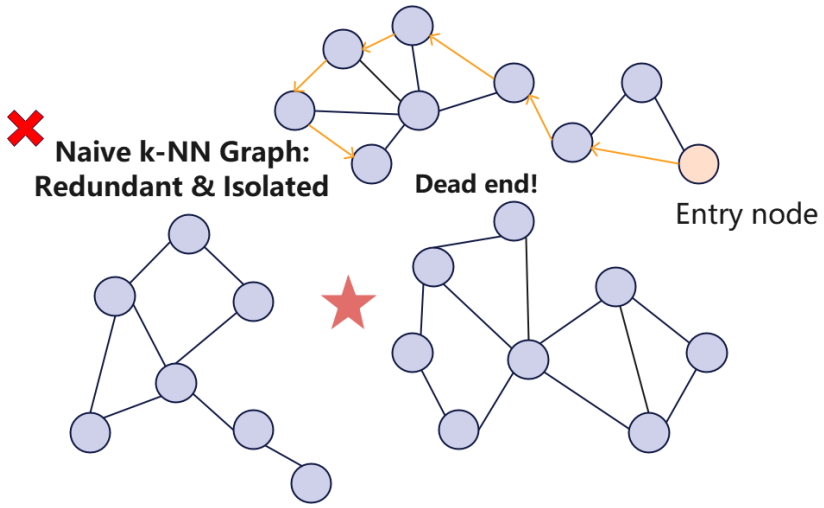
2. Expansion: Always picking the closest unvisited node in ***S*** for expansion

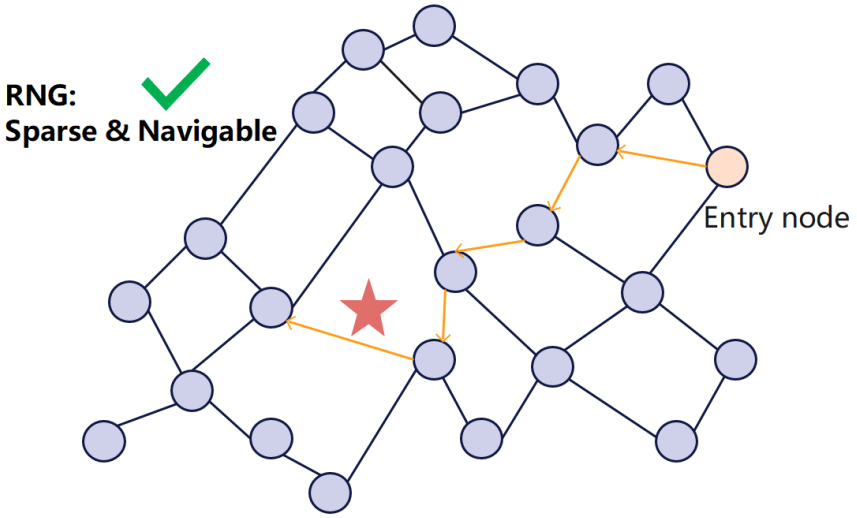


3. Refinement: Computing distances for its neighbors and pushes them into ***S*** *and update* ***S*** *to* keep the closest ***ef*** *(*$ef \geq k$*)* nodes

4. Termination: Returning ***top-k*** results in ***S*** when no neighbors are closer to the query can be found

Graph-Based Methods -- PG Construction



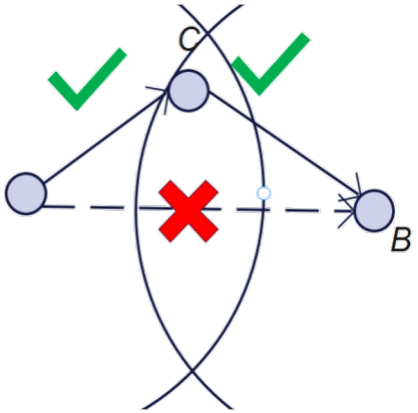


Why not naive k-NN graph?

Connecting simple ***k*** nearest neighbors creates **useless cliques**, thus leading to **degraded graph connectivity**

Solution: RNG (Relative Neighborhood Graph)

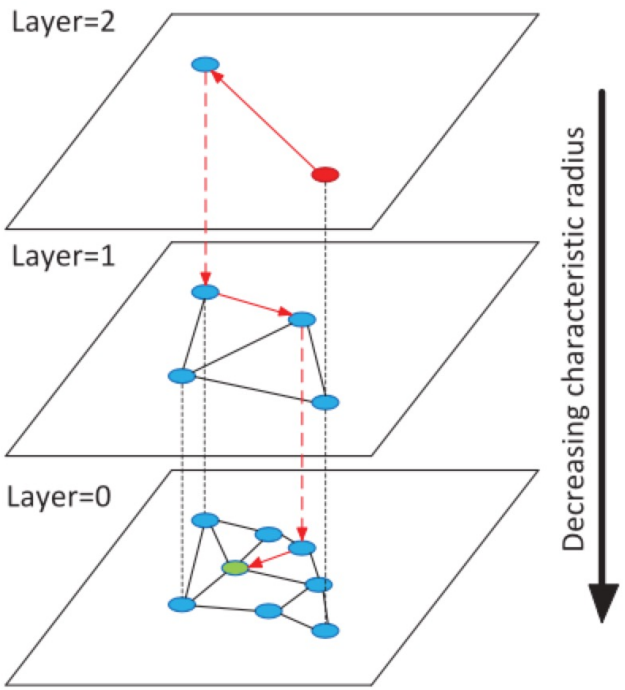
- **Principle**: Linking nearest points in **different directions**
- **Example**: A is not linked to B, as B lies in approximately the same direction as its neighbor C



Graph-Based Methods -- HNSW

Why not one-layer graph ? Why HNSW?

A single-layer graph easily gets stuck in local optima and require long traversal paths. HNSW builds multi-layer graphs, where higher layers contain fewer points that act as "highways" for faster navigation



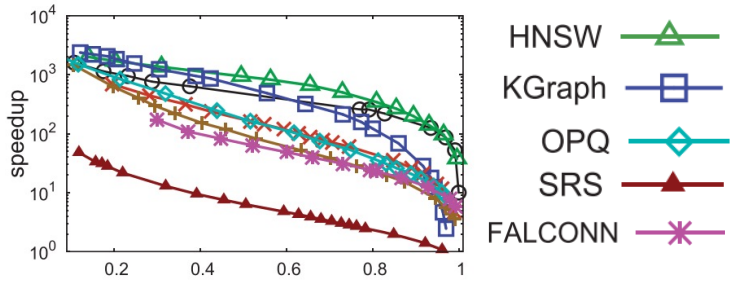
- ***Top Layers***: Extremely sparse, fewer points used for fast, long-range jumps
- ***Bottom Layer***: Containing all vectors for fine-grained search

Search Strategy

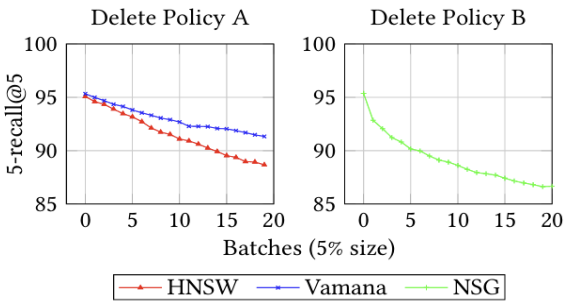
1. Start at the top layer and use "**highways**" to quickly approach closer points
2. When no closer point is found, descend to the next layer
3. Repeat until Layer 0, where a standard greedy search is performed

Graph-based Methods

Pros. Best latency & recall trade-off



Cons. Quality degradation under deletions



Pros. Efficient data insertions

Insertion complexity is $O(logn)$ (similar to search)

Cons. High index size

100 million nodes × 16 neighbors ≈ tens of GB

Best suited for:

- Low latency and recall-critical tasks
- Sufficient memory budget

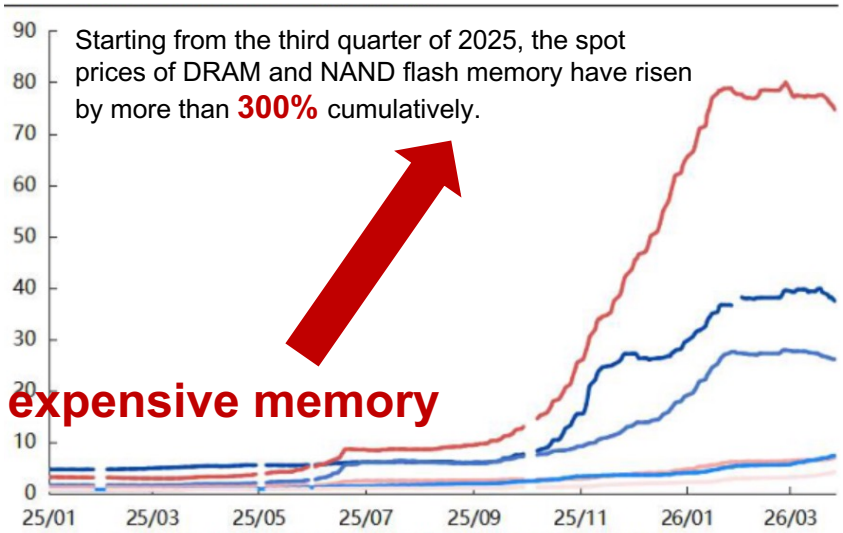
Less suited for:

- Extremely large-scale datasets with tight memory budgets

Part 2: Heterogeneous-Storage VS

Part 2: Heterogeneous-Storage VS

2025/01-2026/03 DRAM Spot Average Price



Source: DRAMexchange, Global Semiconductor Observation, AJ Securities Research Institute

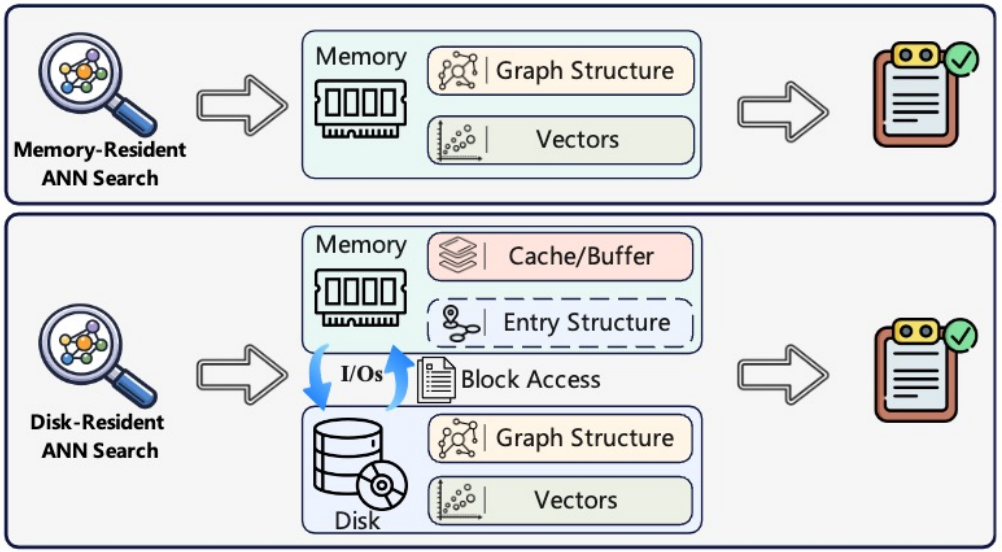
More expensive memory

- **Spot Avg Price: DRAM: DDR5 16G (2Gx8) 4800/5600 (USD)**
- **Spot Avg Price: DRAM: DDR4 16Gb (2Gx8) 3200 (USD)**
- **Spot Avg Price: DRAM: DDR4 8Gb (512Mx16) 3200 (USD)**
- **Spot Avg Price: DRAM: DDR4 4Gb (512Mx8) 2400/2666 (USD)**
- **Spot Avg Price: DRAM: DDR3 4Gb 512Mx8 1600MHz (USD)**
- **Spot Avg Price: DRAM: DDR3 2Gb 128Mx16 1600/1866 (USD)**

Dataset	Scale
GIST1M	1,000,000 (1M)
OpenAI-3072	1,000,000 (1M)
MovieLens-10M	69,000 (69K)
SIFT1B	1,000,000,000 (1B)
SPACEV-1B	1,400,000,000 (1.4B)
LAION-5B	**5,000,000,000 (5B)**

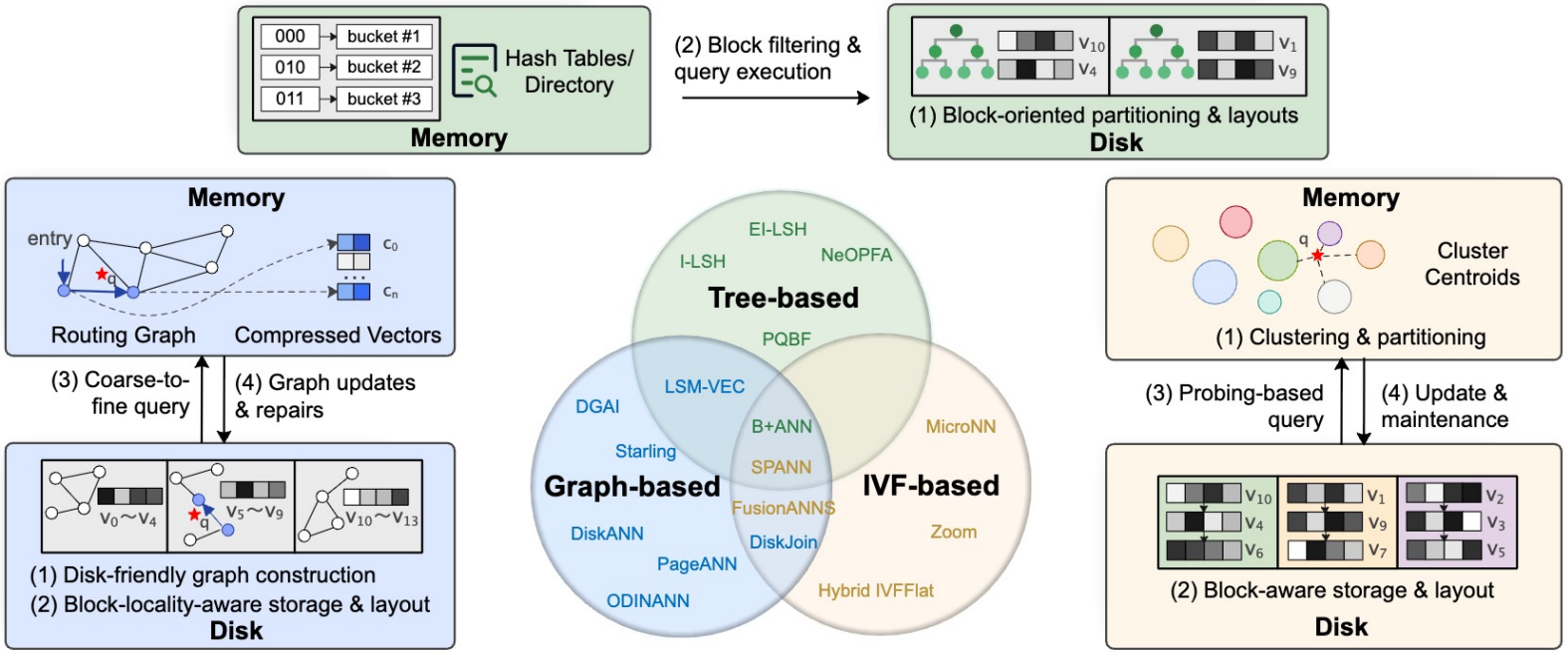
Rapidly growing data volume

Memory-Resident VS to Memory-SSD VS



Method Categories

Based on the primary index used for filtering, we categorize existing methods into: IVF-based, graph-based, and tree-based methods



The following materials can be found in our survey and experimental study:

[1] Y. Song, H. Li, Z. Wu, L. Yi, B. Zhu, X. Zhou, X. Huang, and J. Xu. ***Disk-Resident Vector Similarity Search: A Survey***, 2026.

[2] X. Chen, J. Qu, Y. Song, S. Lu, H. Li, M. Jiang, W. Zhou, J. Xu, X. Zhou, and F. Wu. ***Disk-Resident Graph ANN Search: An Experimental Evaluation***, 2026.

Overview of IVF-based Methods

Q1 (Partitioning)

How to achieve memory-efficient clustering and balanced partitions?

Q2 (Storage)

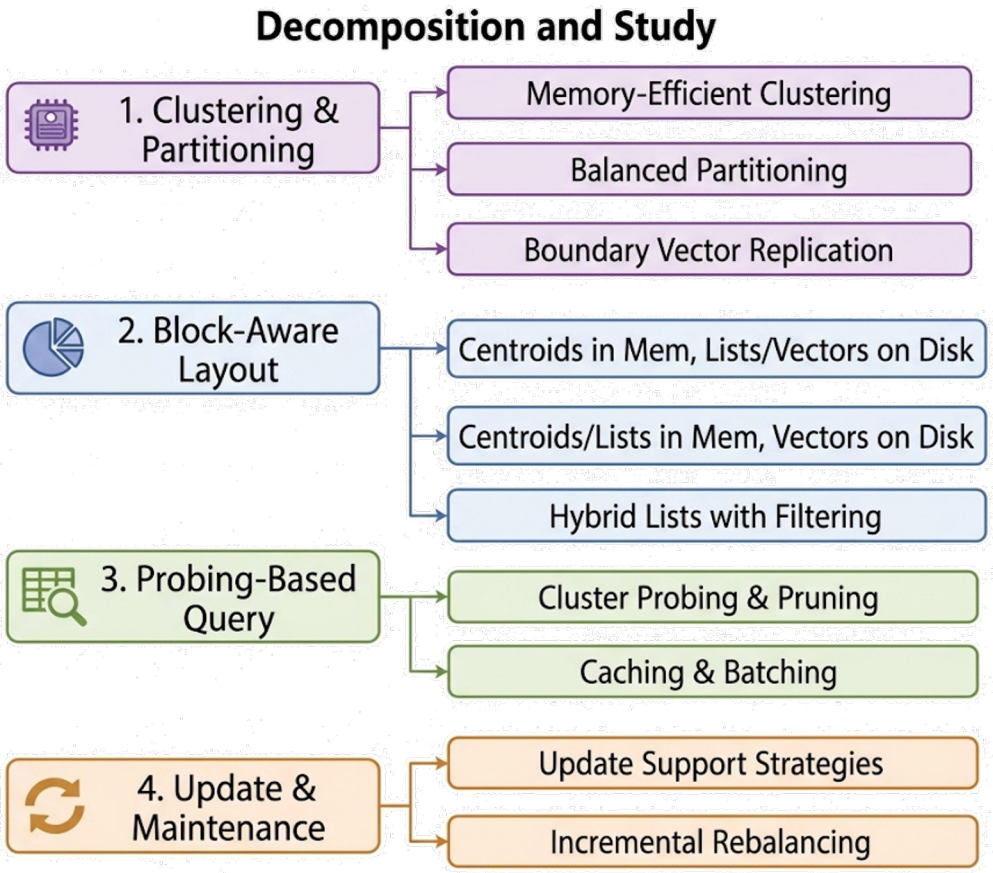
How to optimize disk I/O for page-aligned vector search?

Q3 (Querying)

How to balance search accuracy and latency ?

Q4 (Update)

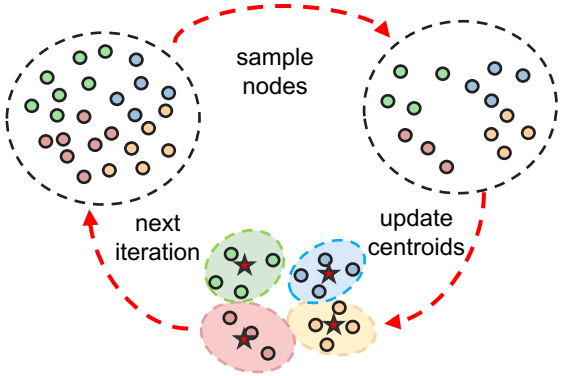
How to maintain index quality with incremental data updates?



Clustering and Partitioning

Limitations and Solutions

L1: Clustering is memory-intensive

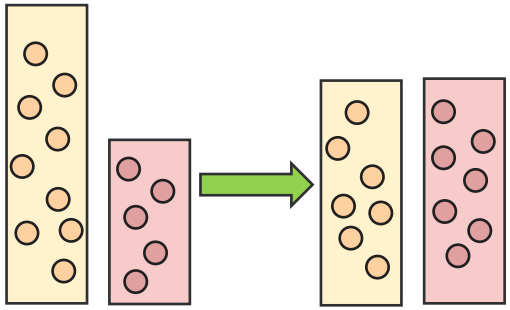


Mini-Batch k-Means

(e.g., MicroNN, Hybrid IVFFlat)

Streaming small data batches from disk to incrementally train centroids

L2: Uneven cluster sizes cause imbalanced I/O

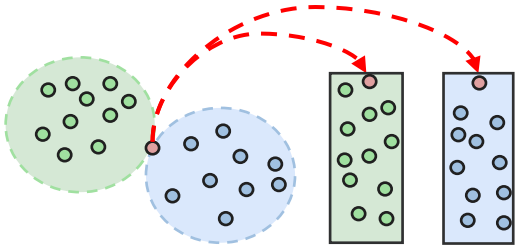


Balanced Clustering

(e.g., SPANN, SPFresh)

Adding size penalties to discourage oversized lists and ensure even partitions

L3: Single-cluster assignment leads to recall loss



Boundary Vector Replication

(e.g., SPANN, FusionANNS)

Storing boundary vectors in multiple nearby lists

Block-Aware Storage and Layout

An IVF-based disk-resident VS method typically maintains three core components: **cluster centroids**, **inverted lists**, and **raw vectors**.

