Scalable Vector Search: From Exact Approaches to Approximate Nearest Neighbors

Diego Rivera Correa
CS6240

Why This Belongs in CS6240

- **Claim:** *Vector search is a distributed data-processing problem wearing an AI interface…*

K-Means	**Graph Search**	**Partitioning**	**Top-k merge**
IVF partitions vectors using centroids	HNSW navigates a proximity graph	Shards decide who scans what	Local results become global top-k

The same course ideas reappear: data layout, locality, parallel scans, communication, skew, and approximate answers.

Agenda

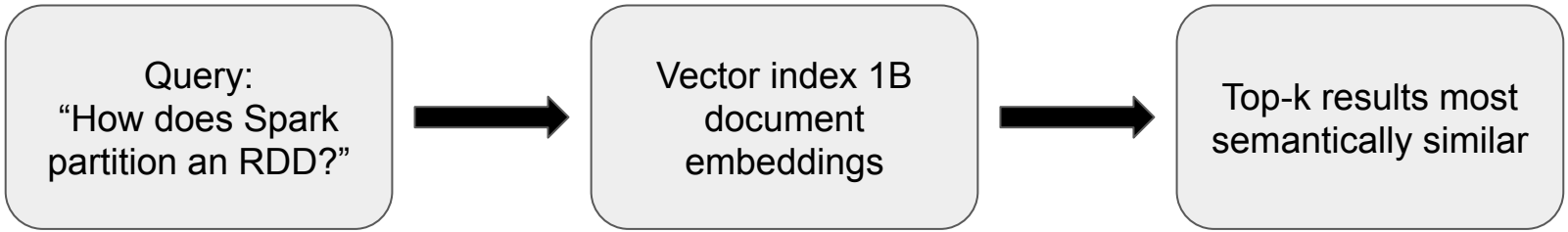
- Vectors and exact search
- Similarity, top-k, cost model
- IVF as partitioning space
- HNSW as graph search problems
- Distributed search

After This Lecture

- Why is exact vector search expensive even though it parallelizes well?
- How does IVF turn nearest-neighbor into a partitioning problem?
- How does HNSW turn nearest-neighbor search into graph traversal?
- How do local top-k results become a global top-k answer?

The Query Problem

- Goal: Given a query object, retrieve the most similar records from a very large collection.



The output is not an exact keyword match. It is a ranking by geometric similarity.

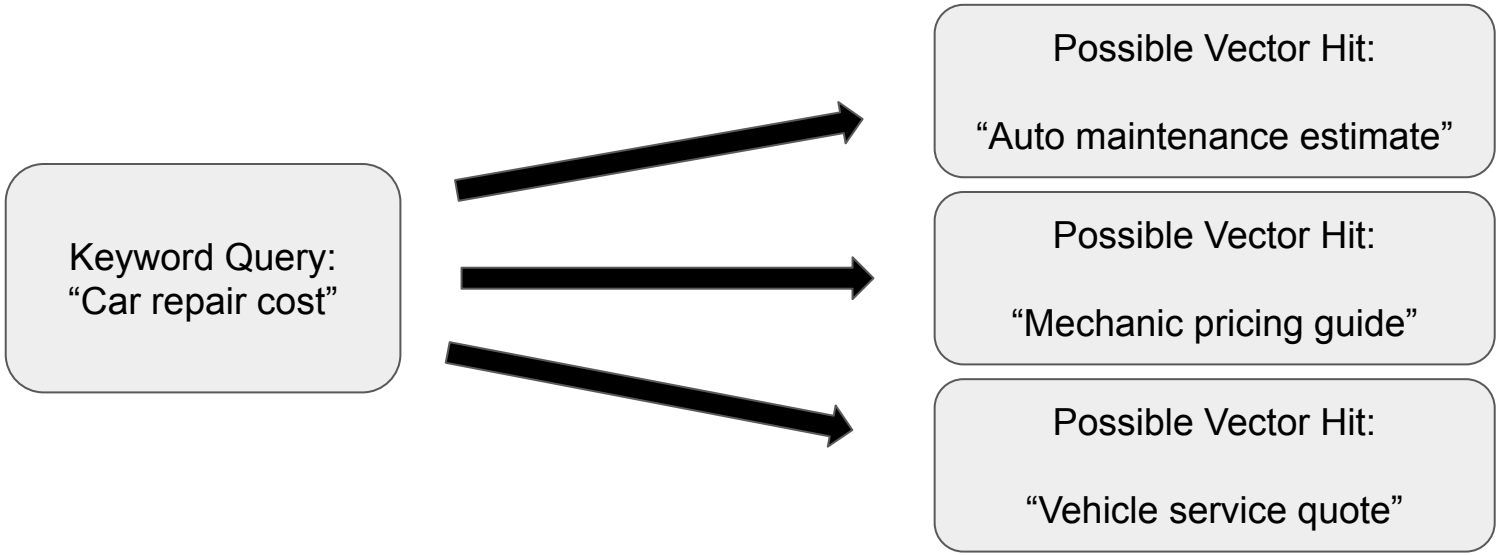
The Query Problem

This query problem appears in many other settings, including:

- Online shopping: Query = a product image or description like “black running shoes.” Search returns the most similar product embeddings.
- Image retrieval: Query = an image, such as a photo of a chair. Search returns visually similar images or objects.

Why Keyword Search is not Enough

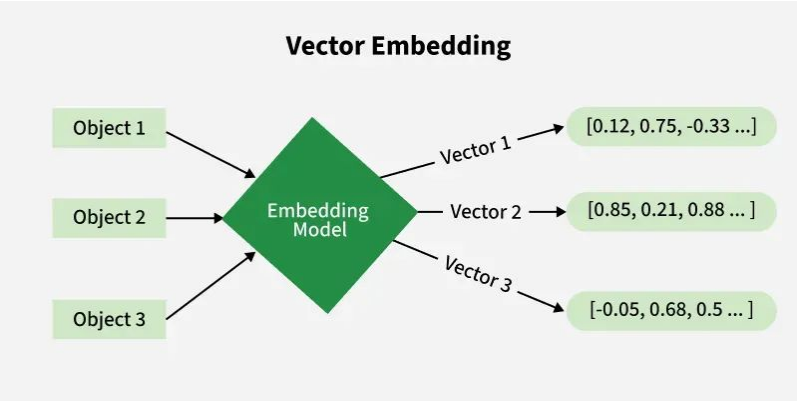
- Semantic matches may use different words.

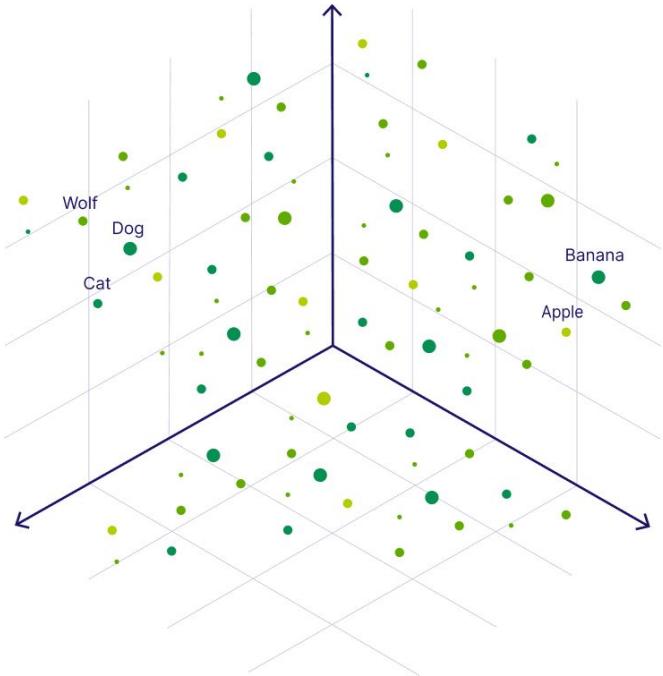


Vector search extends retrieval beyond exact tokens, but introduces high-dimensional nearest-neighbor search.

Embeddings: Objects Become Vectors

- A model maps each object to a d-dimensional representation.