Method	Memory	SSD
SPANN / SPFresh	IVF centroids + routing graph	IVF lists + vectors
MicroNN	IVF centroids	IVF lists + vectors
Zoom	Compact vectors + IVF centroids + IVF lists	Vectors
Hybrid IVFFlat	IVF centroids + Filter structures	IVF lists + vectors
FusionANNS	Compact vectors + IVF centroids + IVF lists	Vectors

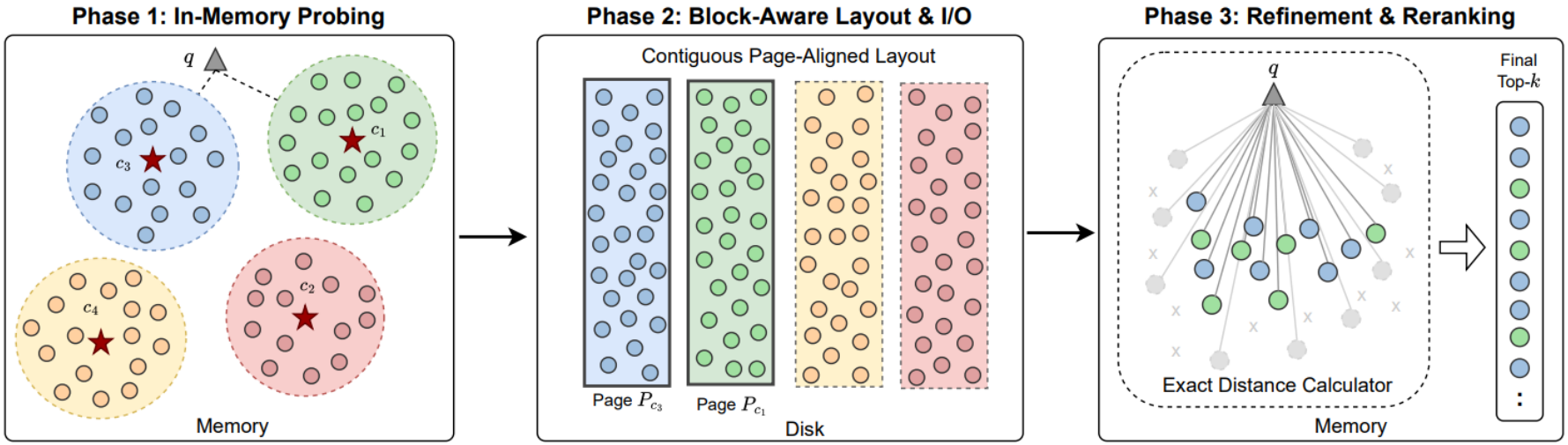
1. Centroids in Memory, Inverted Lists and Raw Vectors on Disk

- Lower memory overhead
- Unable to use ID information in the inverted list for pruning

2. Centroids and Inverted Lists in Memory, Raw Vectors on Disk

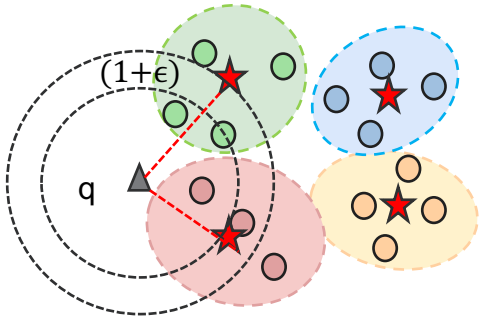
- Higher memory overhead
- Combining other indexes (e.g., PQ) for more effective pruning

Probing-Based Query Strategies



Cluster Probing and Dynamic Pruning

- Lightweight in-memory graph used for finding nearest clusters
- Query-aware dynamic pruning



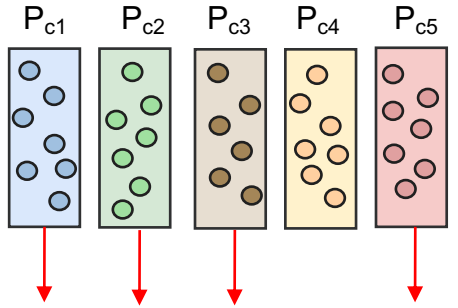
Only scan whose distance to the query vector is within a $(1+\epsilon)$ factor of the closest centroid

Caching and Batching

- Cache hot lists to avoid repeated read
- Group batch queries to read each page only once

q1: load P_{c1} P_{c2}
q2: load P_{c3} P_{c5}
q3: load P_{c2} P_{c5}

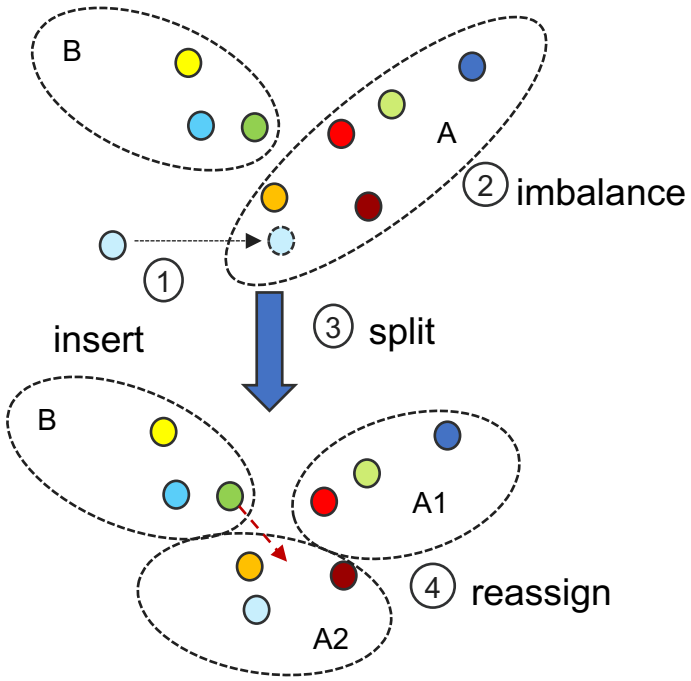
P_{c2} P_{c5} only need load once



Update and Maintenance Mechanisms

Update Support Strategies

- **In-Place Updates with Incremental Rebalancing** (e.g., SPFresh)



1. Insert or delete the nodes
2. Monitor inverted list sizes and detect imbalance
3. Oversize lists are split and undersize lists are merged
4. Vectors are locally reassigned to nearest clusters

- **Out-of-Place Updates** (e.g., MicroNN)

  Leveraging DBMS transactions for atomic inserts/deletes

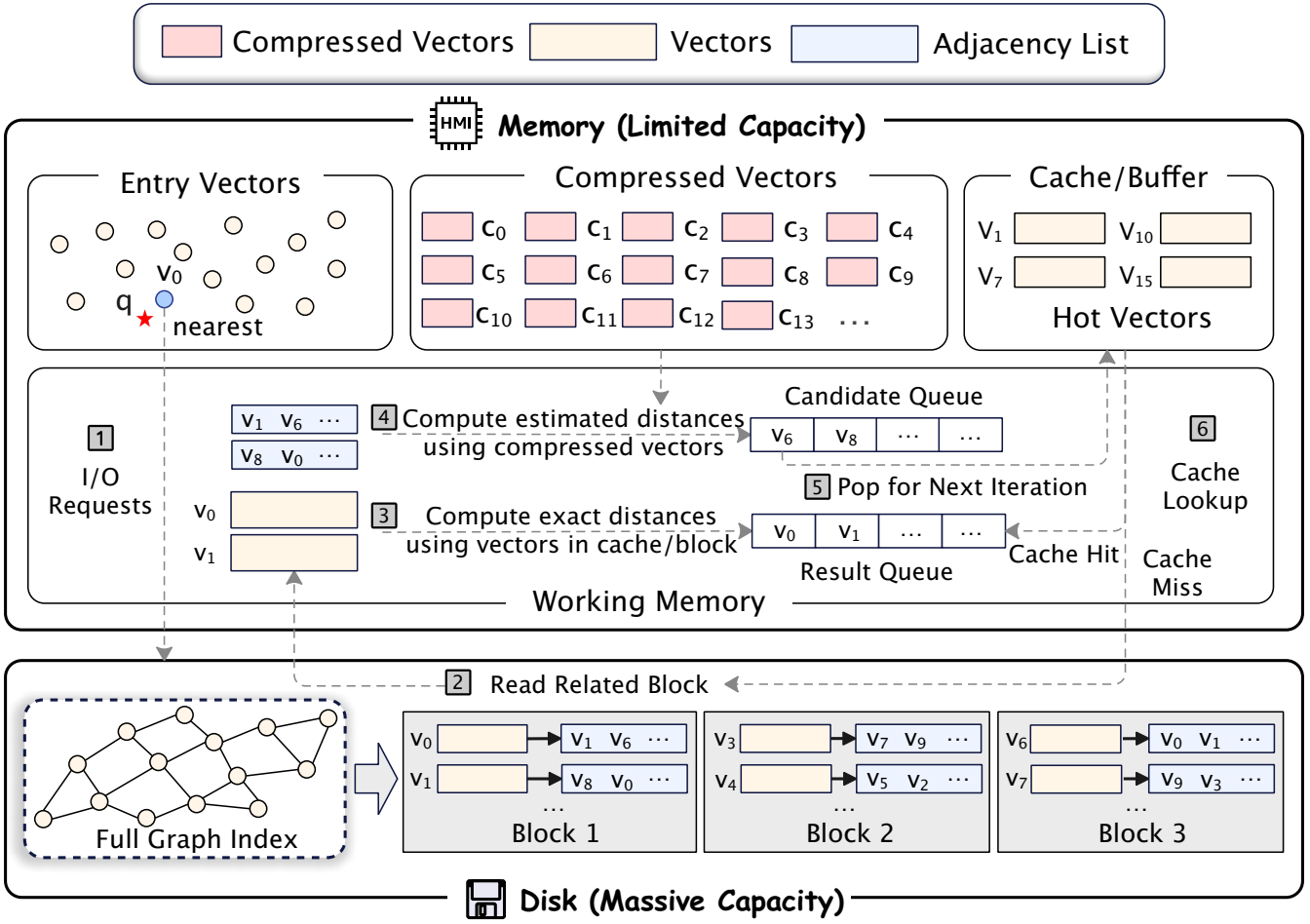
Overview Of Graph-based VS

Memory

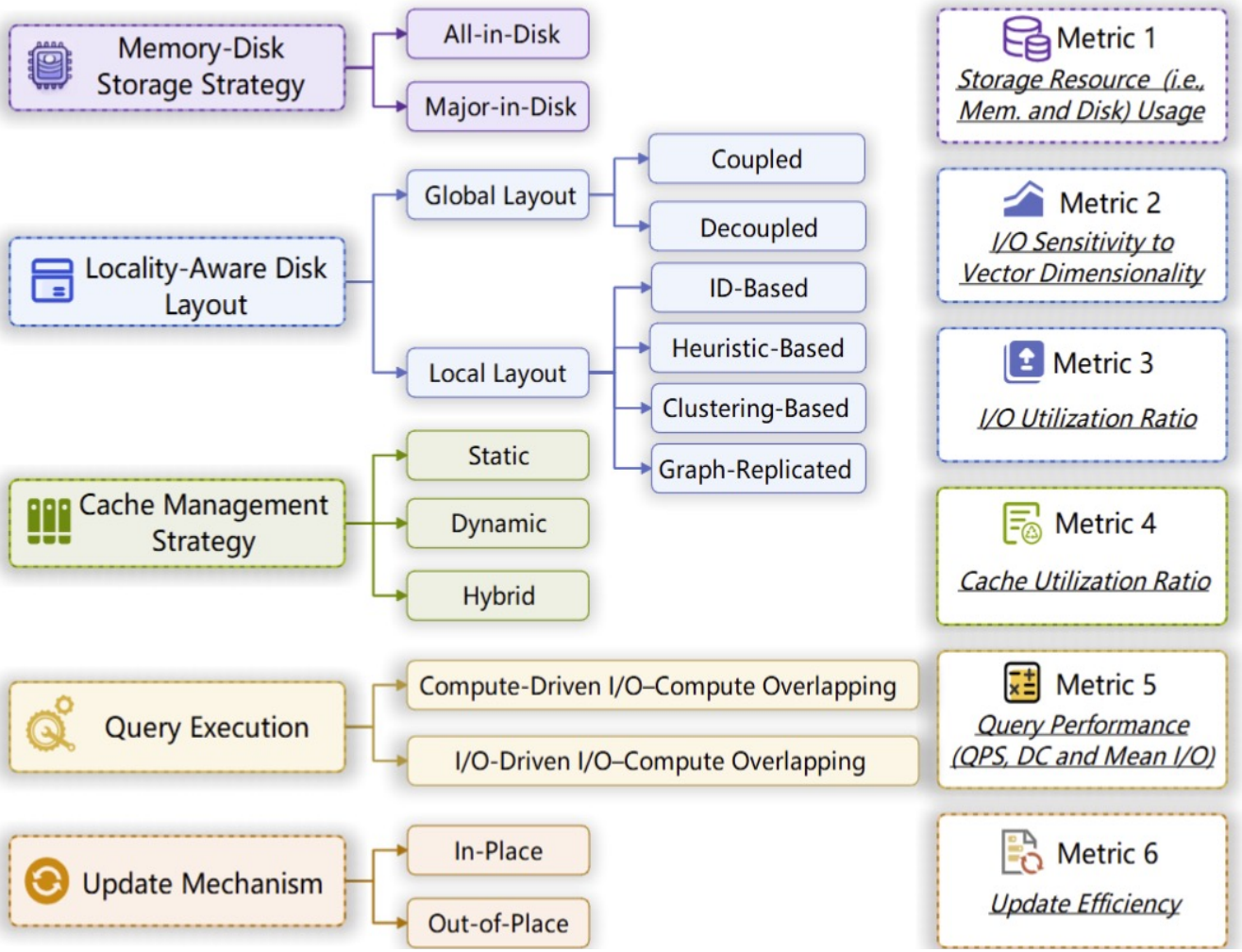
- **Compressed Vectors** for fast distance estimation
- **Entry Vectors** for traversal initialization
- **Fixed-Size Buffer** caching frequently accessed blocks

Disk (SSD)

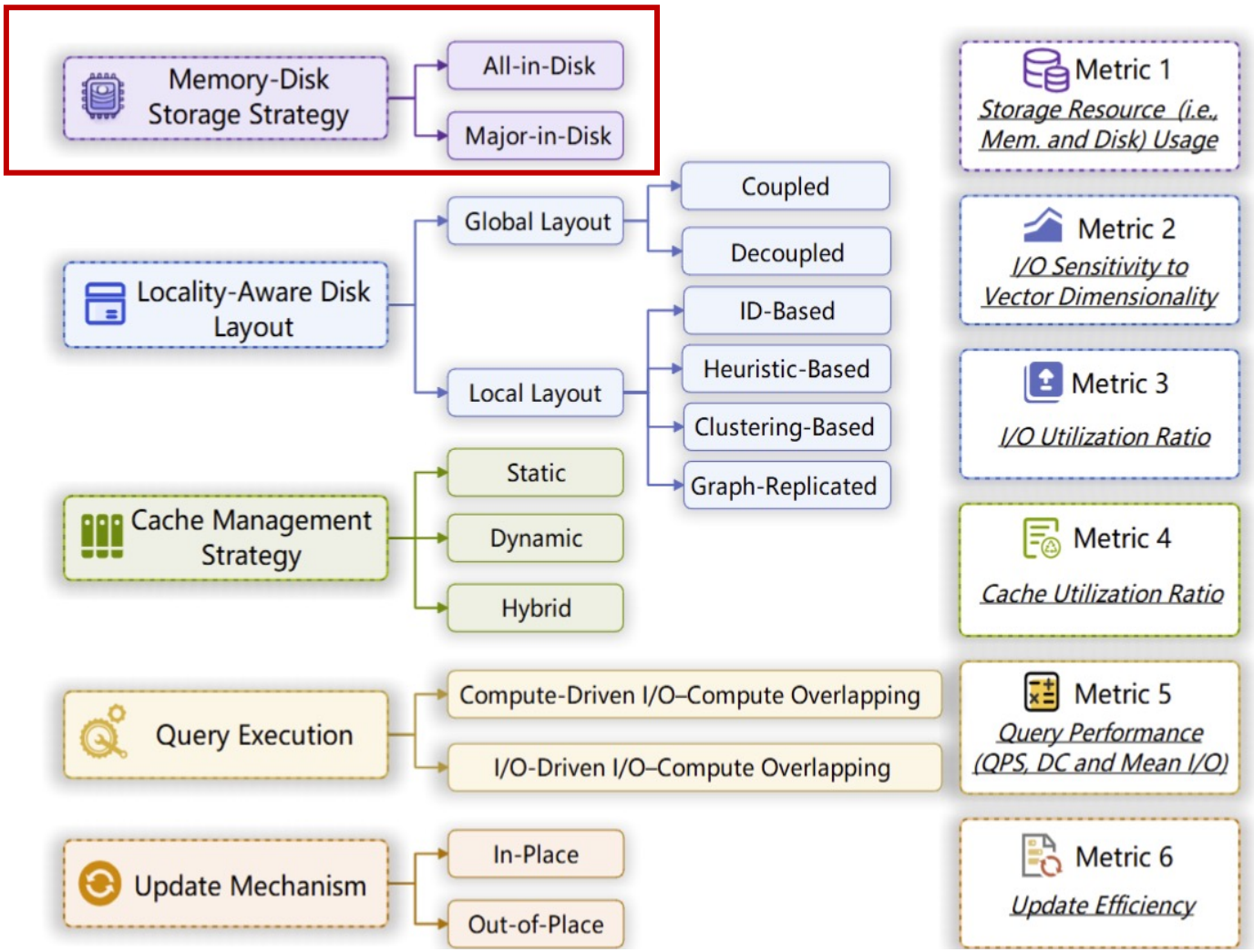
- **Graph Adjacency Lists**
- **Raw vectors**



Technology Decomposition

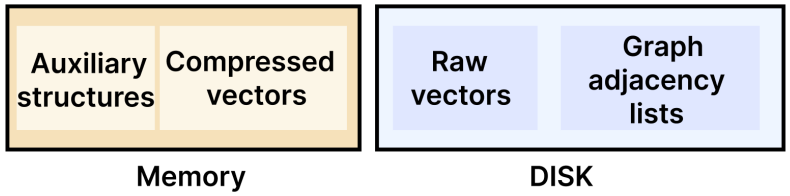


Technology Decomposition



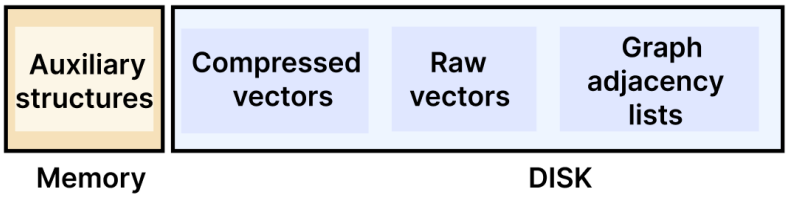
Memory-Disk Storage Strategy

Major-In-Disk



Most widely adopted strategy: Fast with fewer I/Os, but **memory overhead scales with data size**

All-in-Disk



Suitable for **memory-limited environments**, but incurs higher disk access and **poor latency**

Table 2: Memory–disk storage strategies. *Legend:* ▷ = Compressed Entry Vectors, ▲ = Compressed Vectors, ■ = Sampled Entry Graph, ★ = Cache, ● = Adjacency Lists, and ⧫ = Vectors.

Type	Method	Memory	Disk	Applications
Major-in-Disk★	DiskANN [12]	▲ ★	● ⧫	(1) Low-latency/High-QPS ANN services. (2) Deployments with relatively sufficient memory budget.
	Starling [35]	▲ ■ ★	● ⧫	
	Gorgeous [38]	▲ ■ ★	● ⧫	
	MARGO [39]	▲ ■ ★	● ⧫	
	BAMG [15]	▲ ■	● ⧫	
	XN-Graph [40]	▲ ■	● ⧫	
	DGAI [20]	▲ ★	● ⧫	
	PipeANN [8]	▲ ■	● ⧫	
	PageANN [14]	▲† ■ ★	▲† ● ⧫	
All-in-Disk◇	AiSAQ [32]	▷	▲ ● ⧫	(1) Memory-tight deployments (edge/on-device). (2) Scale-sensitive applications where dataset-scale ▲ cannot fit in memory.
	LM-DiskANN [25]	★	▲ ● ⧫	
	LSM-VEC [42]	■ ★	● ⧫	

★ **Major-in-Disk**: compressed vectors in memory; adjacency lists and vectors on disk.
◇ **All-in-Disk** (memory is still utilized): only auxiliary data in memory; all scale-growing components (i.e., compressed/raw vectors, and adjacency lists) on disk.
† Budget-dependent placement.

Storage Strategy Evaluation

Table 4: Disk (GB) and Memory (MB) Usage Across Datasets (Min/Max highlighted *per dataset* across all methods).

Type	Method	GloVe		SIFT-1M		SIFT-100M		Deep		Tiny		MSong		GIST		OpenAI	
		Disk	Mem	Disk	Mem	Disk	Mem	Disk	Mem	Disk	Mem	Disk	Mem	Disk	Mem	Disk	Mem
Major-in-Disk	DiskANN	0.790	160	0.811	108	31.908	2859	1.368	175	9.681	530	2.052	279	4.174	574	16.399	933
	Starling	0.790	123	0.811	120	31.908	3819	1.368	196	9.681	595	2.052	306	4.174	628	16.399	1039
	Gorgeous	5.493	138	4.768	109	413.259	3056	5.277	198	29.564	445	5.878	193	–	–	30.708	818
	PipeANN	0.752	153	0.763	125	31.789	2547	1.272	155	9.537	430	1.896	194	3.815	274	15.259	749.5
	DGAI	0.693	114	0.684	82	30.990	4817	1.144	124	10.494	650	2.090	201	4.001	366	16.230	1018
	PageANN†	0.613	102	0.605	61.5	21.202	2223	1.367	88.5	9.684	298.5	2.053	100.5	4.186	169.5	16.529	456
All-in-Disk	AiSAQ	1.570	33	1.335	30	54.496	28.5	2.033	31.5	9.905	33	2.118	31.5	4.286	36	16.408	48
	PageANN‡	0.514	43.5	0.558	30	63.580	25.5	1.270	30	9.540	28.5	1.900	34.5	–	–	15.260	45

Notes: blue = minimum, red = maximum **within each dataset and metric** (Disk or Mem) across all methods; ties are highlighted; missing entries (–) are excluded.

† PageANN major-in-disk version.

‡ PageANN all-in-disk (low-memory) version.

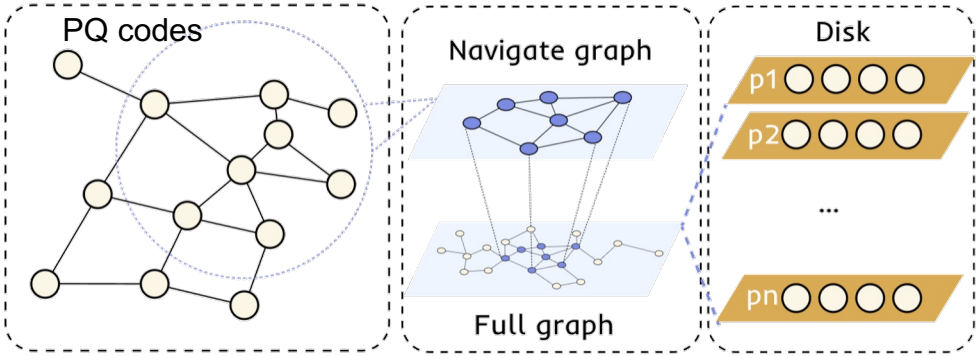
Memory Usage: Major-In-Disk uses much more memory, while **All-In-Disk** keeps a **Small**, constant footprint (25.5–48 MB)

$$M_{total} = M_{nav} + M_{pq} + M_{cache} + C$$

If M_{total} is less than the memory buget, we adopt the Major-In-Disk strategy; otherwise, we switch to All-In-Disk strategy.

Index Construction Cost

The total index construction time can be broken down into three main stages:



1. Navigation and Full Graph Building
2. PQ Training
3. Graph Partitioning and Layout

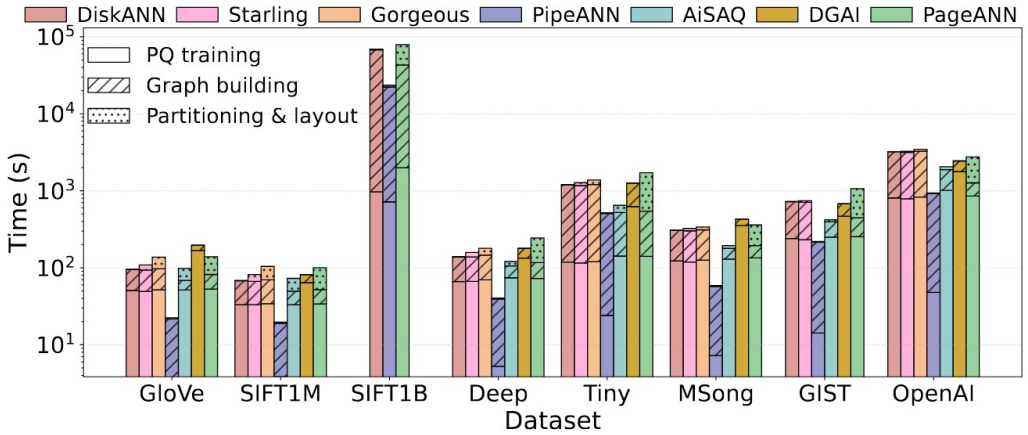
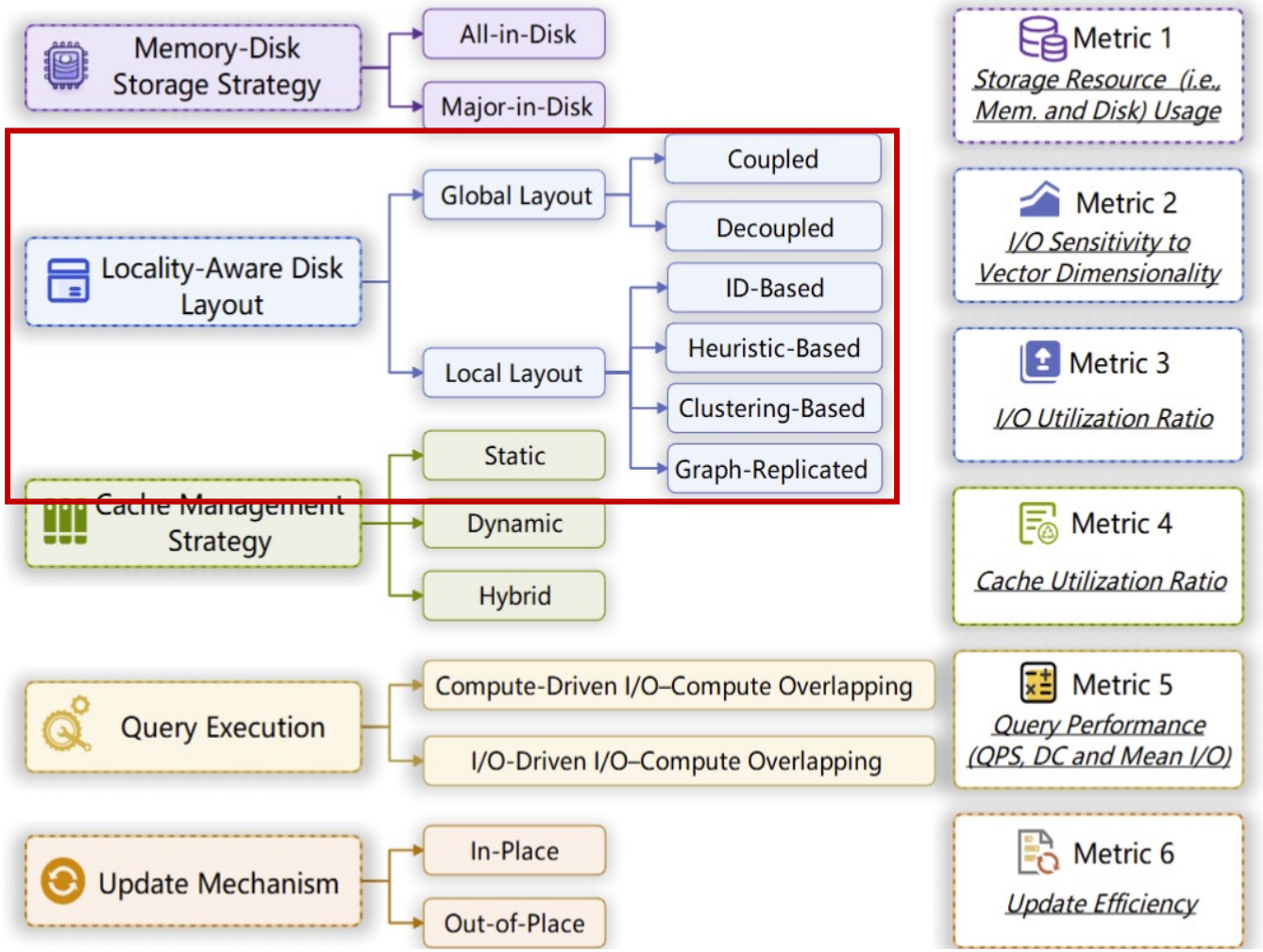


Figure 8: Index construction time.

Finding: (1) For most methods, index construction time is dominated by **PQ training** and **graph building**. (2) Several methods fail to construct on billion-scale datasets due to **excessive memory overhead** and **prohibitive construction times**.

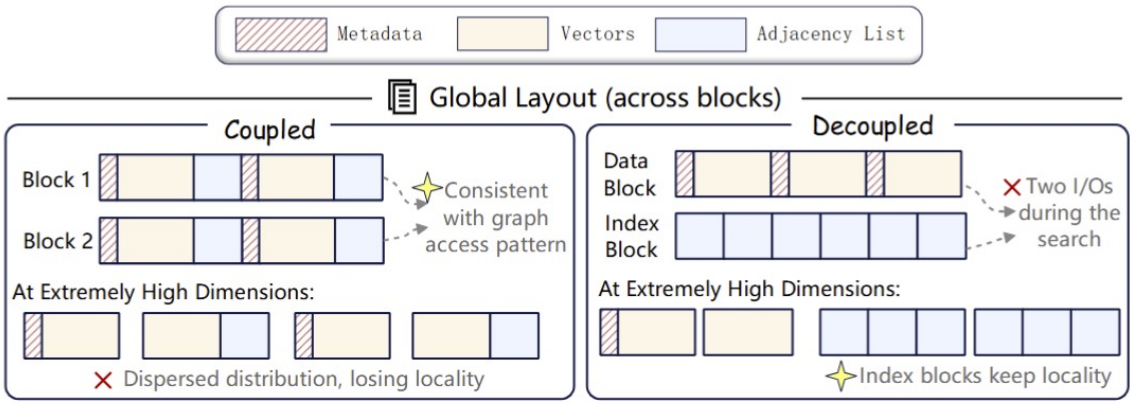
Technology Decomposition



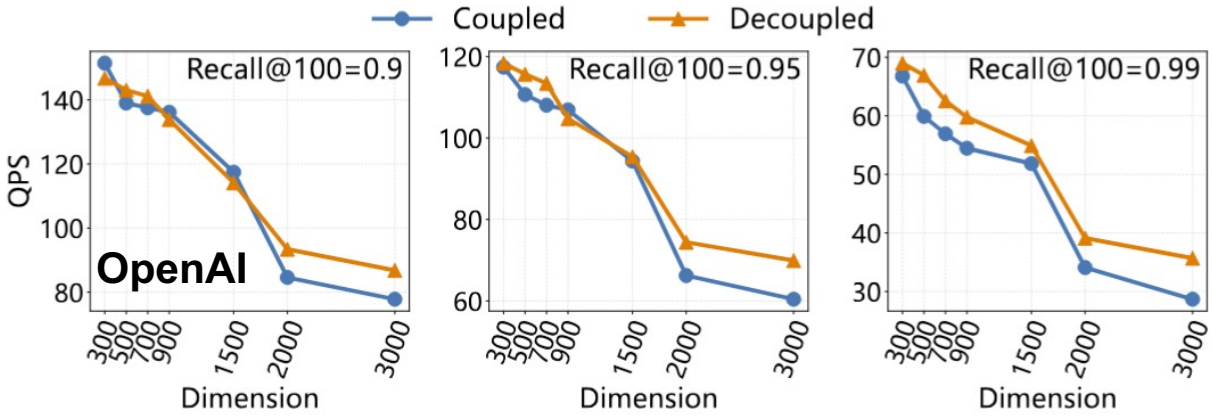
Locality-Aware Disk Layout Strategies

Global Layout

- **Coupled layout:** vector + adjacency lists in one block
- **Decoupled layout:** stores adjacency lists + vectors in separate blocks



Global Layout Evaluation

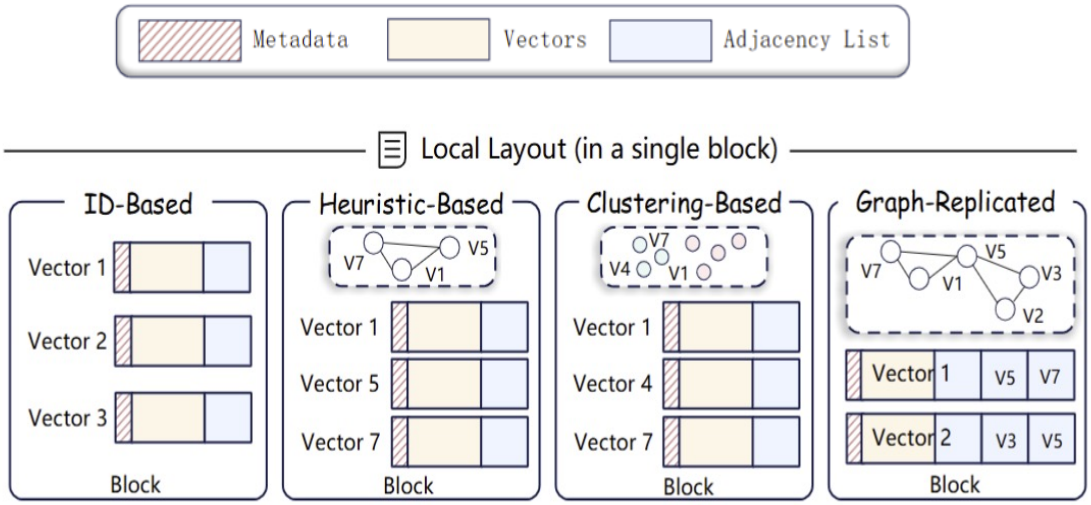


Finding: Coupled layouts perform better in low-dimensional settings, while decoupled layouts are more effective at higher dimensions and recall levels

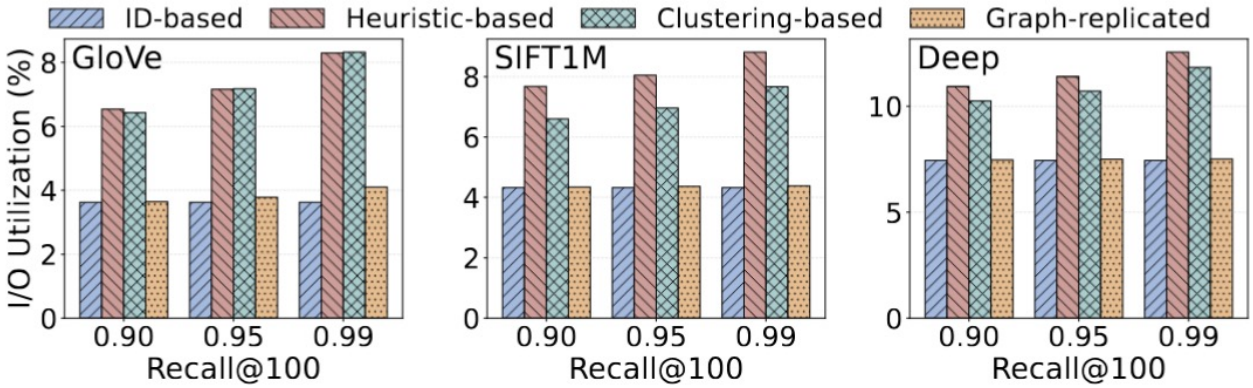
Locality-Aware Disk Layout Strategies

Local Layout

(1) ID-Based : Organize data by ID
(2) Heuristic-Based : Use graph topology to colocate neighbors
(3) Clustering-Based : Groups similar vectors
(4) Graph-Replicated : Replicates adjacency lists



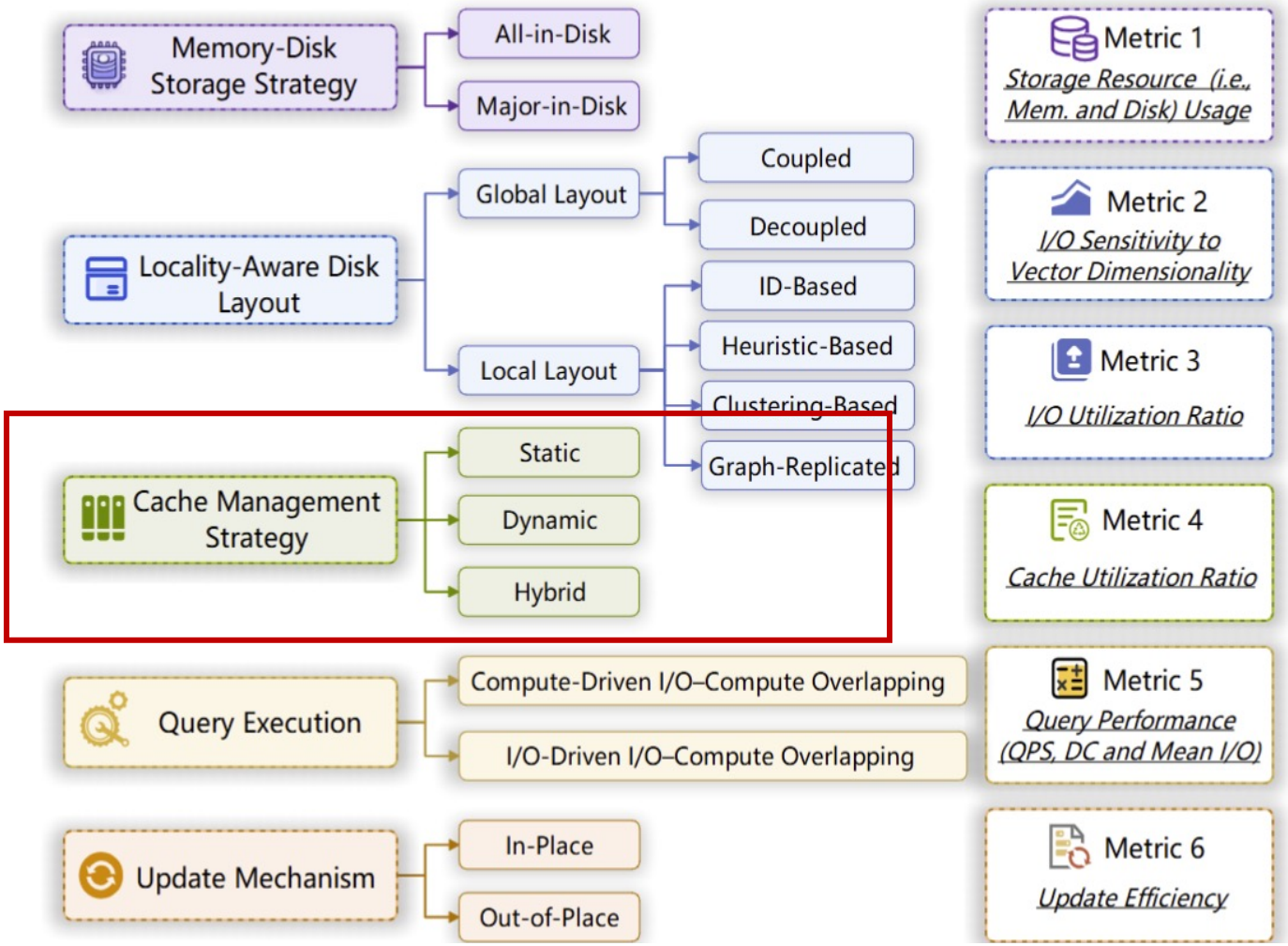
Local Layout Evaluation



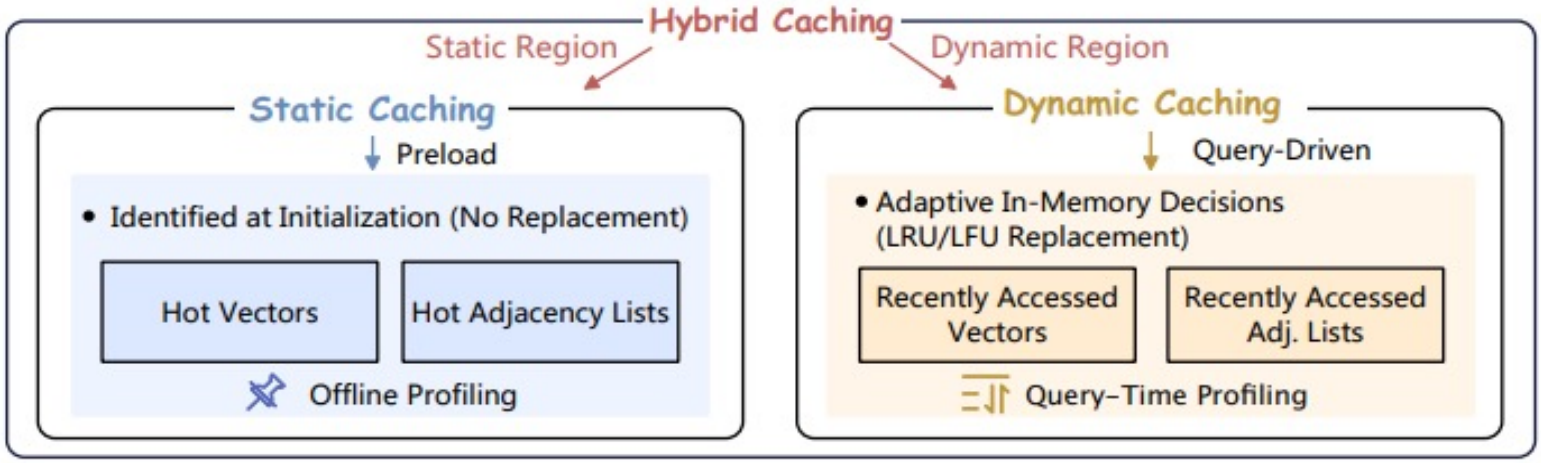
Finding 1: Overall I/O utilization remains **low** across all methods, not exceeding 15%