Vectors as Data Records in a DFS

- A vector is just a large numerical record with an ID and metadata

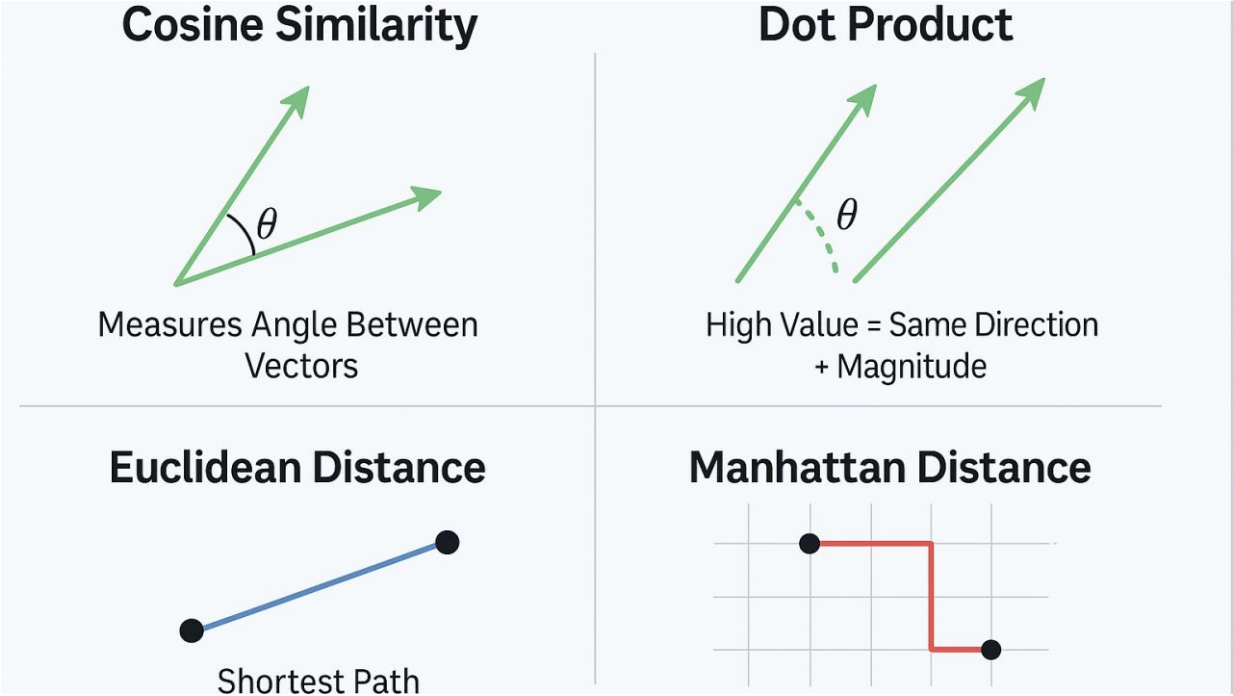
ID	Embedding	Metadata
doc42	[0.12, -0.44, 0.91, …]	Course = CS6240
doc77	[-0.03, 0.50, 0.21, …]	Topic = graph
doc91	[0.73, 0.09, -0.13, …]	Tenant = A

Important detail: metadata filters often run with vector similarity

Example: “nearest documents about Spark, but only from course CS6240 and only after June.”

Similarity Measures

- Different systems use different scores, but the top-k problem is the same…



Formal Task: Top-k Nearest Neighbors

- The vector index implements a ranking function over stored vectors.

Database vectors

$X = \{x1, x2, \ldots, xn\}$

Query vector

q

Return

k vectors with largest $sim(q,x)$

Core problem: Find top-k quickly while scanning as little data as possible and preserving high recall. Let's work on it!

Exact Search Baseline

```
input_vec = q

top_k = {}

list_of_vecs = X

for candidate_vec in list_of_vecs:

        // update top-k list.

        rank_vec(top_k, input_vec, candidate_vec)

return top_k
```

Exact Search Baseline

```
input_vec = q

top_k = {}

list_of_vecs = X

for candidate_vec in list_of_vecs:

        // update top-k list.

        rank_vec(top_k, input_vec, candidate_vec)

return top_k
```

Cost per query = n x d similarity operations.

Exact search is simple, parallel, and accurate. However, it is expensive at high scale.

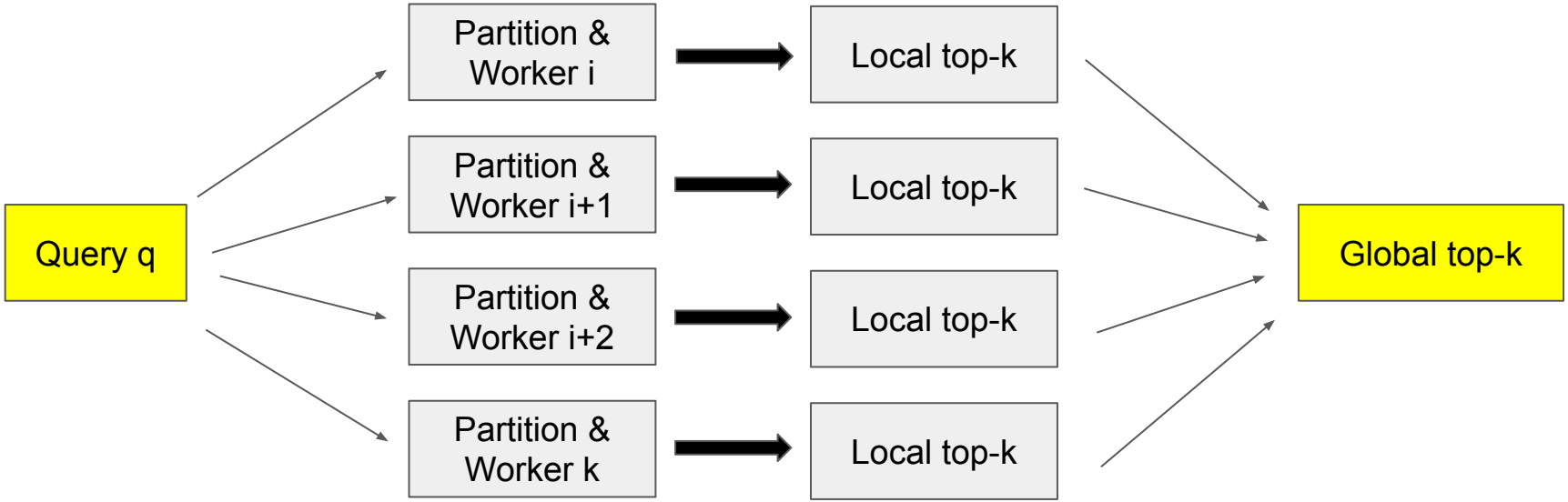
Why Exactness Breaks at Scale

- Suppose we have 1 billion vectors (hence comparisons), with each vector having 768 dimensions.

- If we can process 100 queries/sec, the cost of exact scan per second is approximately 7.68 x 10^13 dimension operations.

- Parallelism helps, but every query still consumes enormous CPU/GPU, memory bandwidth, and network resources.

Distributed Exact Search



This is the baseline distributed query plan that ANN systems try to avoid or reduce.

Exact Baseline Recap

- Strength = Simple, intuitive, easy to parallelize.
- Weakness = Touches every vector for every query.
- Question = Can we avoid scanning most vectors while still returning nearly the same top-k?

Approximate Nearest Neighbor Search

- ANN returns near-best results faster than exact search.
- Exact NN = Costly, but guarantees the true nearest neighbors.
- ANN = Cheaper than Exact NN, returns the same or nearly same neighbors.
- Trade-offs = Tune speed, recall, and memory.

Why Approximate can be Safe

- For retrieval, many good candidates may be nearly equivalent

Exact Rank	Score
A	.950
B	.948
C	.944
D	.931
E	.920

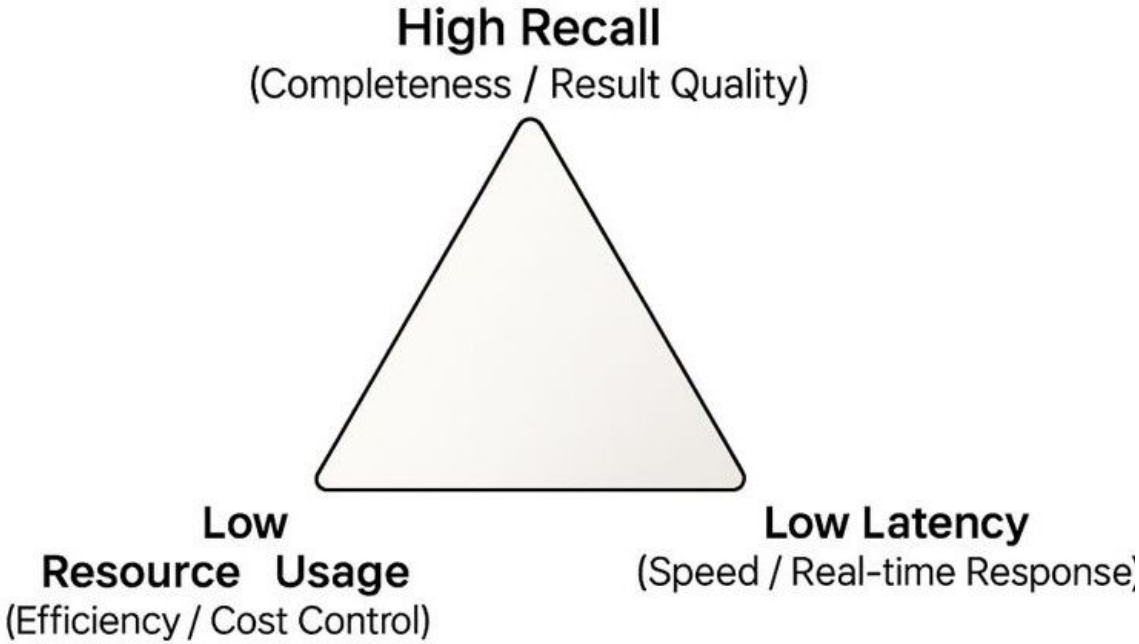
Observation: If many candidates are close in score, returning B rather than A may not change the user-visible answer much.

But: Some applications require exactness: duplicate detection, legal discovery, or safety-critical retrieval.

Recall@k: The Core Quality Metric

- Recall compares approximate results to exact results.
- Example: Exact NN top-5 = {A, B, C, D, E}, and ANN top-5 = {A, C, E, F, G}.
- Recall is their overlap = {A, C, E}. Recall@5 = 3 / 5 = 0.60.
- High recall means the approximate index found most of the exact top-k.

Recall-Latency-Memory Triangle



More speed often means lower recall or more memory.

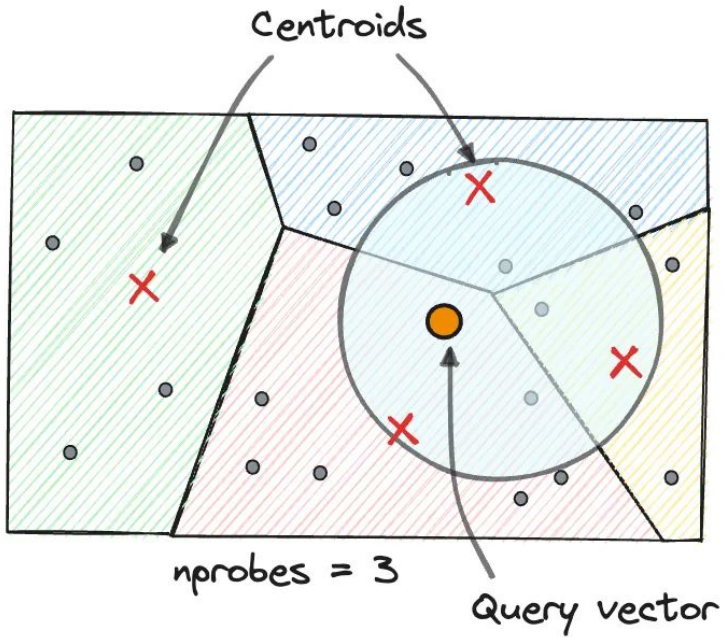
More recall usually means more work.

Where Approximation Enters

- ANN indexes avoid work by making educated guesses.
- **Routing:** Search only promising partitions or clusters.
- **Graph pruning:** Walk through a small subset of nearby nodes/vertices.
- **Compression:** Store approximate codes instead of full vectors.

IVF: Partition the Vector Space

- IVF = inverted file index. Intuition: Cluster first, search selected clusters.



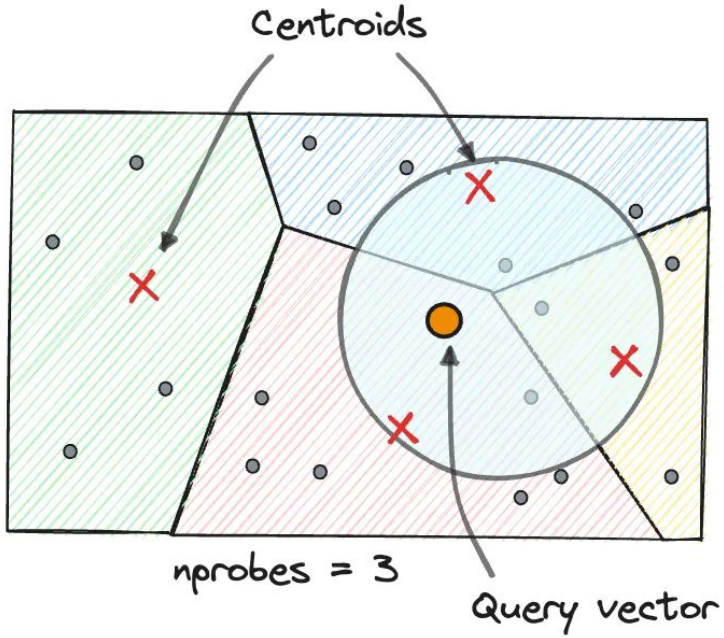
At query time, find the nearest centroids, then scan their list of vectors.

IVF Training Pipeline

- Step 1: Sample training vectors.
- Step 2: Run K-Means
- Step 3: Assign each vector to nearest centroid.
- Step 4: Store vectors in inverted lists.

Centroids are Routing Landmarks

- At query time, centroids are cheap representative of large regions.



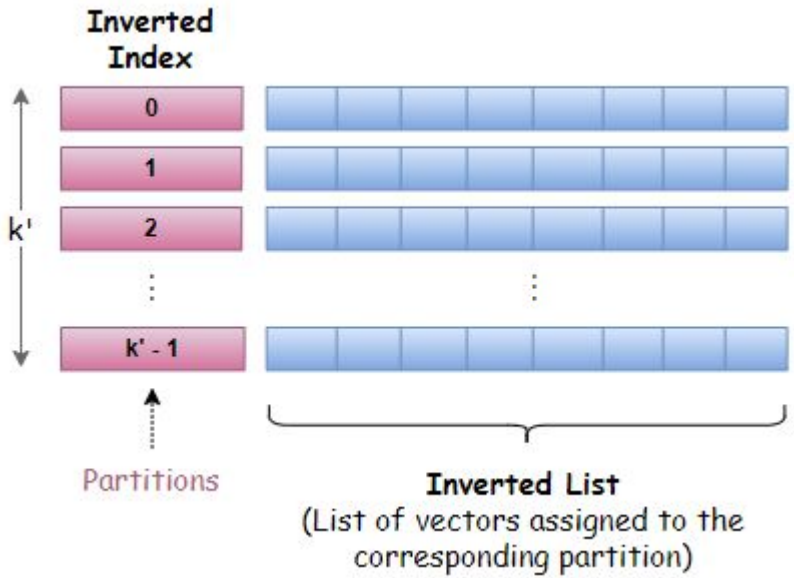
Important simplification: Compare q to 100 centroids rather than millions of vectors.

Then: Search only lists attached to the closest centroids.

Risk: The true neighbor may live just across a cluster boundary.

IVF Index Layout: Adjacency Lists

- Query-time work = score centroids + scan selected lists/



IVF Query Algorithm

- Goal: Query only the closest inverted lists.
- 1. Compute sim(q, centroid) for all centroids.
- 2. Pick nearest **nprobe** centroids.
- 3. Scan vectors inside those lists.
- 4. Merge candidate scores and return top-k.

IVF Numerical Example

- Approximation comes from scanning only a fraction of the database.

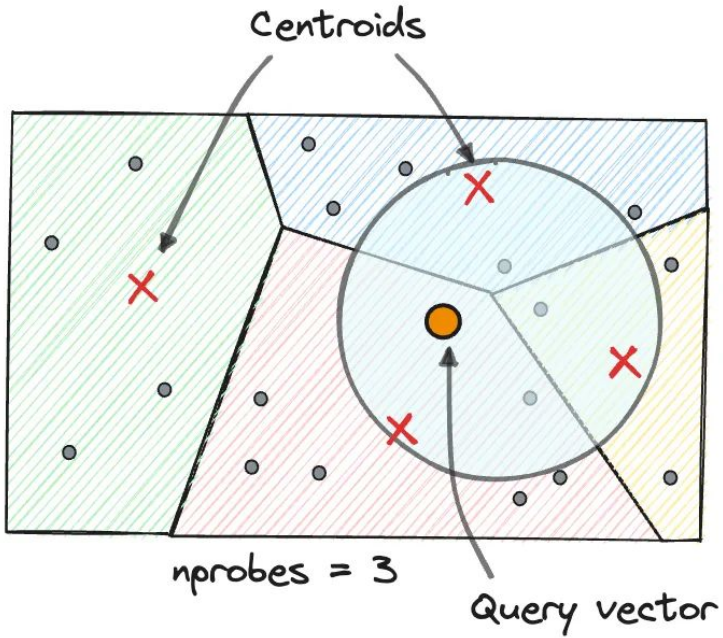
- Example: Assume |D| = 1M, and we set the number of clusters to 100 and **nprobe** to 5.
  - We scan 5 clusters with 10,000 vectors inside each cluster.