Finding 2:
Heuristic-based layout achieves the highest I/O utilization, outperforming others by 1.91×,1.16×, and 1.83×, respectively

Technology Decomposition



Cache Management Strategy



- **Static Caching** (e.g., DiskANN, Starling, Gorgeous, PageANN)

  Retains pre-identified "hot data" in memory without replacement

- **Dynamic Caching** (e.g., XN-Graph)

  Adaptively caches data using online replacement policies (e.g., LRU/LFU) based on runtime access

- **Hybrid Caching** (e.g., Govector)

  Partitions memory into a static region for long-term hot data and a dynamic region for runtime access

Comparison of cache strategies

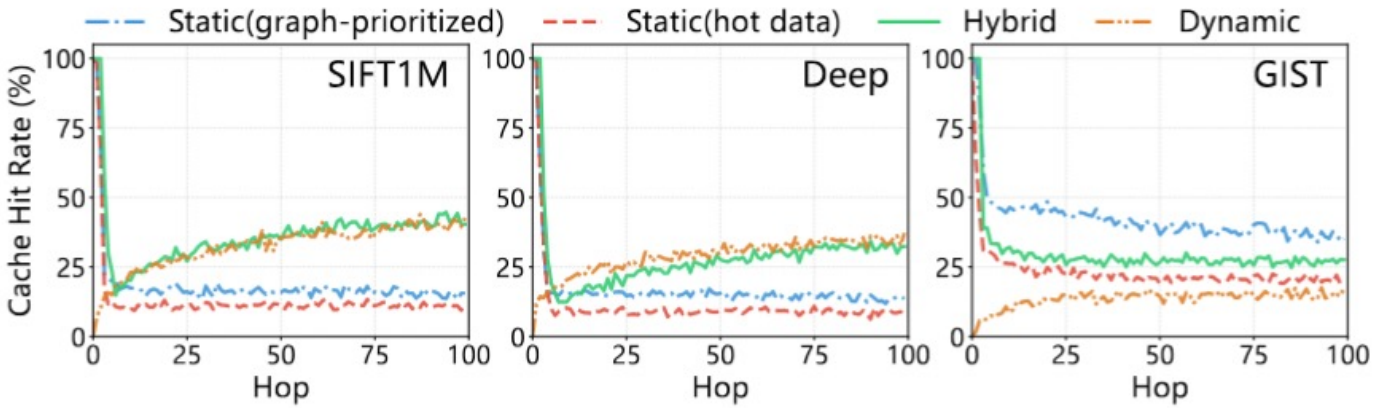
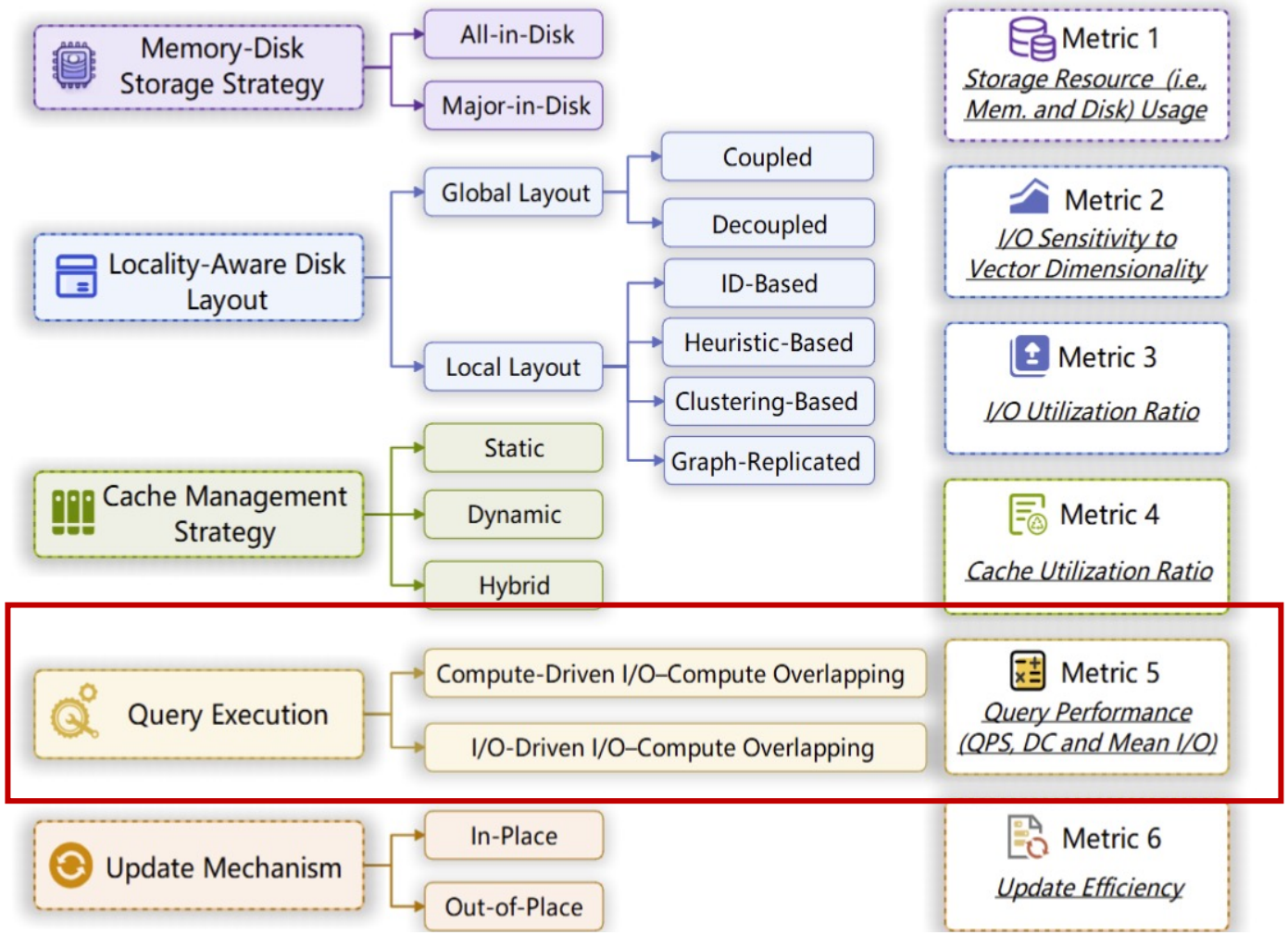


Figure 13: Cache hit rate vs. search hops.

- **Dynamic and Hybrid strategies** outperform static baselines by 2.6x to 2.75x on **low-dimensional datasets** (i.e., SIFT1M and Deep)
- **Static (graph-prioritized) strategy** achieves a hit rate 2.22x higher than dynamic baselines on **high-dimensional datasets** (i.e., GIST)

Finding: The optimal caching strategy depends on **dimensionality**. Dynamic and hybrid caching perform best in low dimensions, whereas static (graph-prioritized) is more effectivein high dimensions.

Technology Decomposition



Query Execution

- To mitigate high disk access latency, async execution overlaps I/O with CPU computation to reduce stall time.

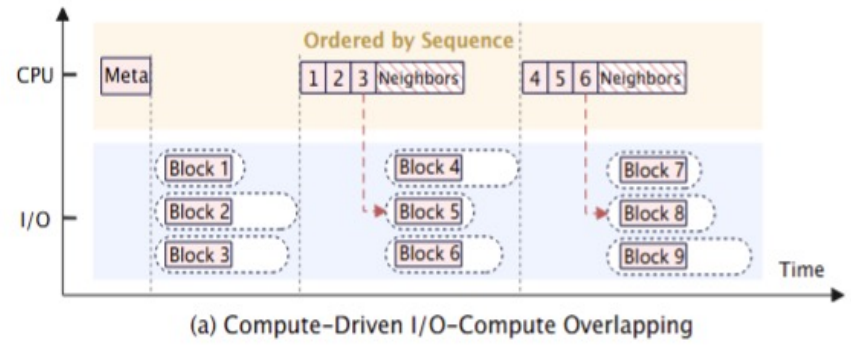


Figure 6: Compute-Driven vs. I/O-Driven Overlapping

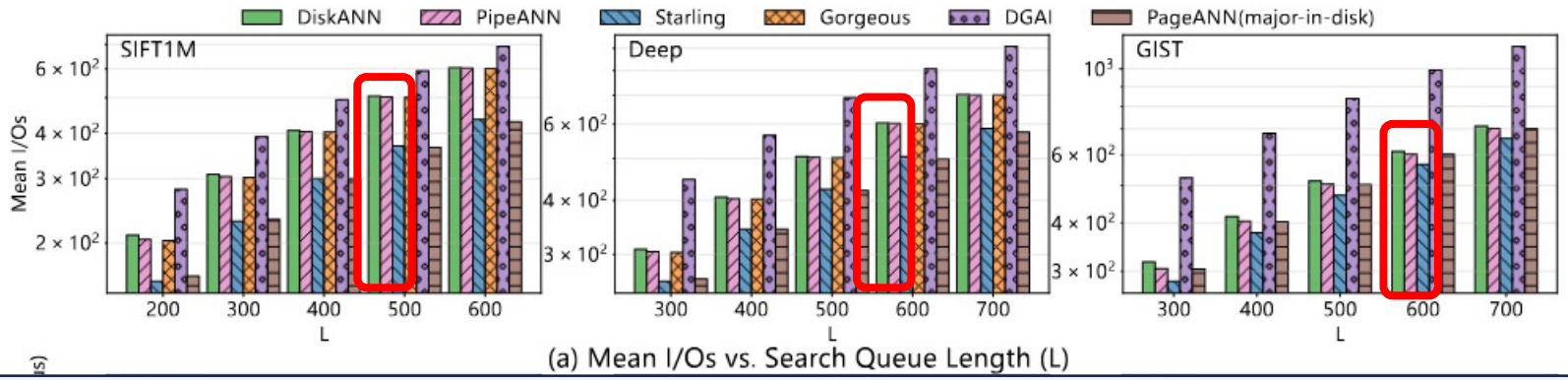
Compute-Driven Overlapping

Issues the next I/O request only after intermediate CPU computation determines which data blocks are required

I/O-Driven Overlapping

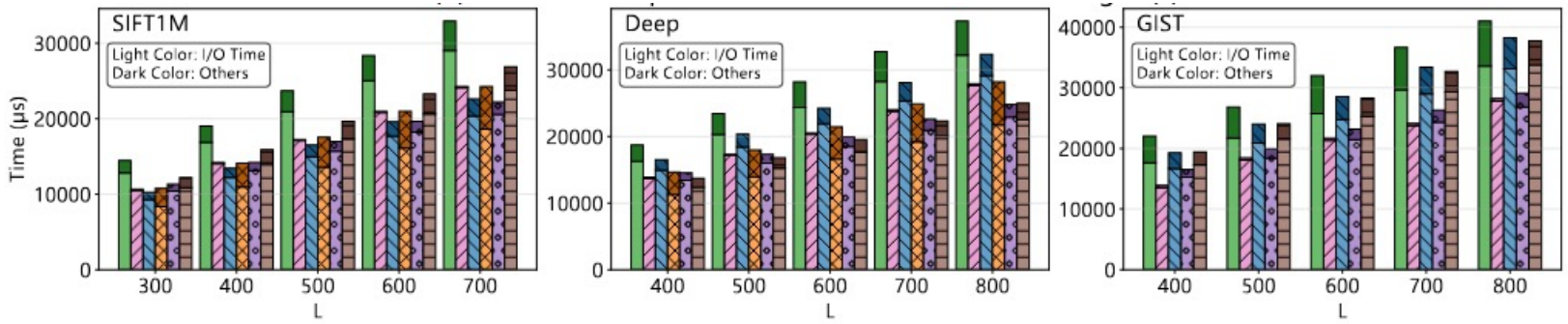
Proactively submits predicted I/O requests and reorders CPU computation according to the arrival order of data blocks.

I/O & Computation Evaluation



(a) Mean I/Os vs. Search Queue Length (L)

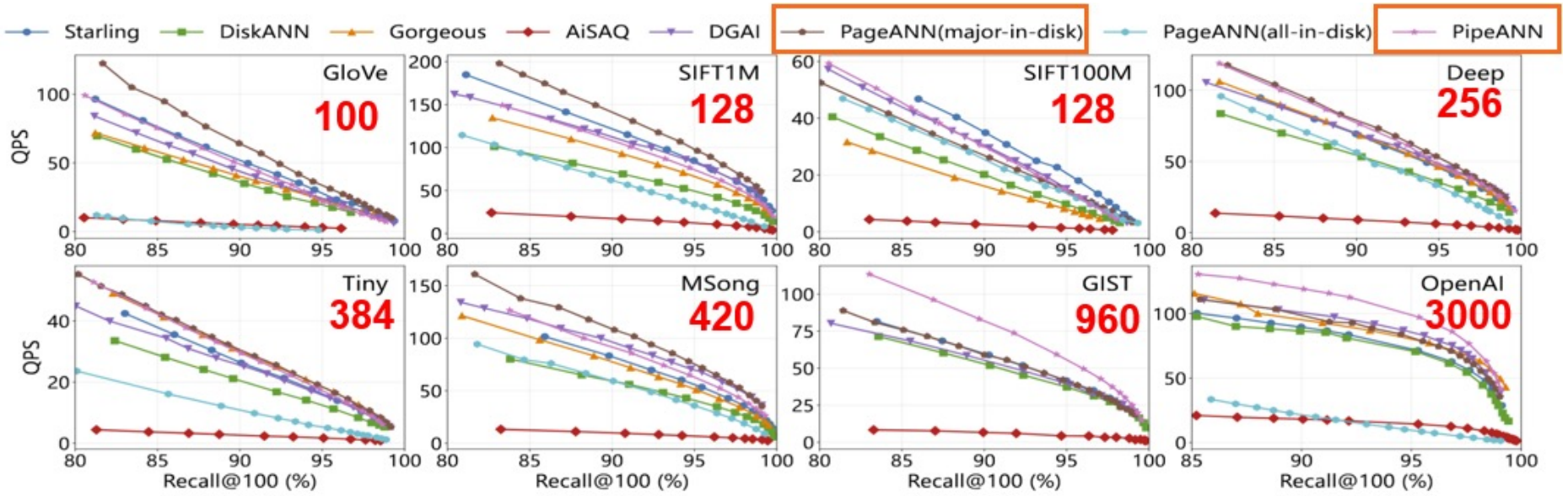
Finding 1: Under a consistent layout (i.e., DiskANN and PipeANN) , IO-driven overlapping incurs **no additional I/O** in a single-threaded setting



(c) I/O and Computation Time vs. Search Queue Length (L)

Finding 2: Async execution masks computation costs, making **I/O dominant in search latency**. However, as dimensionality increases, computation overhead becomes more significant

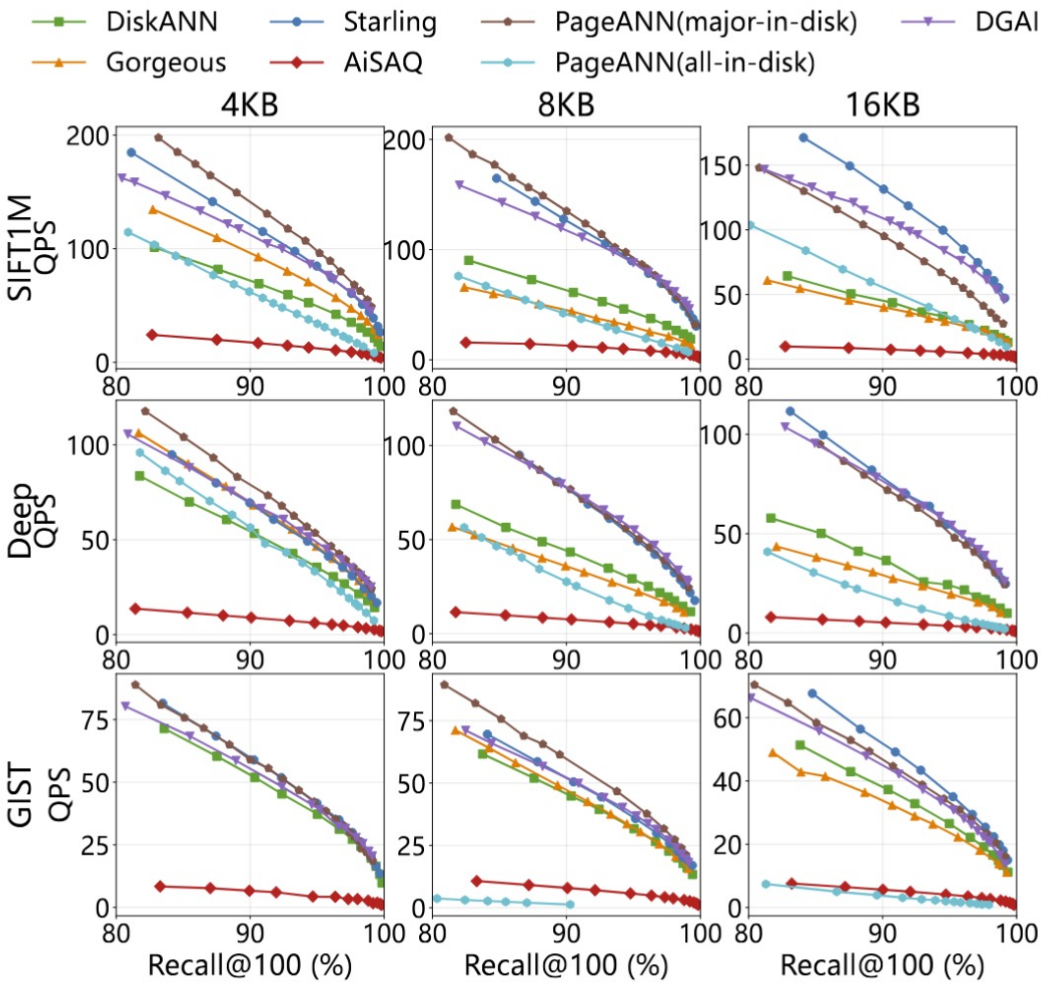
QPS vs. Recall Evaluation



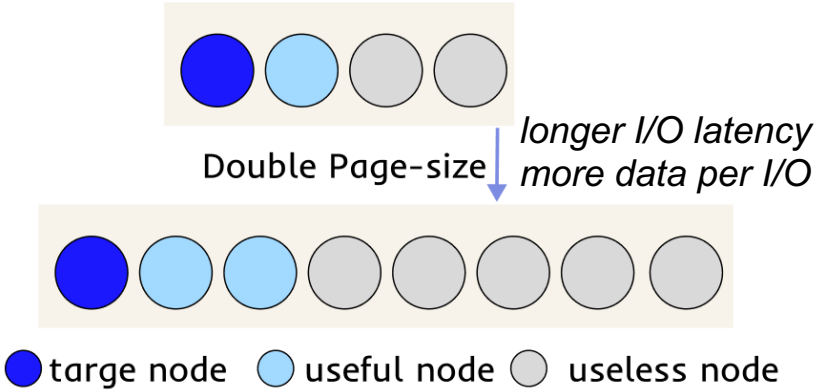
Finding: PageANN (major-in-disk) outperforms in low-to-medium dimensions, whereas PipeANN dominates in high dimensions

- At high-dimensions, **poor block locality** makes multi-block accesses inevitable, favoring **I/O-driven overlapping** to hide read latency
- At lower dimensions, **stronger block locality** makes **graph topology and layout** more critical for reducing block accesses