- Work reduction is roughly a factor of 20 (albeit centroid scoring overhead)

Nprobe as an IVF Quality Knob

nprobe	latency	recall	vectors scanned
1	fastest	lowest	smallest
5	medium	higher	moderated
20	slower	higher still	larger
100	exact over clusters	highest	full scan within IVF

IVF Failure Mode: Boundary Neighbors



IVF as a Distributed Query Plan

- We now explore building and using an IVF Index in MapReduce! The first step is identifying what can be done offline vs online

- Offline (index building): Run K-Means, Assign vectors to nearest centroids, Write IVF index by centrid.

- Online (query time): For query q → select nprobe lists, search selected lists and merge top-k.

Offline Step 0: Train Centroids

- K-means produces coarse routing landmarks for the vector space.

Centroid	Coordinates
C1	[0.1, 0.8]
C2	[0.7, 0.6]
C3	[0.2, 0.2]
C4	[0.8, 0.1]

Centroids are small relative to the database.

- Broadcast them to workers during index construction.
- Each vector can independently choose its nearest centroid.

K-means is not repeated for every user query.

Offline Job 1: Map Assigns Vectors to Centroids

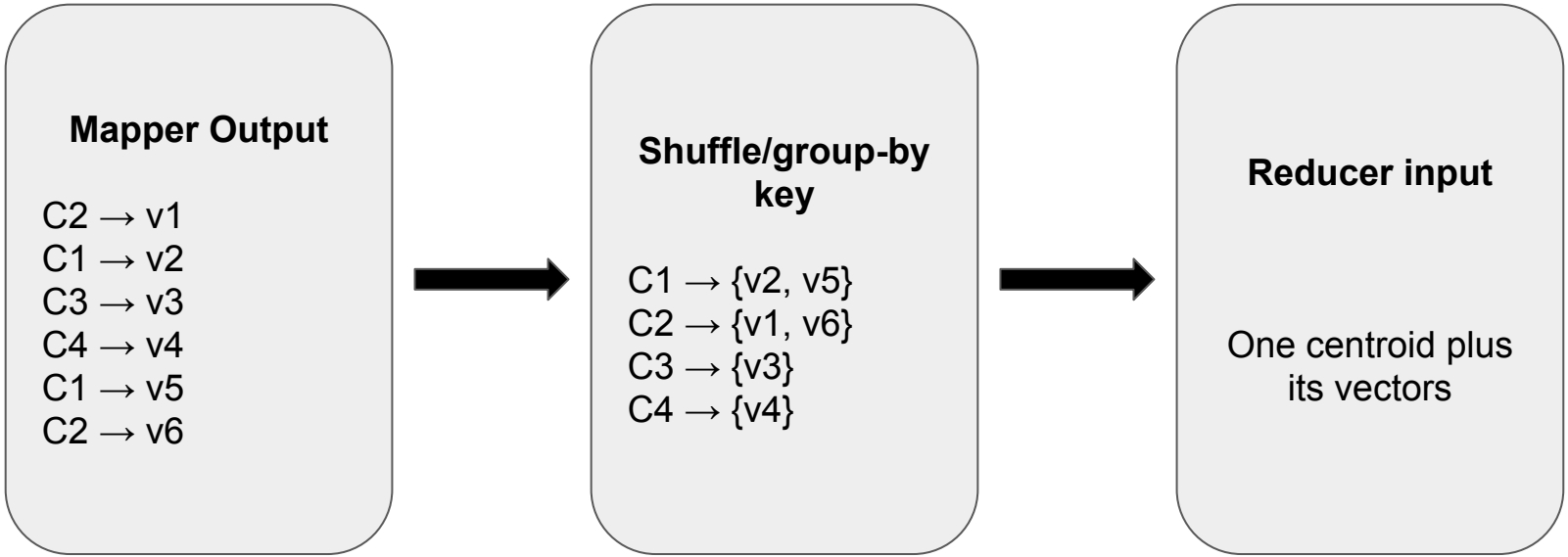
```
map(vectorID, vector x):
    bestC = Null
    bestD = inf

    // load centroids into cache via
    // setup()
    for each centroid c:
        d = distance(x, c)
        If d < bestD:
            bestD = d
            bestC = c.ID

    emit(bestC, (vectorID, x))
```

vector	nearest centroid
v1	C2
v2	C1
v3	C3
v4	C4

Shuffle Creates Inverted Lists



Offline Job 1: Reduce Writes Physical IVF Index

reduce(centroidID, vectors):

for each vector in vectors:
Write vector to
IVF_INDEX/centroidID/

IVF_INDEX/
C1/
part-000
part-001
C2/
part-000
part-001
part-002
C3/
part-000
C4/
part-000

Online Step 1: Choose the Closest Centroids to the Query

Centroid	distance(q, Ci)
C1	0.90
C2	0.20
C3	1.10
C4	0.50

If nprobe = 2, then we select the 2 closest centroids.

We select C2 and C4.

The coordinator only compares q against centroids, not against all vectors.

Correct Query Plan: Select Input Paths, Not Records

- We now define a custom MapInput based on the selected centroids.
- C2 and C4 are the job input paths.
- The win comes from less I/O and less distance computations.

IVF_INDEX/
C1/ not read
C2/ READ
C3/ not read
C4/ READ

Query Job 2: Map Assigns Vectors to Centroids

```
// create localTopK set in setup(). Assume query q is broadcasted to us.

// assume inputSplit is a list of vectors
map(inputSplit):

        for each (vecID, vec) in inputSplit:
                score = sim(q, vector)
                localTopK.add(score)

                If localTopK.size > k:
                        remove lowest score

emit localTopK in cleanup() using a DUMMY key
```

Query Job 2: Reduce Merges Local top-k Lists

reduce(DUMMY, localLists):

```
globalTopK = {} // e.g. empty min-heap

for each localTopK in localLists:
    for each candidate in localTopK:
        globalTopK.add(score)

        If globalTopK.size > k:
            remove lowest score

emit(globalTopK)
```

What is Approximate About IVF?

- **Small nprobe (e.g. 1):** Search one list.
  - Fast, but more likely to miss boundary neighbors.
- **Larger nprobe (e.g. 5):**
  - Search more lists. Slower, but higher recall.
- **Highest nprobe (e.g. Exact search):**
  - Search all lists. Highest recall, highest latency.
- Tuning IVF is mostly tuning the speed-recall trade-off.

From IVF to HNSW

- IVF focuses on: Train centroids, assign vectors to fixed lists, query nprobe lists.

- HNSW focuses on: Build a proximity graph, start from an entry point, walk towards the query.

- **Next Idea:** Replace centroid routing with graph navigation.

HNSW: Build a Graph Over Vectors

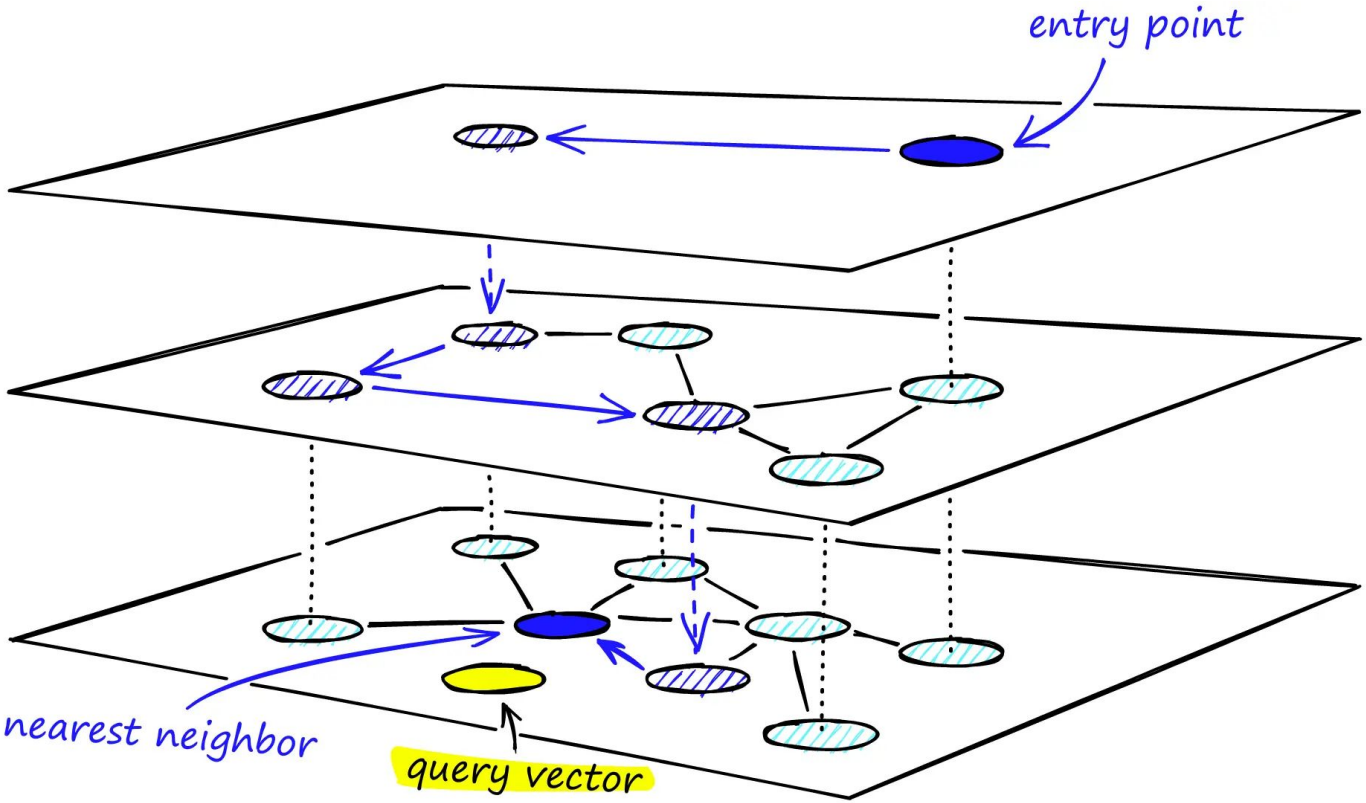
- **Vectors become vertices:** Each stored embedding is a graph node.
- **Nearby vectors get edges:** The graph connects points that are close in embedding space.
- **Search becomes navigation:** Start somewhere and walk toward nodes closer to the query.

HNSW Intuition (Highway Analogy)

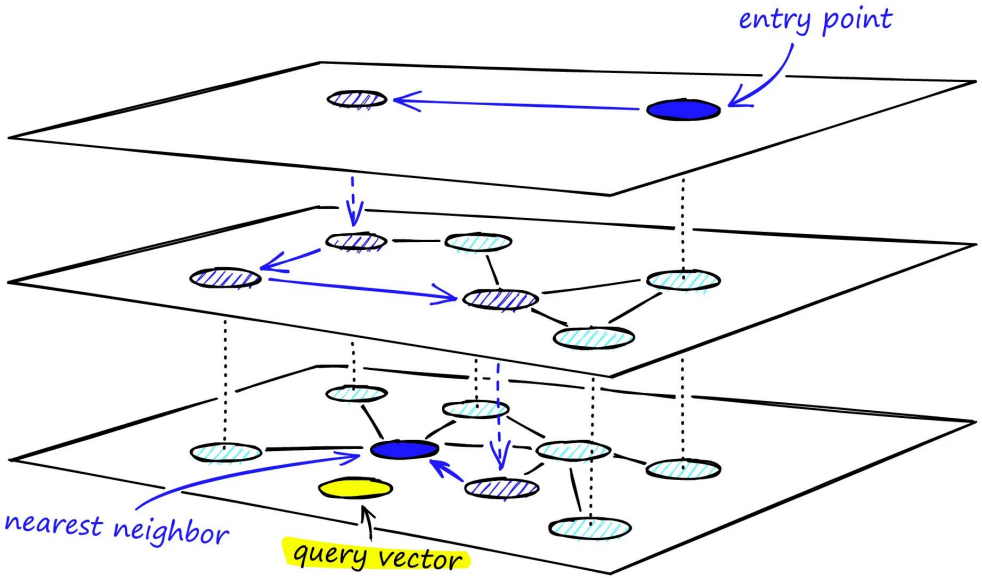
Find a specific house in a massive country using different types of roads:

- **Top layers (Interstate Highways):** Very few nodes and long-distance connections. You can cross the entire country in a few giant hops.
- **Middle layers (State Routes):** Once you exit the highway near the right region, you drop down to medium-range connections.
- **Bottom layer (Local Neighborhood Streets):** This layer contains every single house (vector). Here, you move step-by-step to your exact destination.

HNSW Intuition (Highway Analogy)



HNSW Intuition (Highway Analogy)



1. Start at an entry node, and evaluate neighbors.
2. Move closer and descend. Use long edges to get near q, then switch to denser lower layers.
3. Search locally. Explore nearby candidates, then return the top candidate.

Greedy Graph Search

- **Coarse Jump:** You start at a single entry point on the sparse top layer and greedily hop to the neighbor closest to your query vector.

- **Refinement:** When you can't get any closer on that layer, you drop down to the exact same node on the layer below.

- **Pinpointing:** You repeat the greedy navigation, zooming in closer and closer through the layers until you hit the bottom floor and find your exact nearest neighbors.

How HNSW Index is Built

- **Step 1:** Start with the vectors already inserted.
- **Step 2:** Search for neighbors and use current graph to find nodes near new vector X.
- **Step 3:** Add edges and connect X to several nearby nodes.
- **Step 4:** Assign levels. X always appears at the bottom; sometimes higher.

IVF vs HNSW

Feature	HNSW	IVF-Flat
Algorithm Type	Graph-based	Cluster-based
Recall	Very high	Moderate to high
Search Speed	Fast	Very fast (depends on nprobe)
Memory Usage	High	Moderate
Index Build Time	Slower	Faster (after training)
Supports Real-time Updates	Yes	No
Training Required	No	Yes (k-means)
Best For	Dynamic, high-recall systems	Static, large-scale datasets

Memory and Scale

HNSW is Fast, but Memory-Hungry

- **Raw vectors:** Every embedding must be stored or retrieved.
- **Neighbor lists:** Each vector stores links to nearby vectors. More links usually improves search quality.
- **Metadata:** Levels, IDs, offsets, implementation-dependent overhead.
- **Systems takeaway:** HNSW trades additional memory and build time for much faster query-time navigation. What happens when the index is too large or too hot for one machine?

When Does 1 Machine Stop Being Enough?

- **Capacity:** Index no longer fits in memory.
- **Throughput:** Too many queries for second.
- **Reliability:** One machine becomes a single point of failure.
- **Distribution** is not just “split the file”. It changes query routing, recall, memory, and fault tolerance.

Vectors and Memory Problems

- Consider we have 100M vectors, all with 768 dimensions, and each dimensions being represented with 4 bytes.

- The amount of memory needed to store that data amounts to 100M x 768 x 4 bytes = 307GB!

- **Then add:** ANN index such as IVF/HNSW, metadata, replicas, etc.

Sharding Vector Indexes

Terminology

Term	Meaning in this lecture	Why it matters
Partitioning	Divide data/index into pieces	Creates parallel work
Sharding	A partition used as a unit of ownership/distribution across machines.	Determines query routing
Replication	Store extra copies of data/index pieces	Improves throughput and fault tolerance

Partitioning says what the pieces are; sharding says where the pieces live.