Impact of Page Size

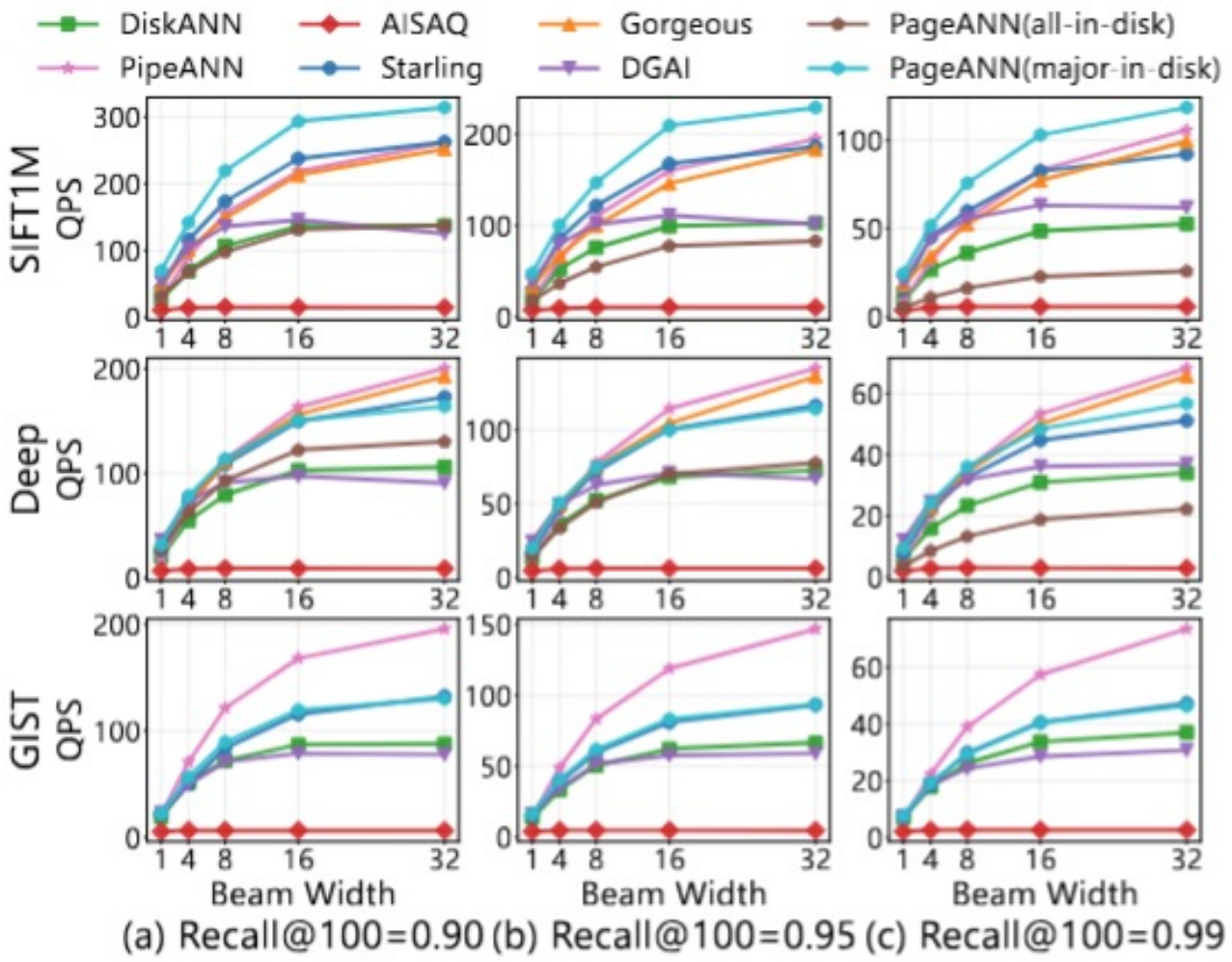


Finding: QPS generally **declines as page size increases** from 4 KB to 16 KB



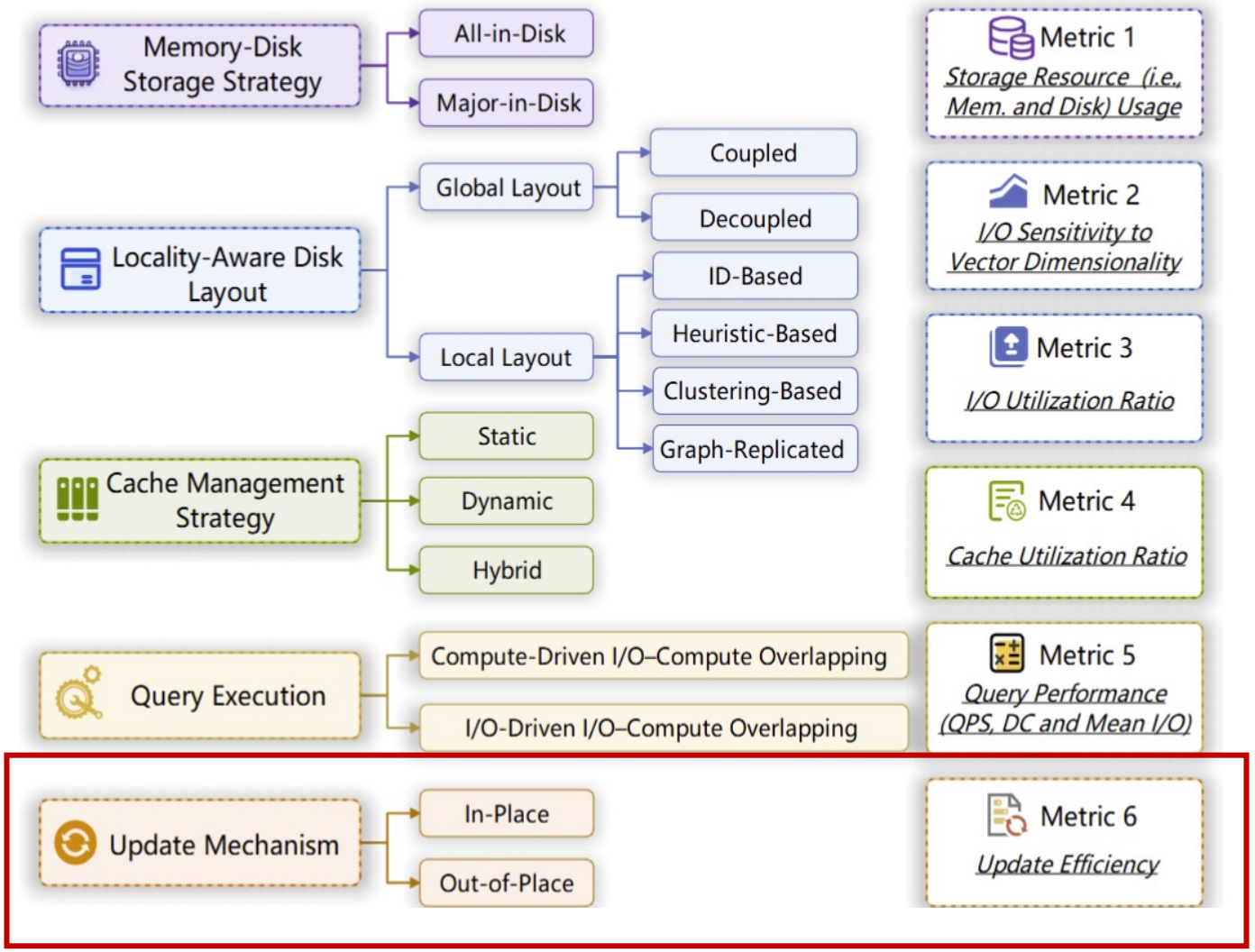
Without an effective layout strategy to exploit this extra data, most of it becomes redundant, introducing unnecessary I/O cost.

Impact of Beamwidth



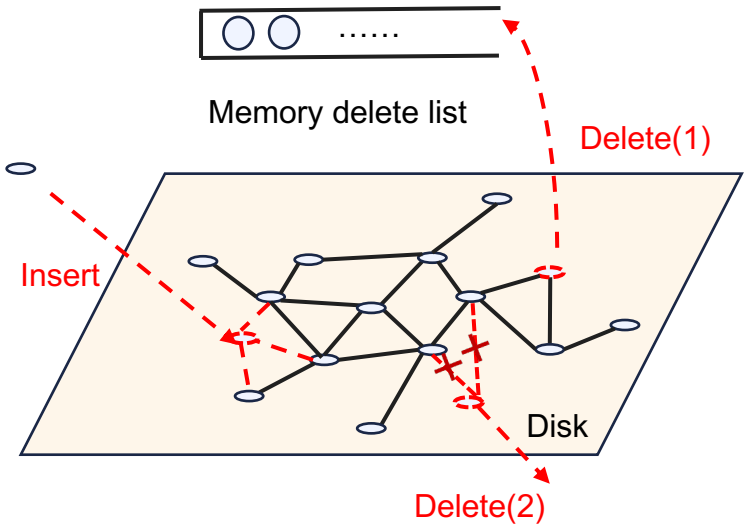
Finding: Increasing the beam width improves query performance, but the marginal gains progressively reduce, plateauing at around 16

Technology Decomposition



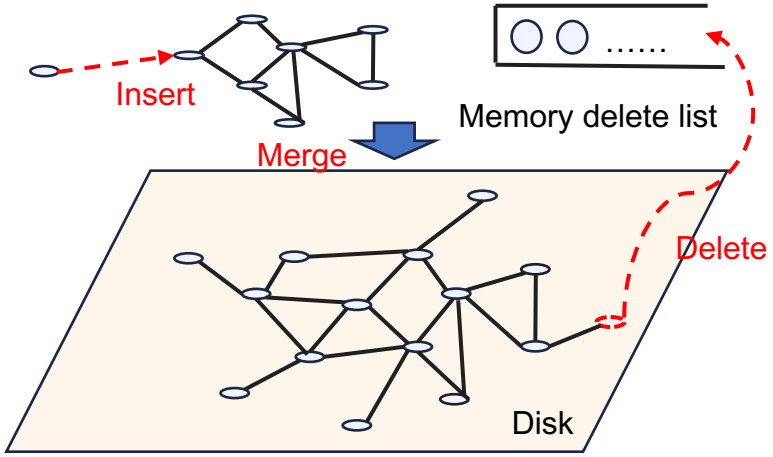
Update Mechanism

In-Place Update



- **Direct Modification:** edits on-disk adjacency list directly and maintains local graph topology.
- **Deferred Cleanup**: deletions are recorded immediately, while space recycling is handled later in the background.

Out-of-Place Update



- **Batched Merging:** updates are first buffered in memory and periodically merged into the disk index.
- **Memory Overlay:** new data is kept in memory for immediate query visibility, and later integrated into the disk index asynchronously.

Update Method Evaluation

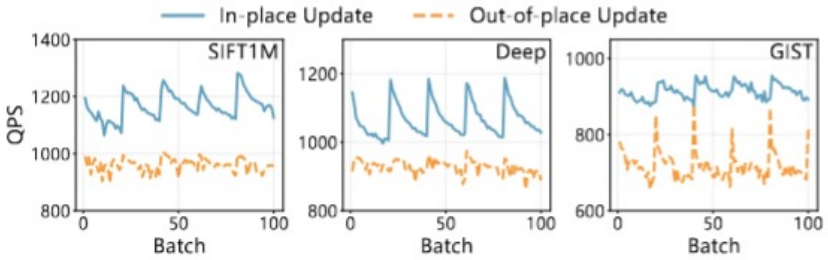


Figure 15: Search QPS under concurrent updates.

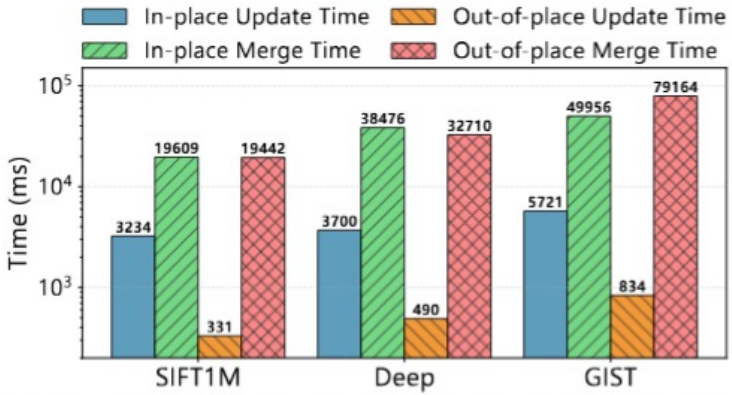


Figure 16: Average update and merge time (The merge time for in-place updates is consumed by the batch removal of nodes from the deletion queue).

- The in-place strategy sustains **1.15x – 1.27x** higher search throughput than out-of-place across all datasets.

- The out-of-place strategy achieves significantly lower update latency, ranging from **6.8x** to **9.7x** less than the in-place approach.

Finding: The in-place update favors query-heavy workloads by sustaining higher QPS, whereas the out-of-place update better suits update-heavy or balanced workloads due to its lower update latency.

Technique selection

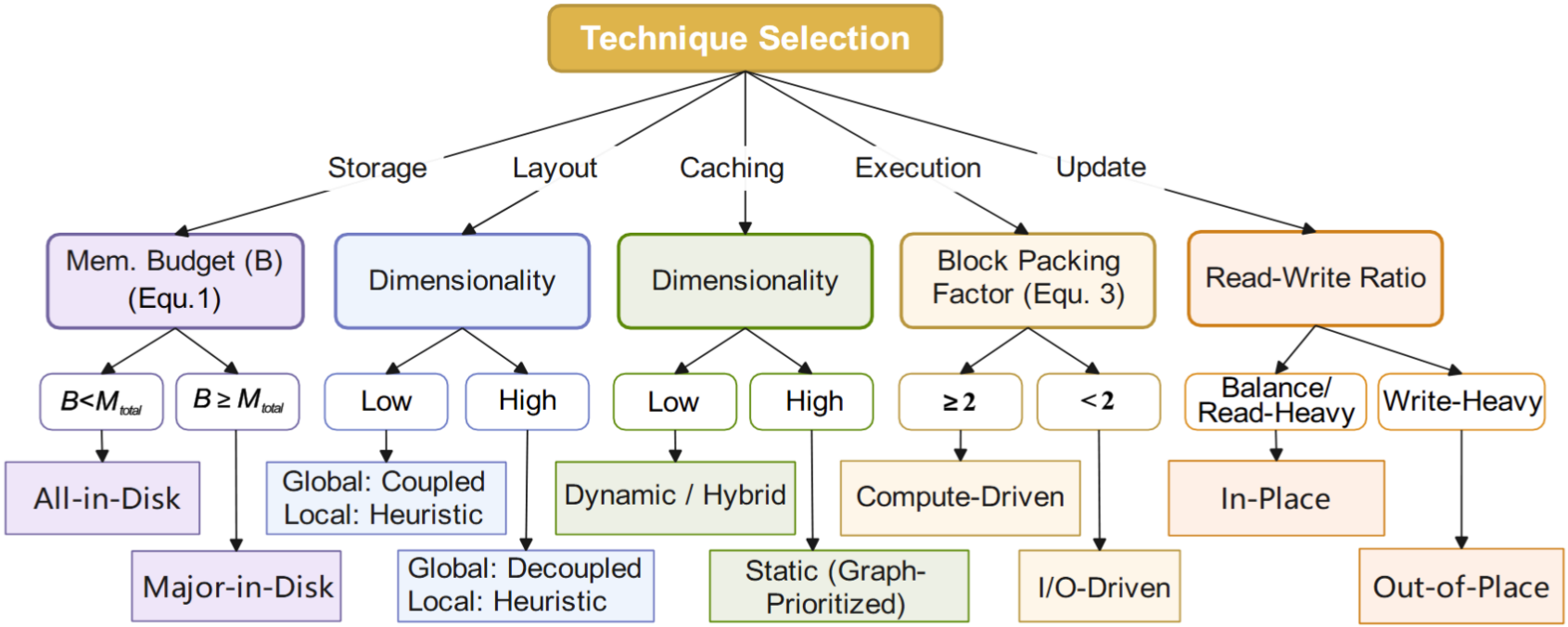
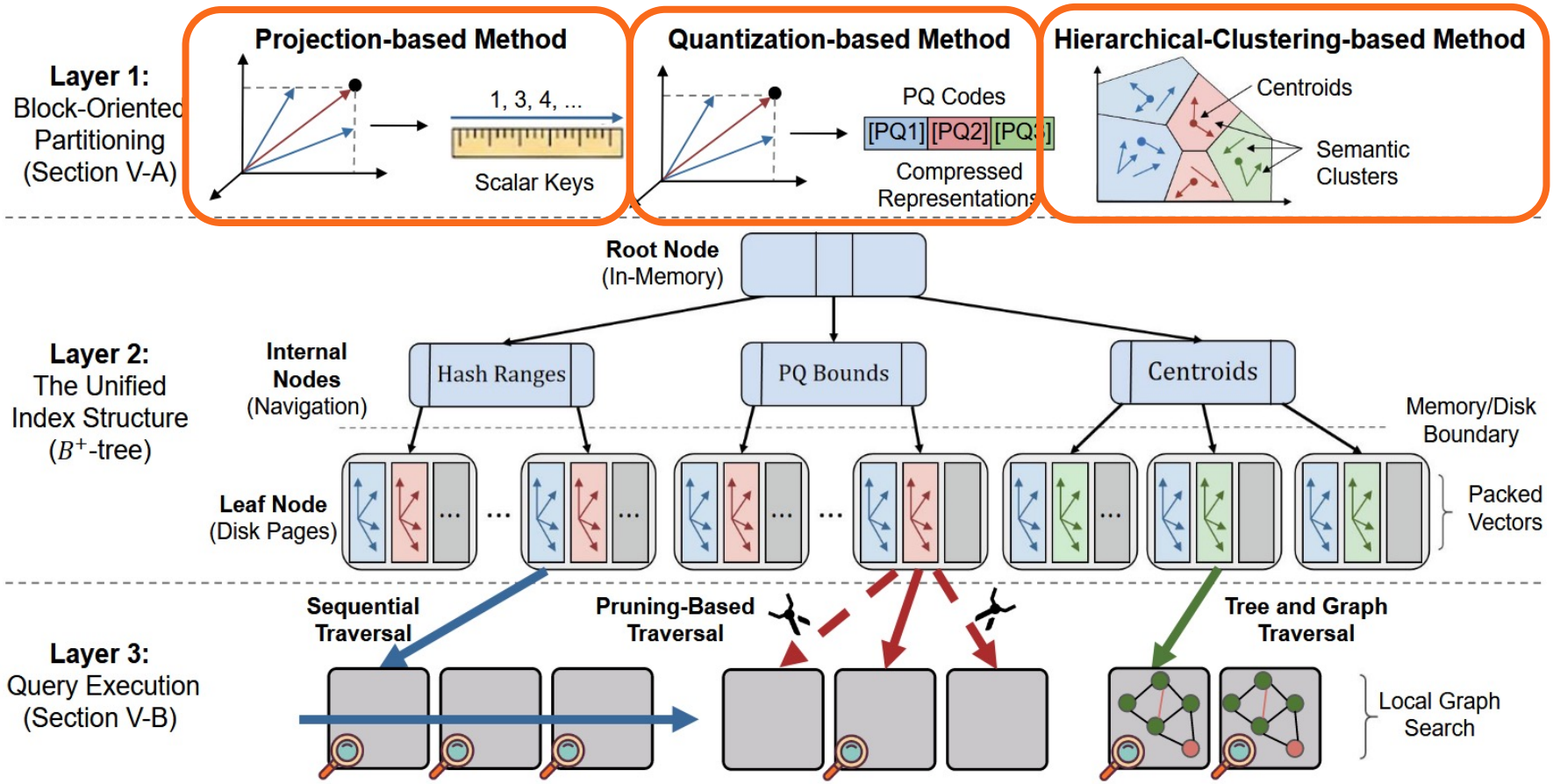


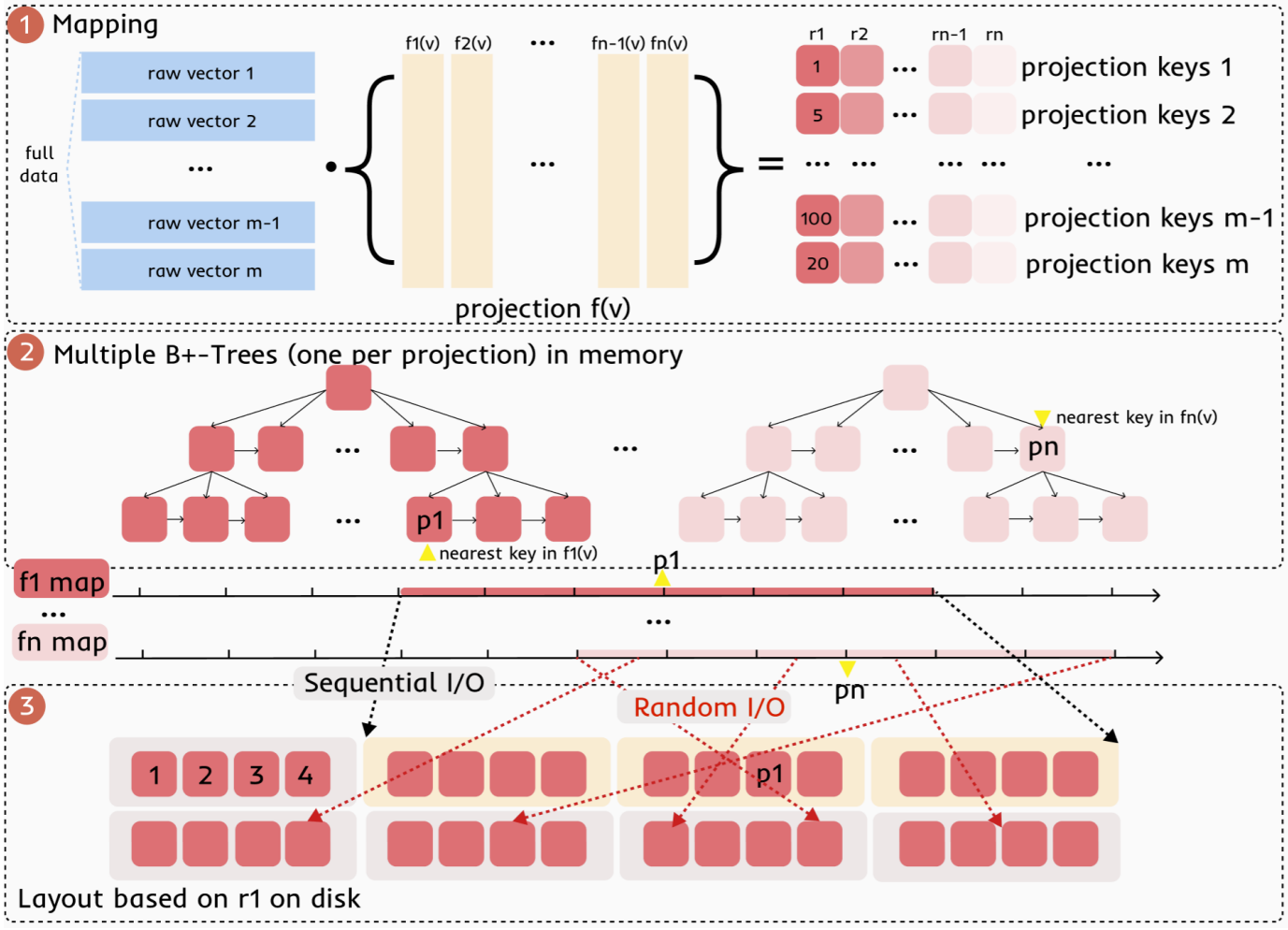
Figure 17: Technique Selection.