Terminology (Shards and Partitions)

- All shards are partitions, but not every partition is a shard. Suppose we have 1B vectors and 4 machines. We might create 4 shards:
  - Shard 1 in M1
  - Shard 2 in M2
  - Shard 3 in M3
  - Shard 4 in M4

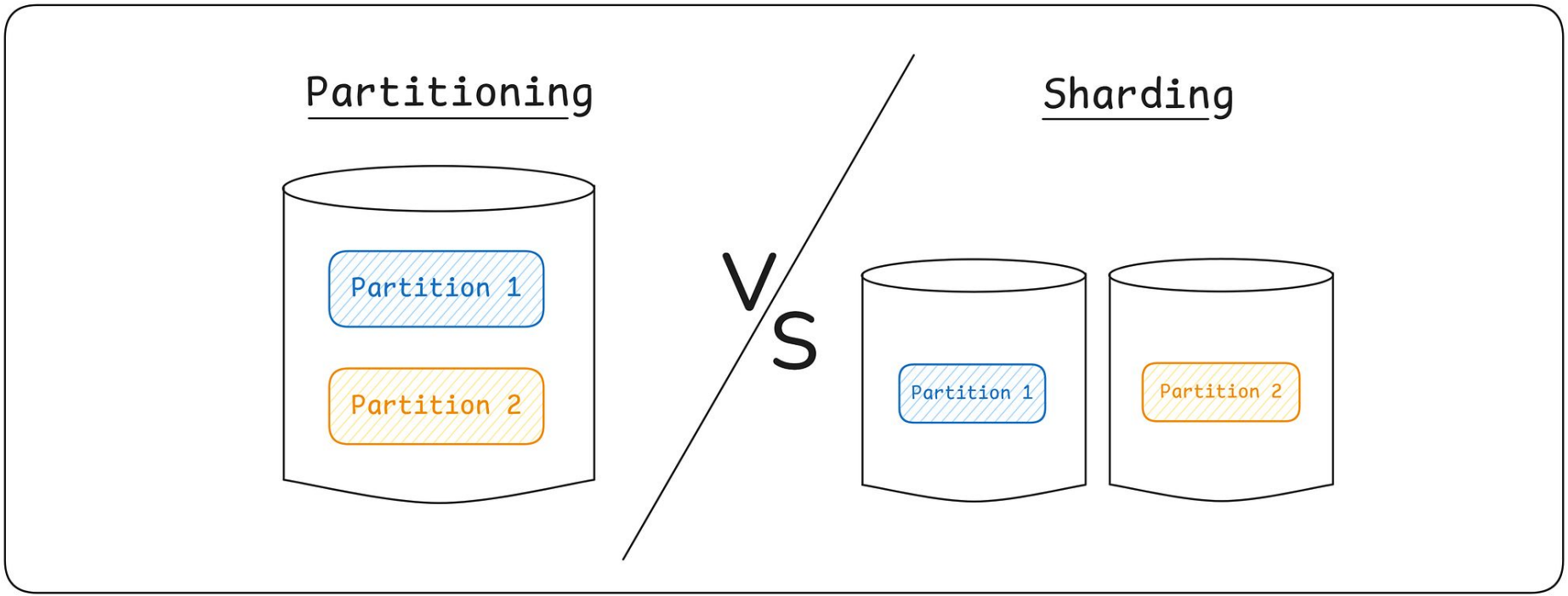
- The shard answers: *Which machine is responsible for this data?*
  - Inside each shard, we could build another index that partitions its data further (local partitions).
  - We will see how we use indexes for smart shard search.

Terminology

- **Routing index:** Small structure used to decide which shards should receive a query.

- **Fanout:** Number of shards contacted for one query.
  - Example: If q is shipped to Shards B and C, fanout = 2.
  - Higher fanout = more shards searched = potentially higher recall, but more network traffic and computation.

Terminology



Sharding Intro

- **TUE Lecture IVF question:** *How do we avoid reading irrelevant vectors?*

- **THU Lecture Sharding Question:** *How do we place and serve the index across machines?*

- Same vocabulary, different level of the system stack.

Sharding Strategy Menu

- **Hash-Sharding:** Simple balance; hard to route selectively.
- **IVF Sharding:** Good routing, may have skew.
- **Metadata Sharding:** Useful filters; could hurt recall.
- **Hybrid Sharding:** Common in practice; more complex routing.

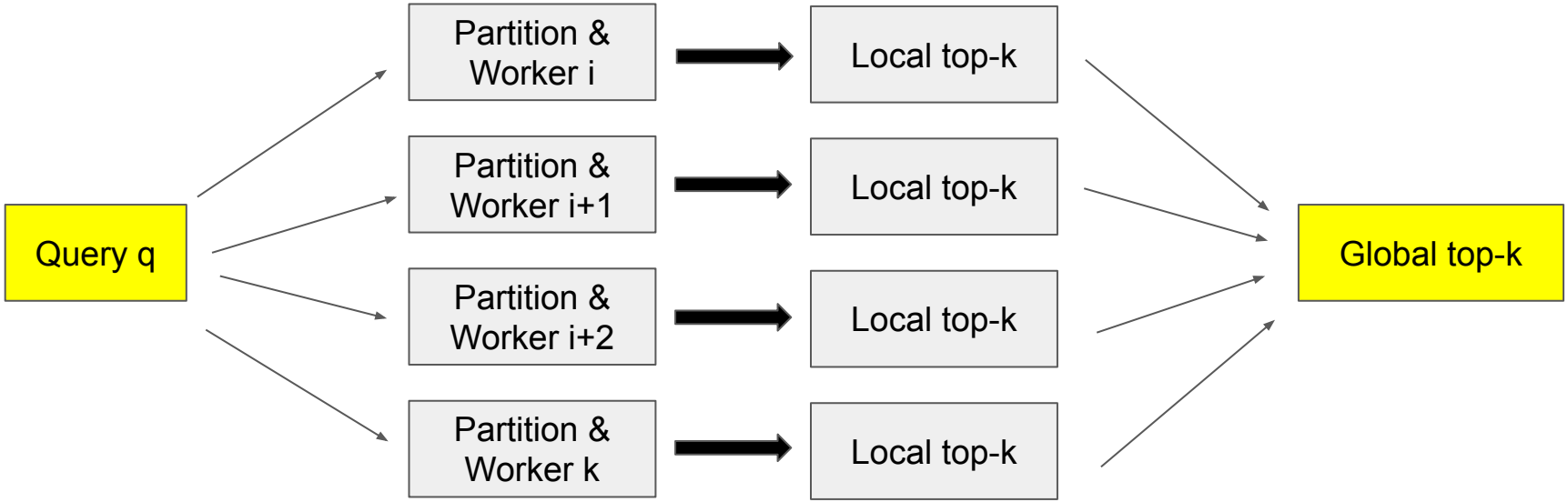
Hash-Sharding

- Assume 12 vectors and 3 shards. Define shard(vec) = hash(vecID) mod 3

Shard 0	Shard 1	Shard 2
v3	v1	v2
v6	v4	v5
v9	v7	v8
v12	v10	v11

Good for balance, but the query has no obvious single shard to visit.

Querying with Hash Shards: Broadcast Query



Hash sharding usually requires fanout to all shards for vector similarity search.

Communication Cost with Local Top-K

- **Assume we have 100 shards with k = 10**

- **Bad plan:** Send all scored vectors from every shard

- **Better plan:** Send only 10 candidates from each shard (1,000 candidates total)

IVF Sharding: Centroids Route Query

- Assume we have 4 centroids C1, C2, C3, C4.

- We determine the k-nearest centroids to our input query q, and simply route q to the selected centroid lists.

Centroids	Physical Location
C1	Shard A
C2	Shard B
C3	Shard C
C4	Shard D

Query q and nprobe = 2:

distance(q, C1) = 0.90
distance(q, C2) = 0.20
distance(q, C3) = 1.10
distance(q, C4) = 0.50

Router launches work only on Shard {B, D}

Naive Filtering vs Indexed Routing

Naive Filtering

Read all vectors

If centroid not in {C2, C4}
    Discard
Else:
    Score vectors

Indexed Routing

Input paths:
    IVF_INDEX/C2/*
    IVF_INDEX/C4/*

C1 and C3 are never read

Filtering in Map saves CPU, physical indexing saves I/O.

Skew: Some Shards get too Much Work

Centroids	Vectors	Centroid probes / min
C1	1M	100
C2	20M	5,000
C3	900K	80
C4	1.2M	20

C2 is both large and popular

Longer local scans
More concurrent queries
Lower latency

Fixing Skew

Split Hot Lists

C2 → C2a, C2b, C2c

Replicate Hot Lists

C2 copy on several machines

Common strategy: create many small partitions, then assign/rebalance them across workers.

Replication: Capacity and Fault Tolerance

- Replication means storing extra copies of selected shards.

- Assume we have Shard C2 copies 1, 2, and 3.

- Having 3 copies means:
  - More read throughput
  - Survive worker failure
  - More memory/storage cost

Routing Is Not Only About Similarity

- So far, routing has been based on the query vector:
  - Which shards should we probe?
  - Should popular shards be replicated?

- But real queries often contain **additional constraints**: “Find similar headphones, but only Sony products under $200.”

- Now routing depends on both: vector similarity + metadata constraints.