Overview Of Tree-based ANN

Tree-based methods first map vectors into orderable and low-dimensional representations and then organize them using tree-based structure to enable efficient pruning



Tree + Projection

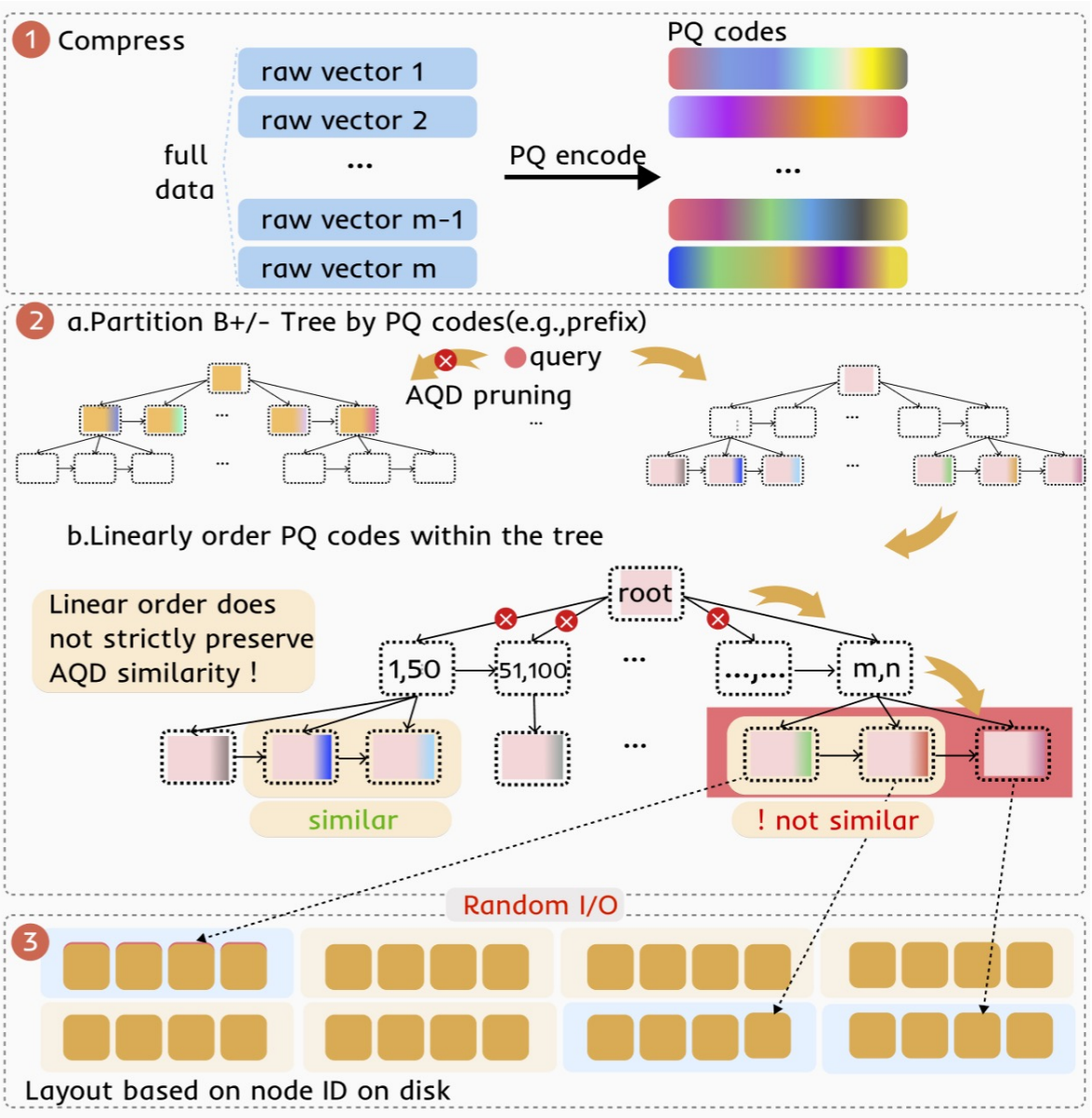


(1) Mapping vectors into low-dimensional Representations using hash mapping

(2) Access projected objects sequentially **in increasing order of their distance** to the query

(3) Sequential I/O occurs within each accessed disk

Tree + Quantization



(1) Compress raw vector into PQ codes

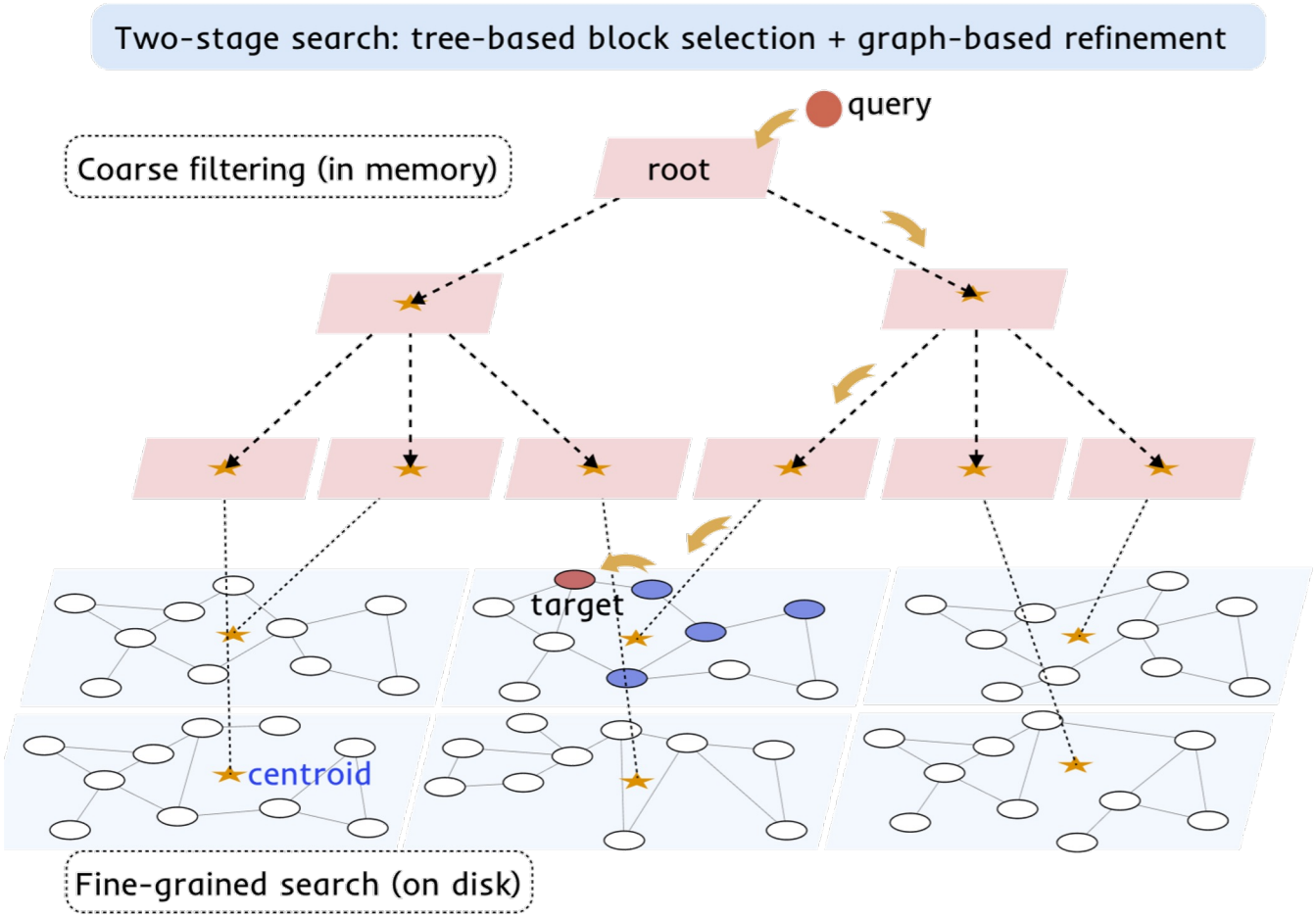
(2) Vectors are partitioned by PQ codes and indexed using a **PQB+-forest**, which linearly orders PQ codes within each tree

(3) Use PQ distances to first select promising PQB+-trees, and then prune subtrees within each selected tree

Tree + Hierarchical-Clustering

(1) Data Partition & Layout

- Partition data via k-means
- Group similar vectors into contiguous blocks
- Build a B+-tree with centroids (internal) and vectors (leaf)



(2) Coarse Filtering

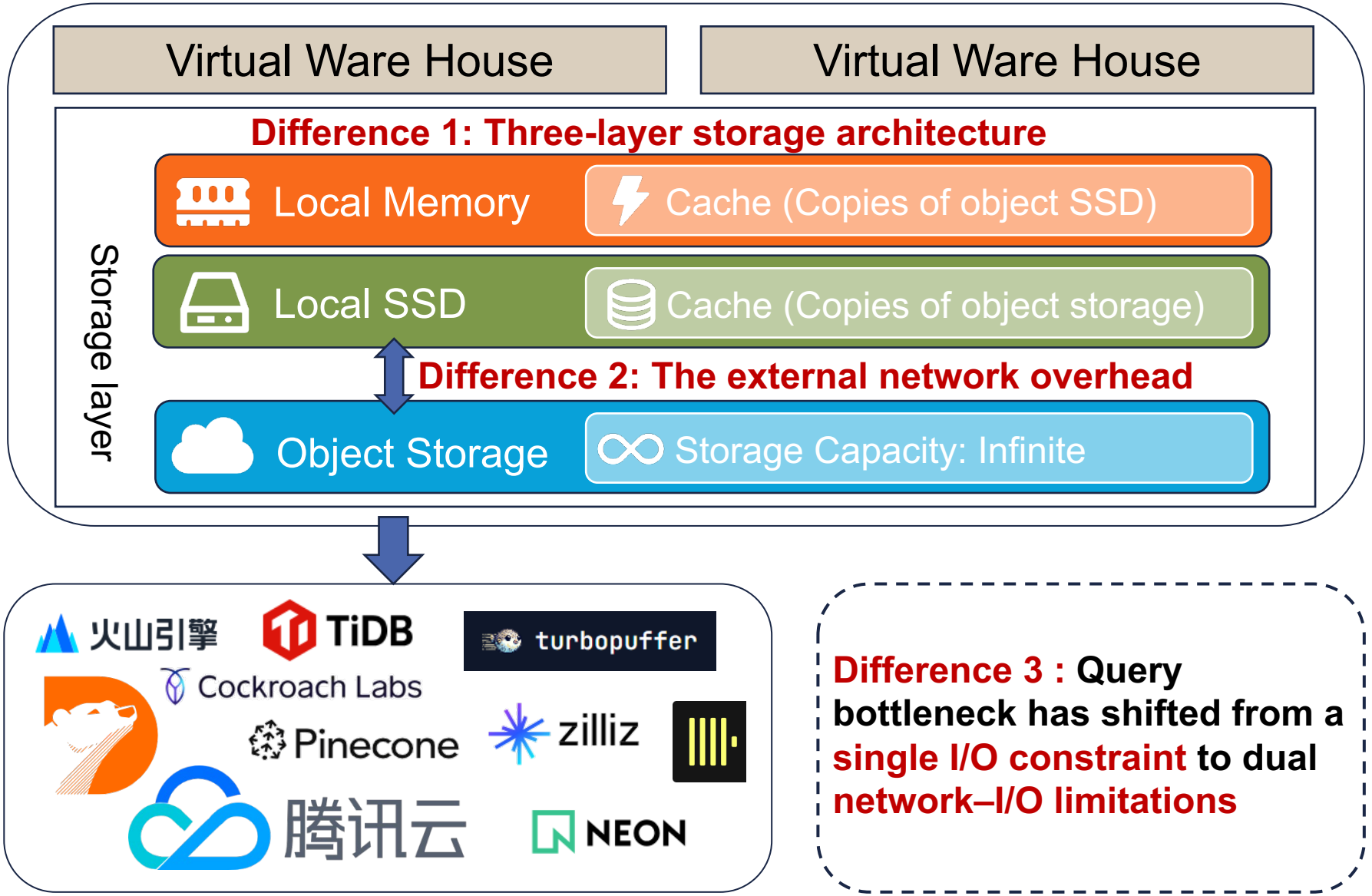
- Traverse the B+-tree using cluster centroids
- Identify a small set of promising leaf nodes

(3) Fine-Grained Search

- Load blocks
- Perform in-block graph search
- Refine final candidates

Part 3: Elastic Multi-Tiered VS

Elastic Multi-Tiered VS



Difference 3 : Query bottleneck has shifted from a single I/O constraint to dual network–I/O limitations

Elastic Multi-Tiered VS zilliz

Zilliz Cloud Tiered Storage Architecture

Three-tier cold-hot storage with on-demand loading

Fastest

Local Memory(Hot Tier)

routing metadata, index headers, and recently accessed vectors

Local SSD(Warm Tier)

frequently queried index files and recent update logs

Largest

Storage and layout

Cold-Hot Data Separation

Hot data in memory, warm data on SSD, cold data in object storage

On-Demand Loading

Cold data loaded to SSD/memory only when needed

Query and updates

Hierarchical Query

Memory locate → SSD search → object storag e fetch

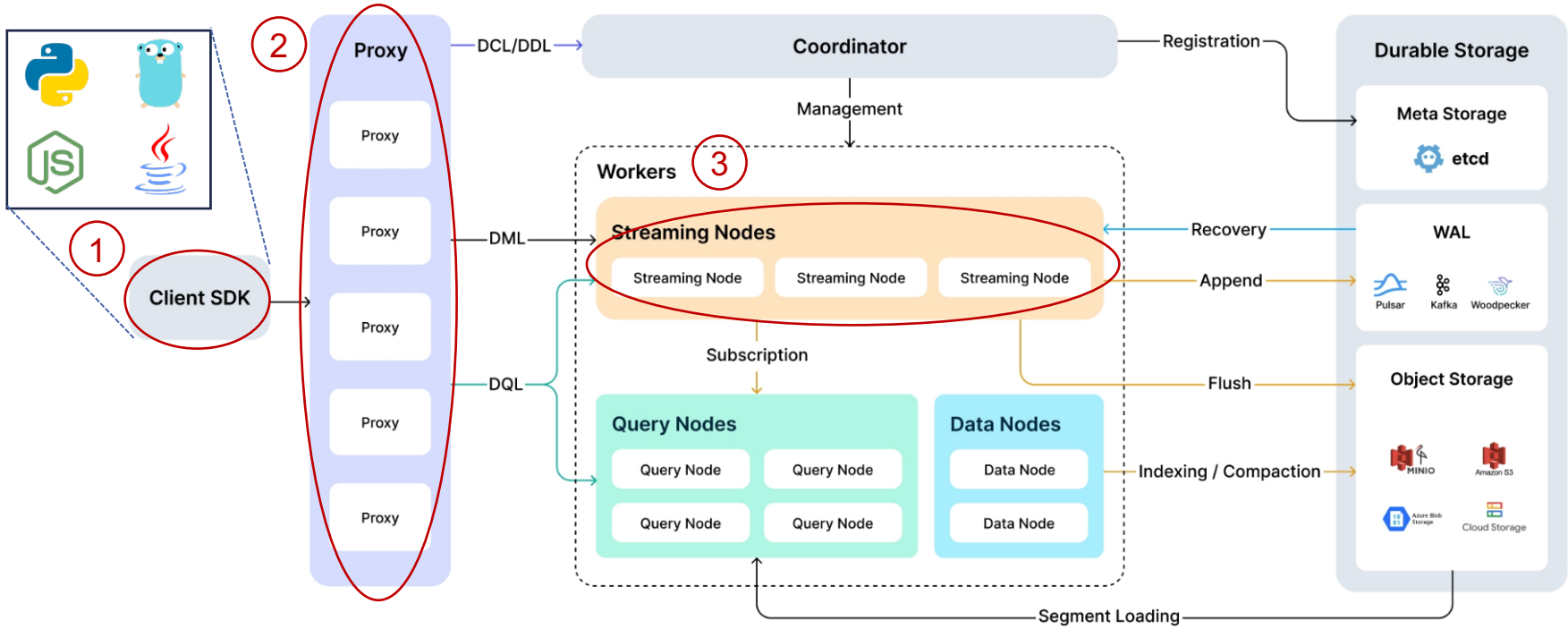
Append-Only Updates

Delta logs, tombstones + background compaction, avoiding full rebuild

Elastic Multi-Tiered VS zilliz

Zilliz Cloud Query Execution Flow

Distributed, Hierarchical Aggregation

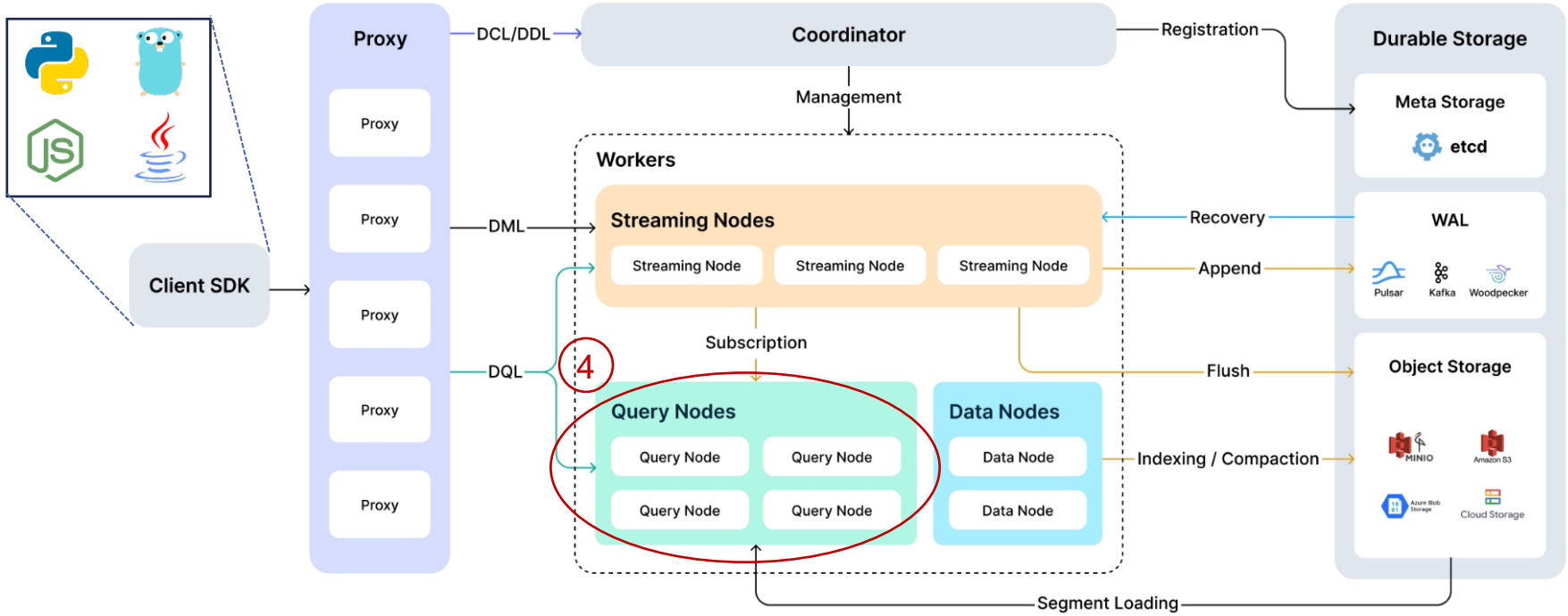


1. Client sends a search request via SDK/RESTful API to available Proxy.
2. Proxy locates target nodes.
3. Streaming Nodes search growing data and trigger Query Nodes for sealed data.