Metadata Filters + Vector Search

- Assume a user has a query: “Wireless Headphones”, with metadata filters:
  - Brand = Sony, and Price <= 200$

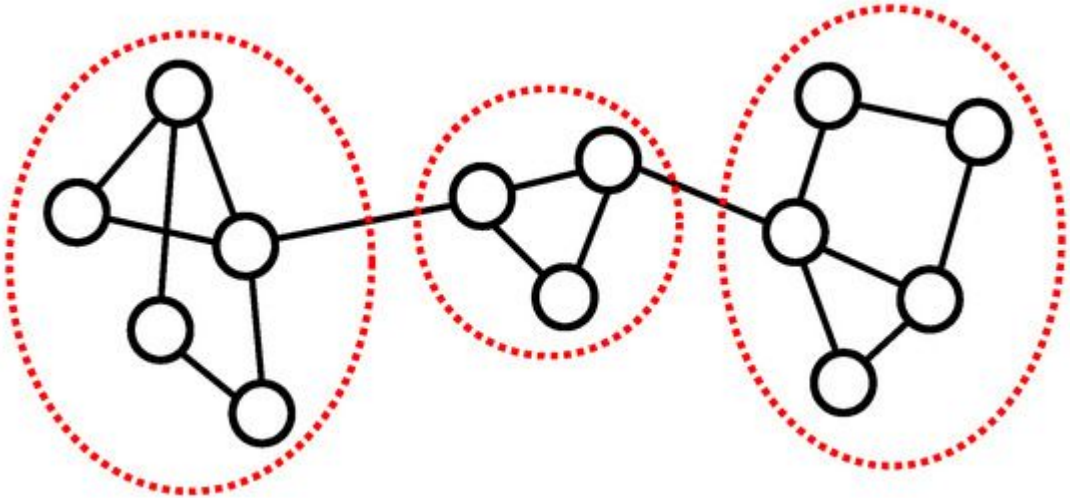
- **What happens if we filter before search?** [potentially] less work, but may need metadata-aware indexes (this not easy nor always logical).

- **What if we filter after vector search?** Simple, but may return too few valid candidates.

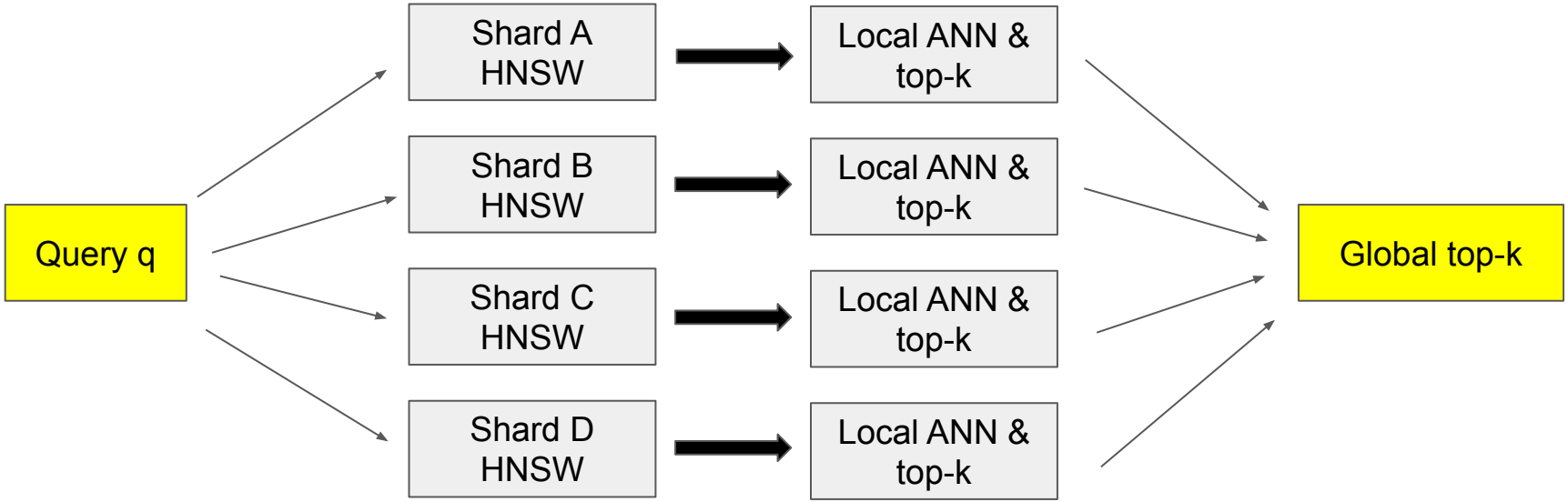
Distributed HNSW and Vector DBs

Distributed HNSW Challenge

- In HNSW, graph edges can connect vectors that live on different machines. A graph traversal may need remote hops unless the system changes the plan.



Approach 1: Independent HNSW per Shard



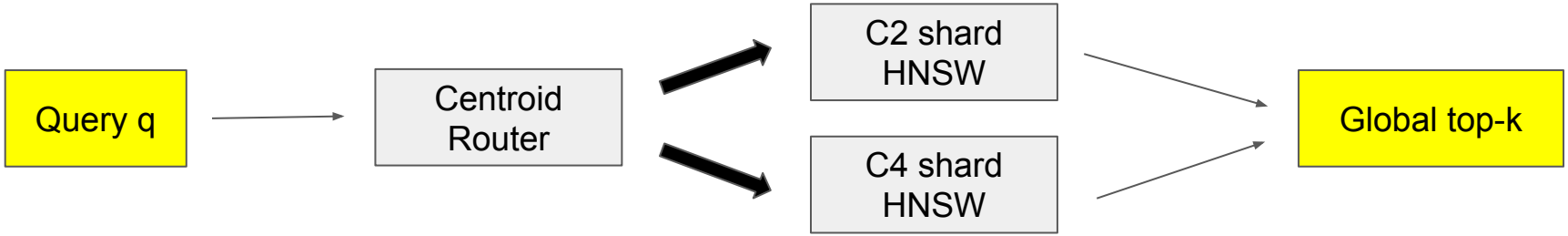
Coordinator merges candidates. Simple and common.

Approach 1: Pros and Cons

Pros	Cons
Easy to scale horizontally	Query may fan out to many shards
No remote graph traversal inside a shard	Global recall can drop if each shard is approximate
Local failures are isolated	More machine participate per query
Works with hash sharding	Coordinator must merge many candidates

Large-scale systems often choose a simple plan that is operationally robust.

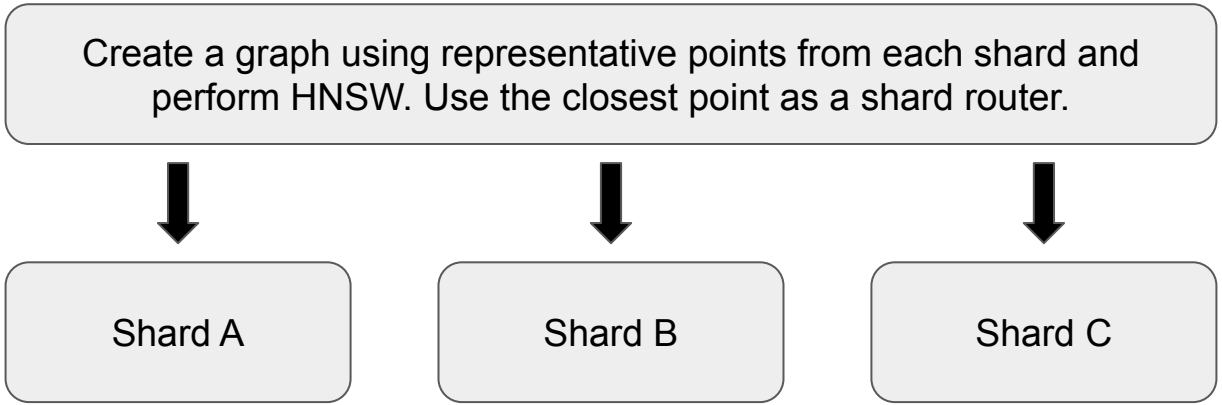
Approach 2: IVF Router + HNSW Inside Selected Shard



This reduces fanout: Route to promising regions, then use HNSW for fast local search.