Elastic Multi-Tiered VS zilliz

Zilliz Cloud Query Execution Flow

Distributed, Hierarchical Aggregation



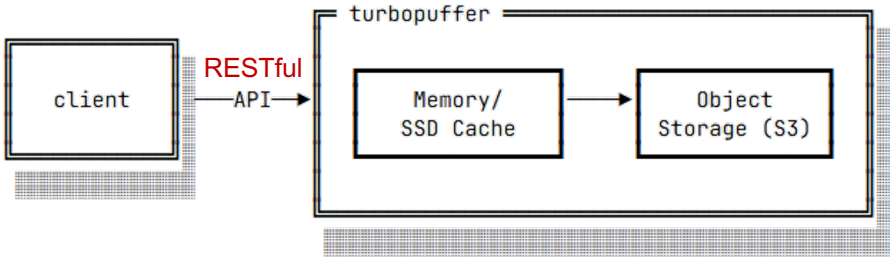
4. Query Nodes search sealed segments using local vector indexes (e.g., IVFPQ, HNSW).
5. Results are merged across segments and nodes, then returned.

Elastic Multi-Tiered VS turbopuffer

turbopuffer Core Capabilities

Hierarchical Caching, SPFresh Index, and Object-Native

Hierarchical Caching



Autonomous SPFresh indexing

Two-step Search: 1) Fast centroid lookup; 2) Batch fetching of cluster offsets from S3.

Caching Strategy

- Caches active data; offloads the rest to object storage
- Pre-warm caches via pre-flight query hints

SPFresh Index

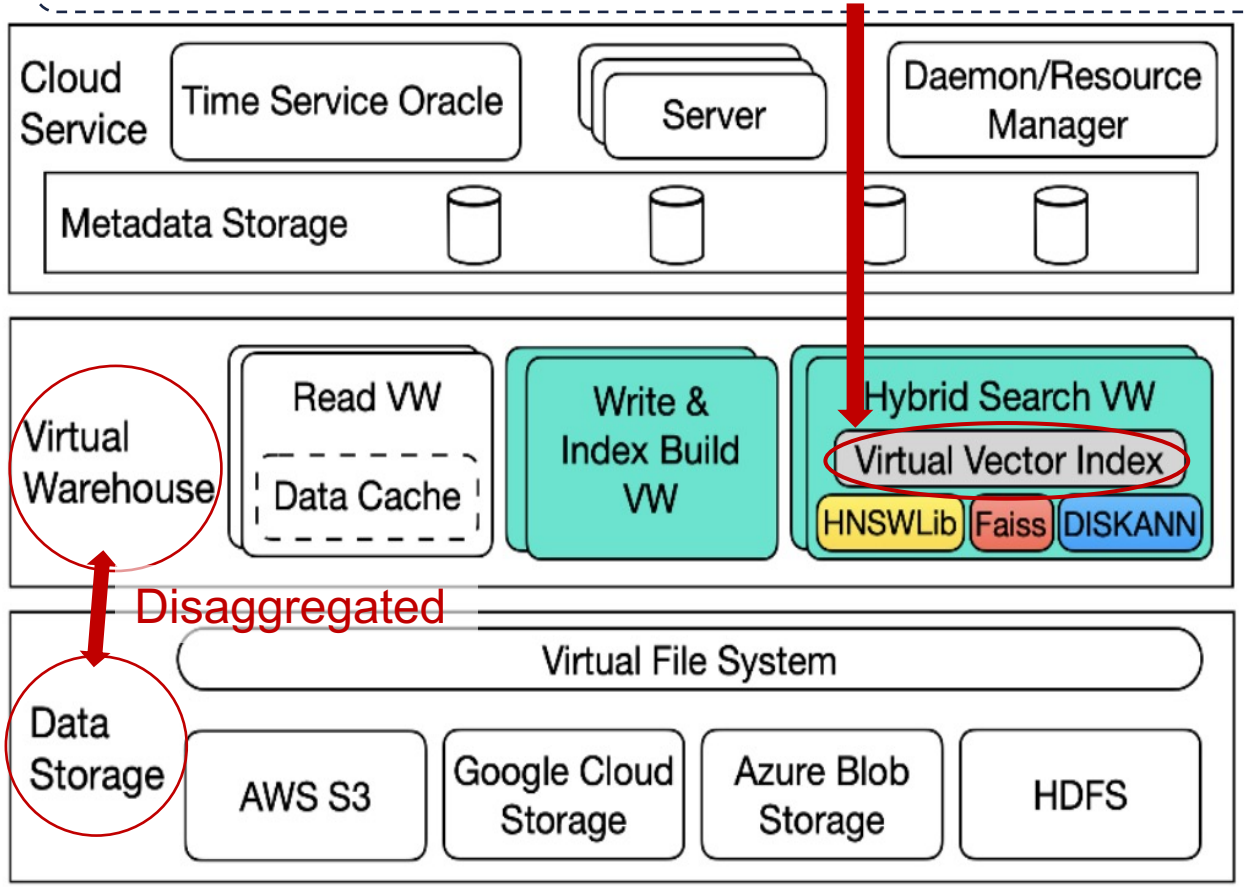
- Autonomous: Zero-config. Indexes are built and optimized automatically in the background.
- Fast Pruning: Only search relevant clusters by performing a fast centroid lookup in memory.
- Object-Native: Effectively masks object storage latency via massive I/O parallelism.

Elastic Multi-Tiered VS 火山引擎

BlendHouse System Architecture

Disaggregated architecture, and Multi-tiered hierarchical caching

Virtual Vector Index : standardized abstraction layer that provides unified interfaces for pluggable and extensible vector index management.



Multi-Tiered Storage

Local Memory
- Metadata(index structure information)
- Actual data(Original vector data)

Local Disk
- Local disk index cache

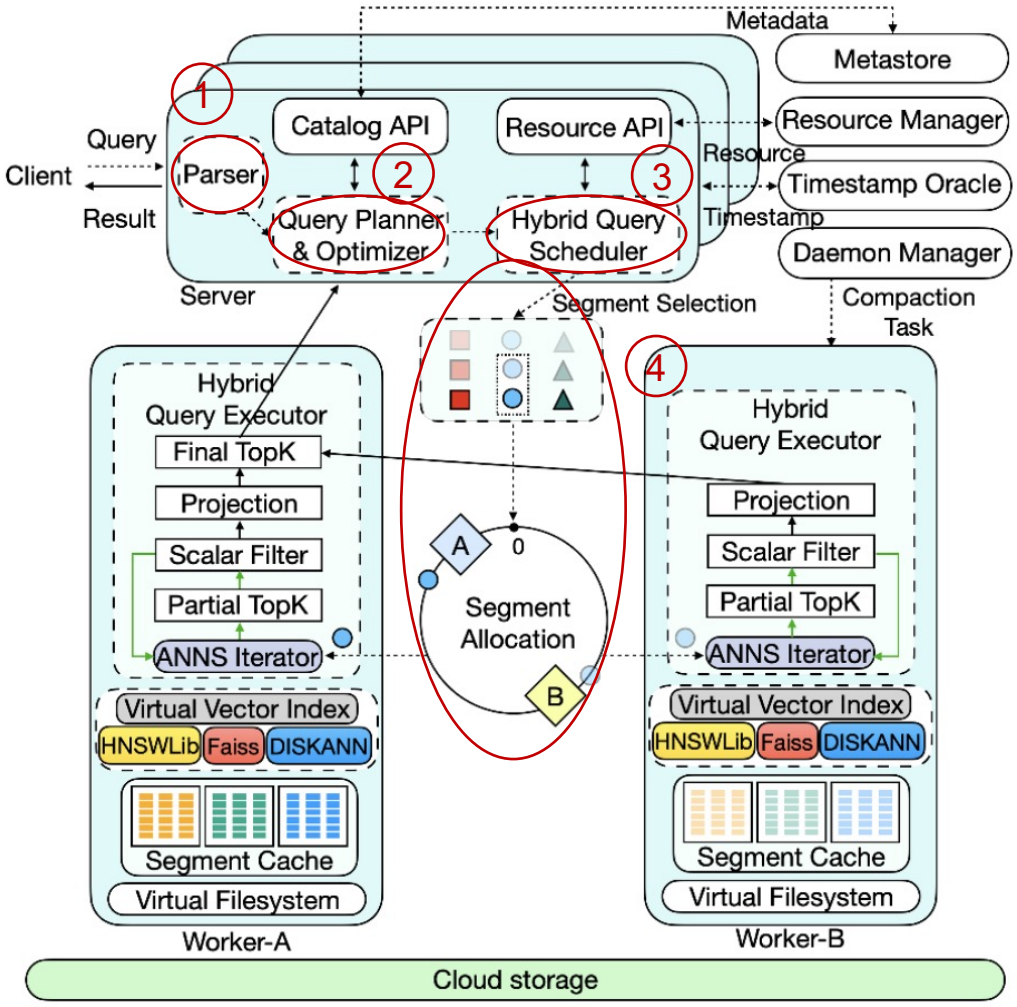
Remote Storage
- Vector data
- Indexes

The system employs the **LRU** policy for cache management.

Elastic Multi-Tiered VS 火山引擎

BlendHouse Vector Search Framework

Cost-based hybrid query processing



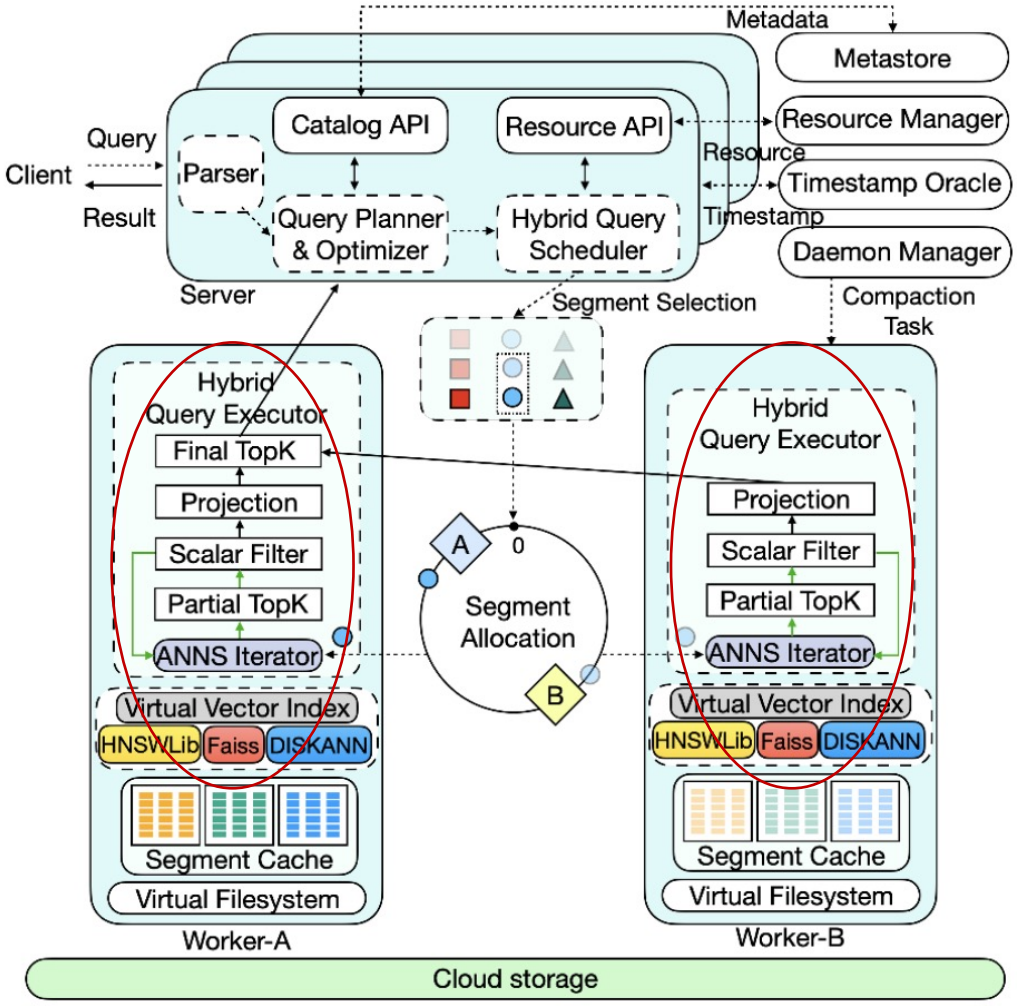
Plan generation and Scheduling

1. **Parser** receives the SQL query and generates the initial query plan.
2. **Query Planner & Optimizer** performs RBO/CBO optimization.
3. **Hybrid Query Scheduler** dispatches instructions of tasks.
4. **Segment Allocation** utilizes a consistent hashing ring to implement a scheduling strategy.

Elastic Multi-Tiered VS 火山引擎

BlendHouse Vector Search Framework

Cost-based hybrid query processing



Hybrid Query Executor

6. **ANNS Iterator** provides an incremental vector retrieval interface.
7. **Scalar Filter** performs attribute filtering on non-vector data.
8. **Partial and Final TopK** perform multi-level reduction to yield the global Top-K.

Conclusion

Challenges of Existing Vector Search Systems

Query Interface

Query interface barriers

- Rely on proprietary APIs/SDKs (e.g., Faiss)
- Lack standard SQL, increasing complexity
- High difficulty for business integration

Data-Index Strategy

Separation of data and index

- Storage-compute separation via object storage
- Indexes built per horizontal data segment
- Indexes stored as isolated binary files

Query Execution

Execute index island

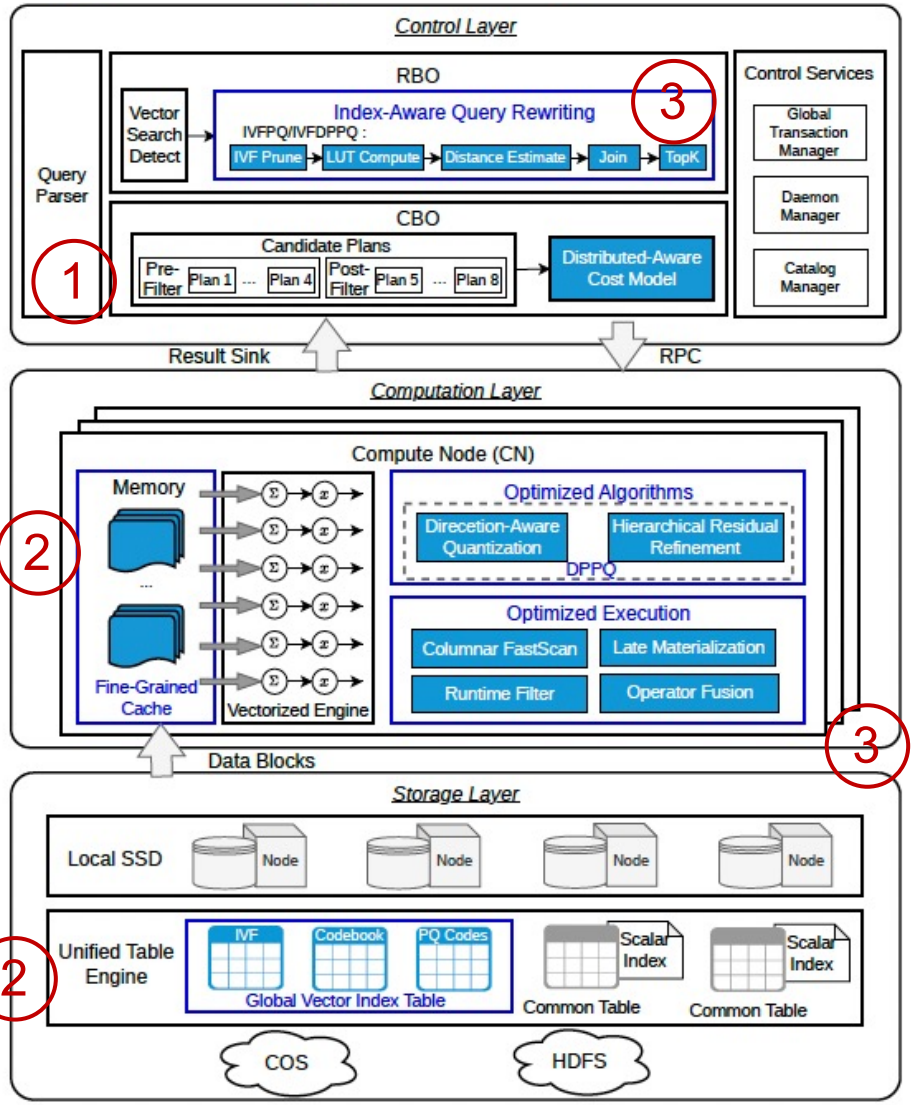
- Vector search runs as an opaque "black-box"
- SQL optimizer lacks a shared cost model
- Disjointed relational filtering and search

The TEngineDB-V Solution: Unified SQL Semantics × Table-Centric Global Indexing × Relational Execution & Distributed Optimization

TEngineDB-V Overview

An OLAP-Native Vector Search System for Large-k Workloads at Tencent

1. **Query Interface**
   - SQL Representation of Vector Search Semantics
2. **Storage & Index Strategies**
   - Table-Centric, Unified Storage Engine
   - Segment-Decoupled Global Indexing
3. **Query Execution**
   - Relational Execution
   - Distributed-Aware Optimization



TEngineDB-V Solutions: Query Interface

SQL Representation of Vector Search Semantics

Core Advantages

- **Low Learning Curve:** Bypass complex APIs (e.g., Faiss) and fully reuse standard SQL.
- **Seamless Integration:** Execute vector search and scalar filtering together in a single SQL query.
- **Native Data Modeling:** Treat vectors natively (array<float>) within standard distributed tables.

Hybrid Search Results

Vector Search

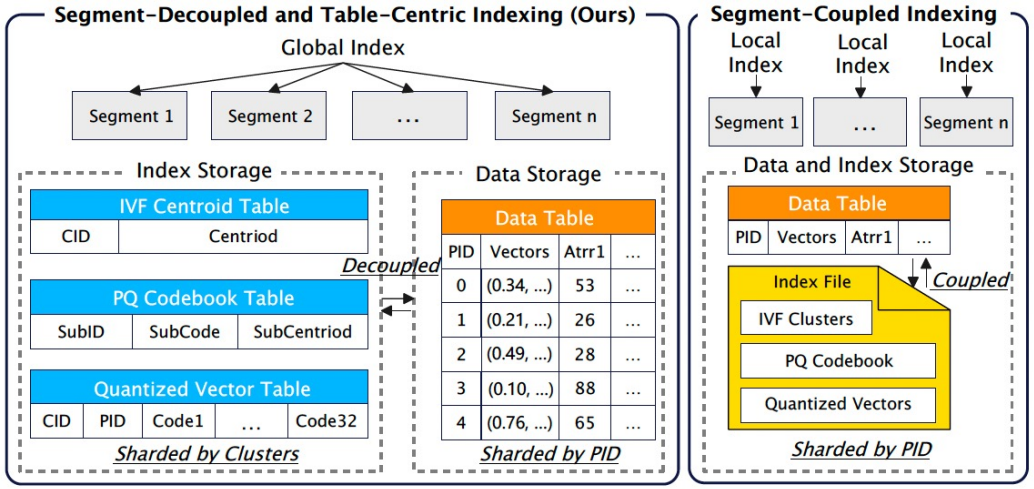
Scalar Filter

```sql
SELECT category, count(*), min(dist), max(dist)
FROM (
    SELECT category, approx_l2_distance(
        embedding, [0.1,0.2, ...]
    ) dist
    FROM images WHERE width > 1024
    ORDER BY dist LIMIT 1000000
) t
GROUP BY category;
```

```sql
CREATE TABLE images (
    id bigint,
    embedding array<float>,
    category varchar,
    height bigint,
    width bigint
)
PRIMARY KEY(id)
DISTRIBUTED BY HASH(id) BUCKETS 512;
```

TEngineDB-V Solutions: Storage & Index Strategies

Table-Centric, Unified Storage Engine



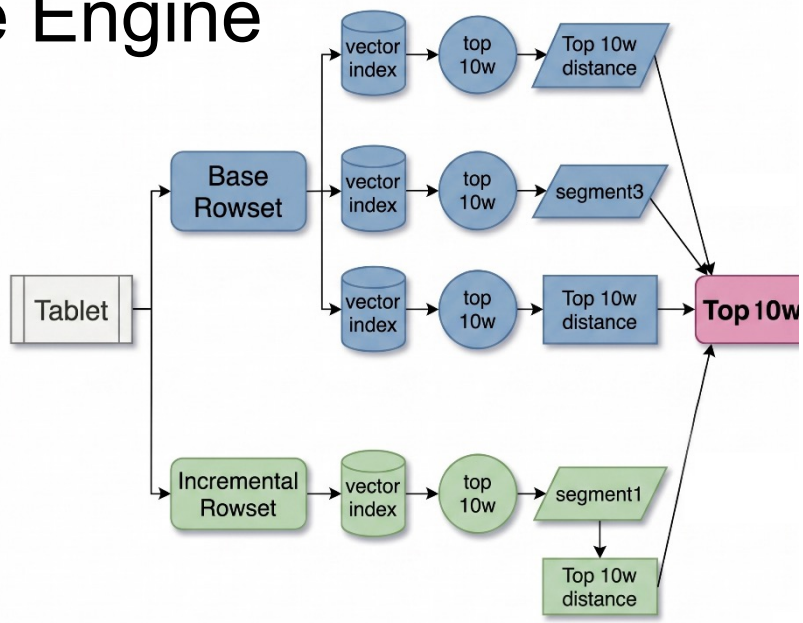


Table-Centric, Unified Storage Engine

- **Storage Compression:** Reduces disk space and I/O overhead via built-in compression.
- **Smart Caching:** Caches specific blocks on-demand to boost memory efficiency.

Segment-Decoupled Global Indexing

- **Avoid Scatter-Gather:** Eliminates performance bottlenecks for large-k workloads.
- **Precise Reads:** Prevents loading unnecessary pruned data.

TEngineDB-V Solutions: Query Execution

1 IVF Prune

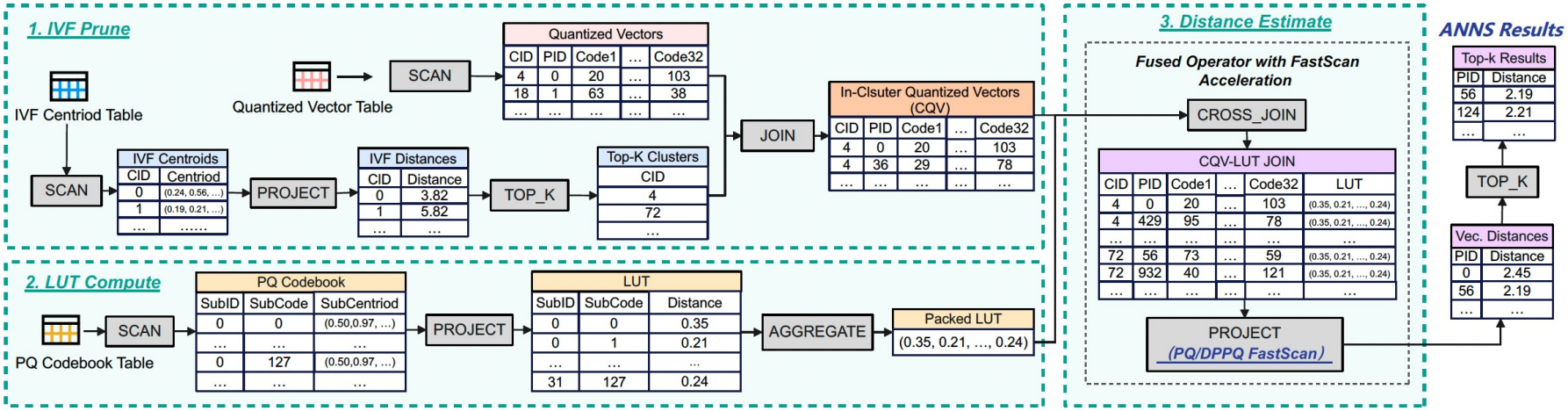
- Calculate Query-to-Centroid distances.
- Select nearest clusters via Top_K.
- Extract candidates via JOIN.

2 LUT Compute

- Build Look-Up Table (LUT) via PROJECT.
- Aggregate rows via AGGREGATE.
- Convert to single array to avoid data copies.

3 Distance Estimate

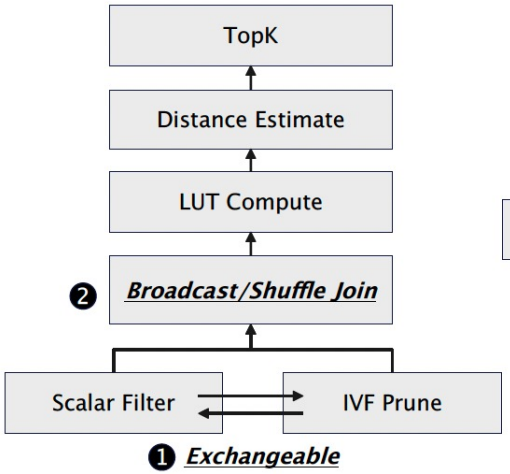
- Broadcast LUT to candidates via CROSS_JOIN.
- Accumulate distances via table lookup.
- Output final Top-K results.



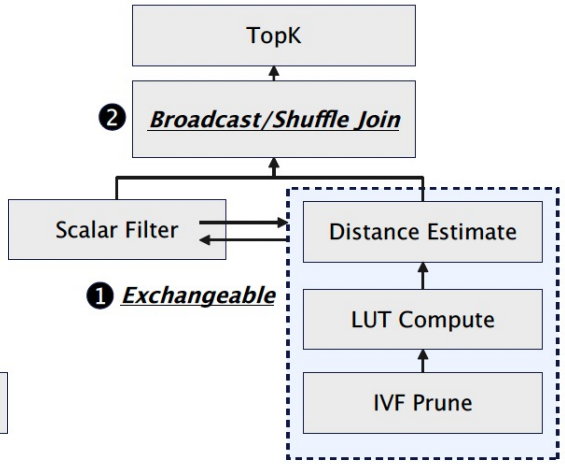
IVF-PQ Based Pipeline

TEngineDB-V Solutions: Query Execution

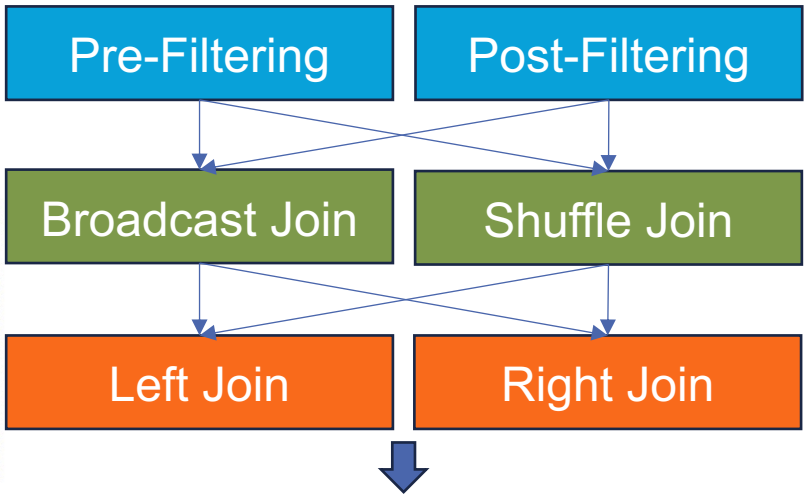
Table-Centric, Unified Storage Engine



(a) Pre-Filter



(b) Post-Filter



2x2x2 = 8 Query Plans

Symbol	Meaning
N	Total rows
N_f	Rows after filter
M_p	Rows after IVF Pruning
K_{ann}	Post-filter candidates
R_F	Scalar row width
R_V	Vector row width
C_{build}	Build cost
C_{probe}	Probe cost
C_{hash}	Hash cost
C_{dist}	Distance cost
P	# Nodes

Cost model for 8 plans

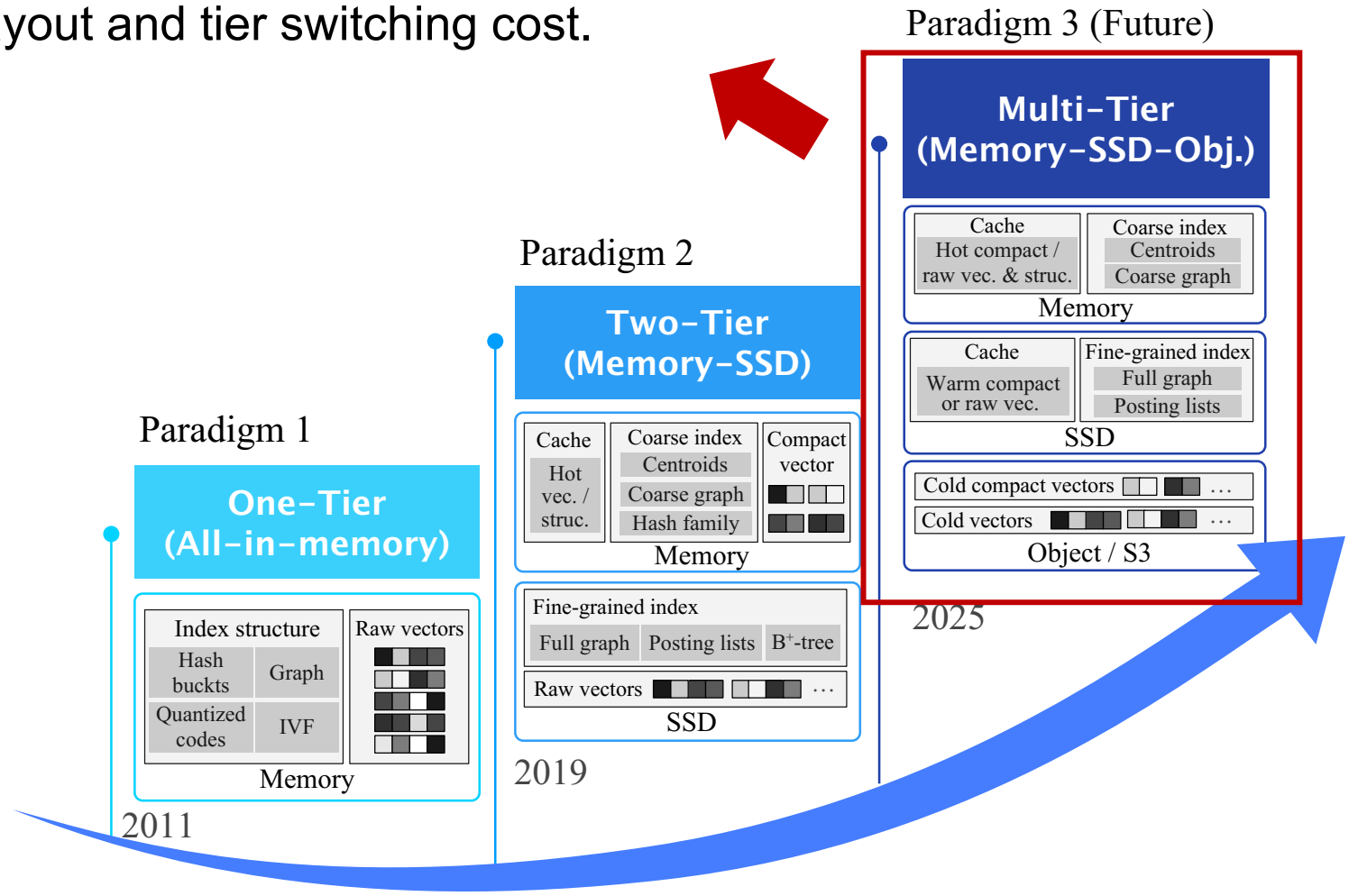
Plan	CPU Cost		Memory	Network
	ANNS	Join		
Plan 1 (Pre-Filter-Broadcast-Probe)	M_pC_{dist}	$M_pC_{build} + N_fC_{probe}$	M_pR_V	PM_pR_V
Plan 2 (Pre-Filter-Shuffle-Probe)	M_pC_{dist}	$(N_f + M_p)C_{hash} + M_pC_{build} + N_fC_{probe}$	M_pR_V	$N_fR_F + M_pR_V$
Plan 3 (Pre-Filter-Broadcast-Build)	N_fC_{dist}	$N_fC_{build} + M_pC_{probe}$	N_fR_F	PN_fR_F
Plan 4 (Pre-Filter-Shuffle-Build)	N_fC_{dist}	$(N_f + M_p)C_{hash} + N_fC_{build} + M_pC_{probe}$	N_fR_F	$M_pR_V + N_fR_F$
Plan 5 (Post-Filter-Broadcast-Probe)	M_pC_{dist}	$K_{ann}C_{build} + N_fC_{probe}$	$K_{ann}R_V$	$PK_{ann}R_V$
Plan 6 (Post-Filter-Shuffle-Probe)	M_pC_{dist}	$(N_f + K_{ann})C_{hash} + K_{ann}C_{build} + N_fC_{probe}$	$K_{ann}R_V$	$N_fR_F + K_{ann}R_V$
Plan 7 (Post-Filter-Broadcast-Build)	M_pC_{dist}	$N_fC_{build} + K_{ann}C_{probe}$	N_fR_F	PN_fR_F
Plan 8 (Post-Filter-Shuffle-Build)	M_pC_{dist}	$(K_{ann} + N_f)C_{hash} + N_fC_{build} + K_{ann}C_{probe}$	N_fR_F	$K_{ann}R_V + N_fR_F$

Jointly evaluates CPU, memory, and network overhead.

Part 4: Challenges and Open Problems

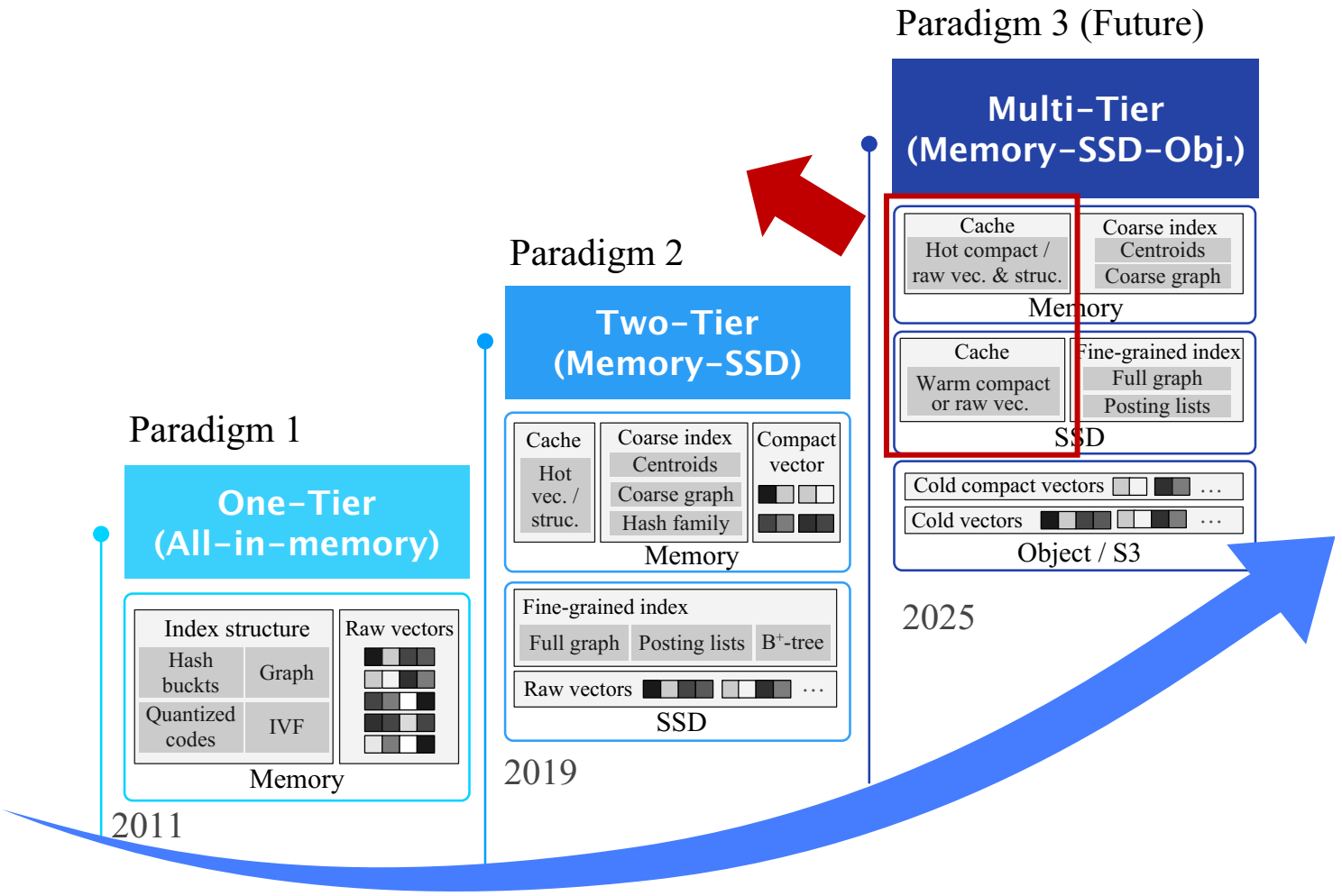
Enhancing Elastic Multi-Tiered VS

1. **Tier-aware Index Co-Design,** where index construction and query strategies must jointly consider SSD- and object-storage-aware data layout and tier switching cost.

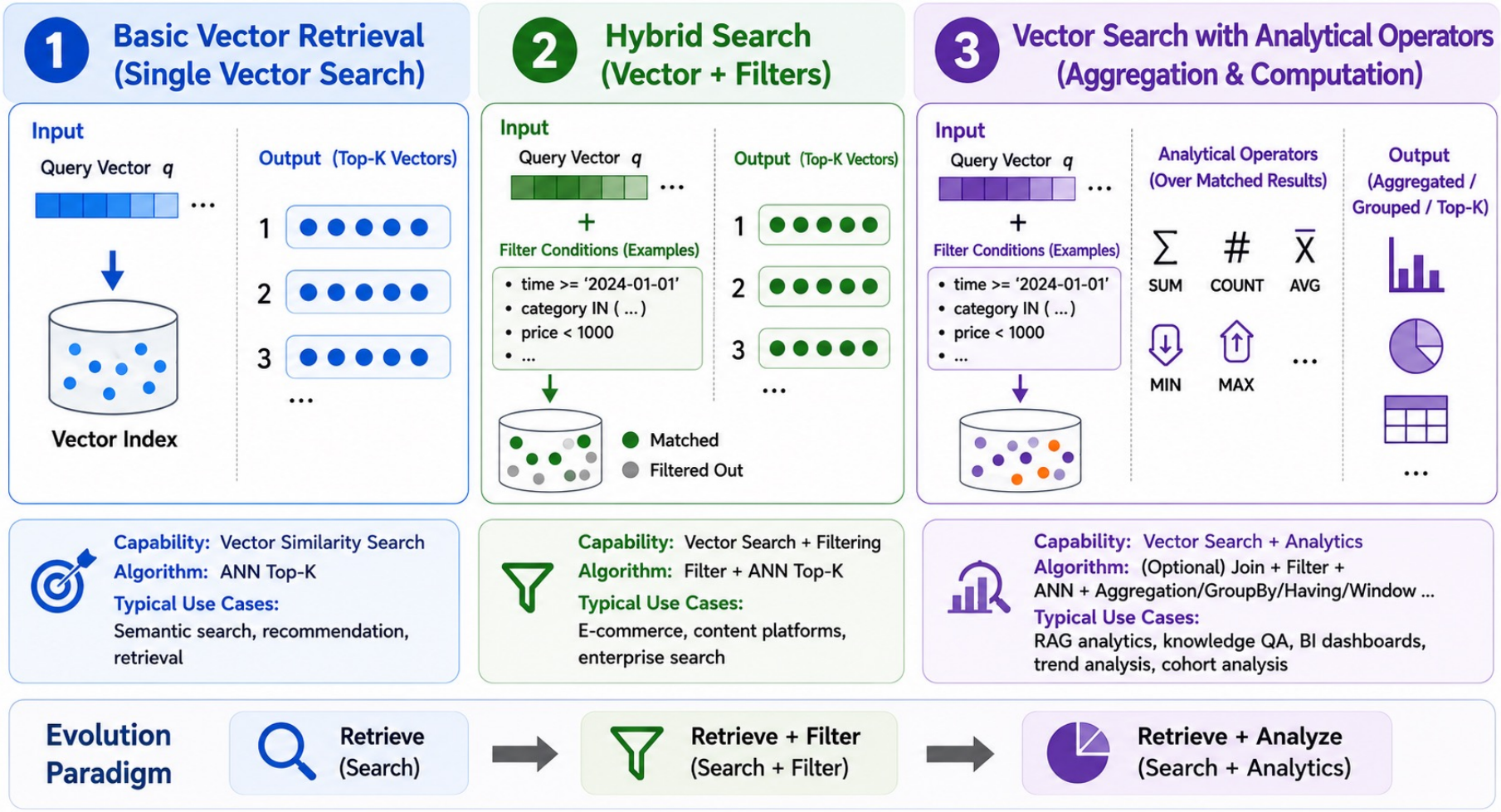


Enhancing Elastic Multi-Tiered VS

2. **Adaptive and Predictive Caching**, which leverages access prediction to proactively migrate vectors and reduce latency.



Challenges and Open Problems



Thanks for your listening !

Yitong Song[1] Xuanhe Zhou[2] Christian S Jensen[3] Jianliang Xu[1]

[1]Hong Kong Baptist University
[2]Shanghai Jiao Tong University
[3]Aalborg University

2026-06-02