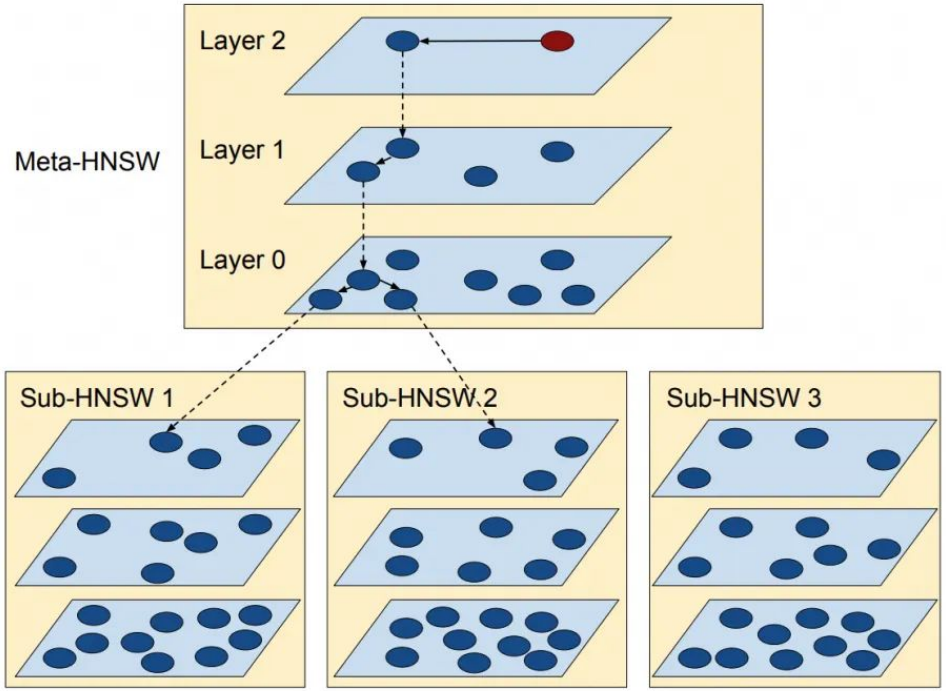
Approach 3: Replicate Coarse Navigation

- Idea = Keep a small routing structure everywhere, but keep full vectors shared.



Use the coarse layer to choose candidate shards. Search full local indexes only where needed. Trade-off: extra memory for less query fanout.

Approach 3: Replicate Coarse Navigation

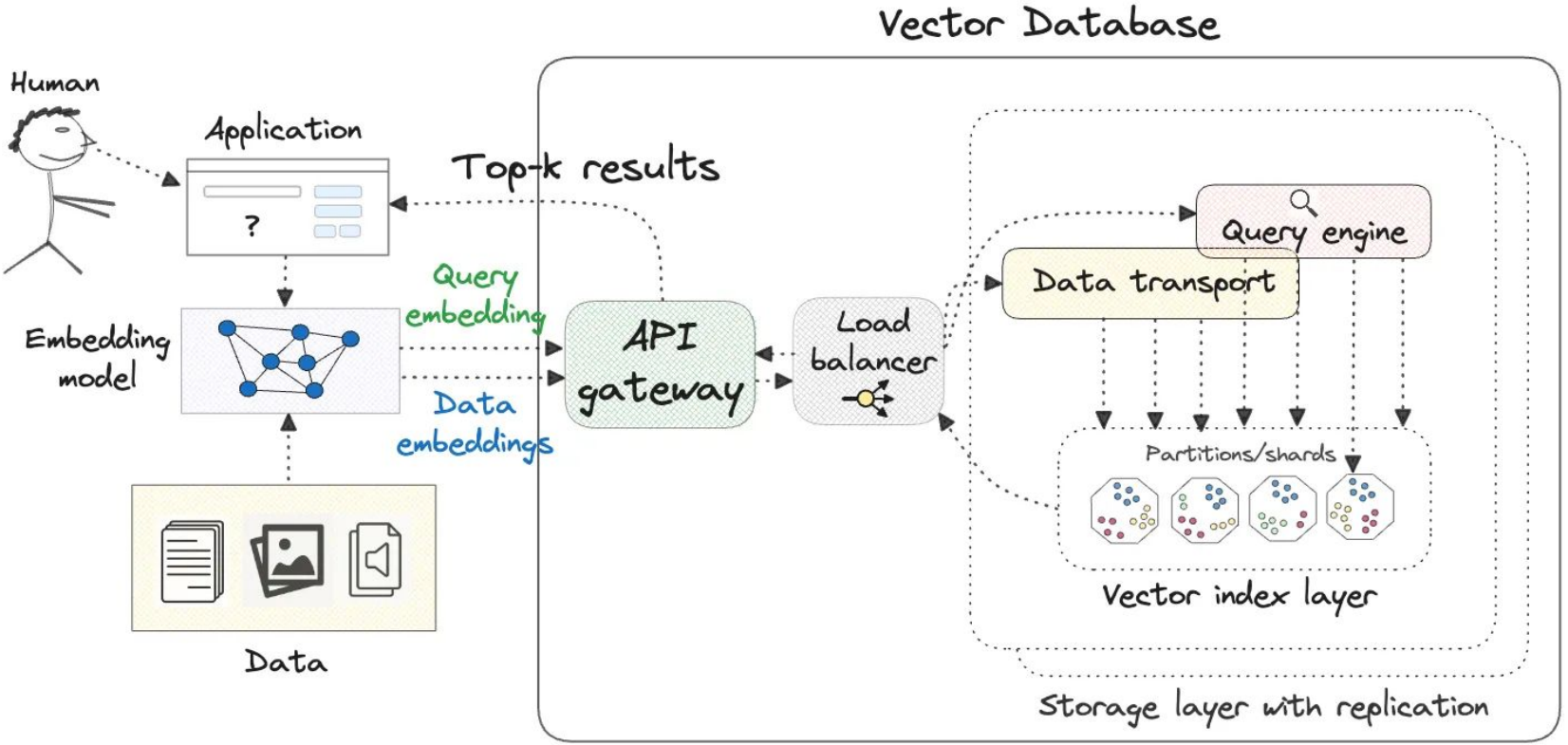


No Free Lunch :(

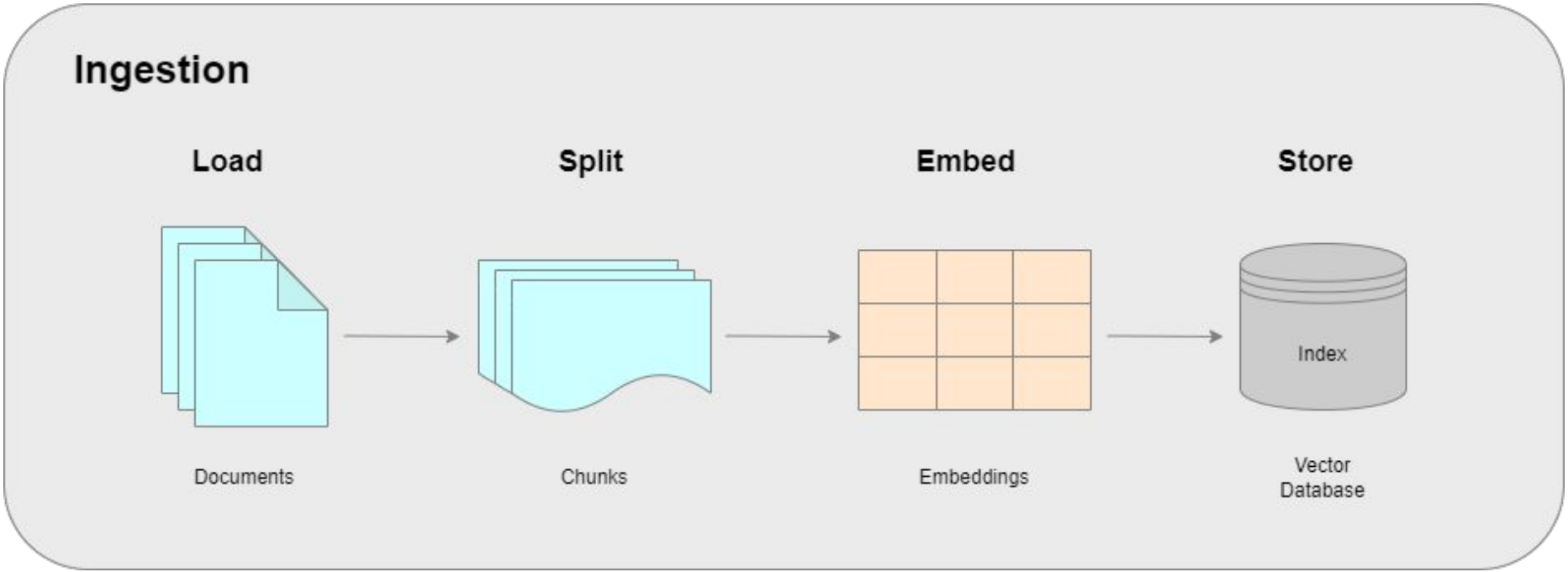
Plan	Query routing	Strength	Weakness
Hash shards	fanout to all shards	balanced placement	high fanout
IVF shards	route to selected centroids	less I/O	boundary and skew
Per-shard HNSW	local graph search	fast local ANN	merge/recall complexity
Hybrid	router + local ANN	practical design	more moving parts

Architecture is workload-dependent: index size, query rate, recall target, and filters.

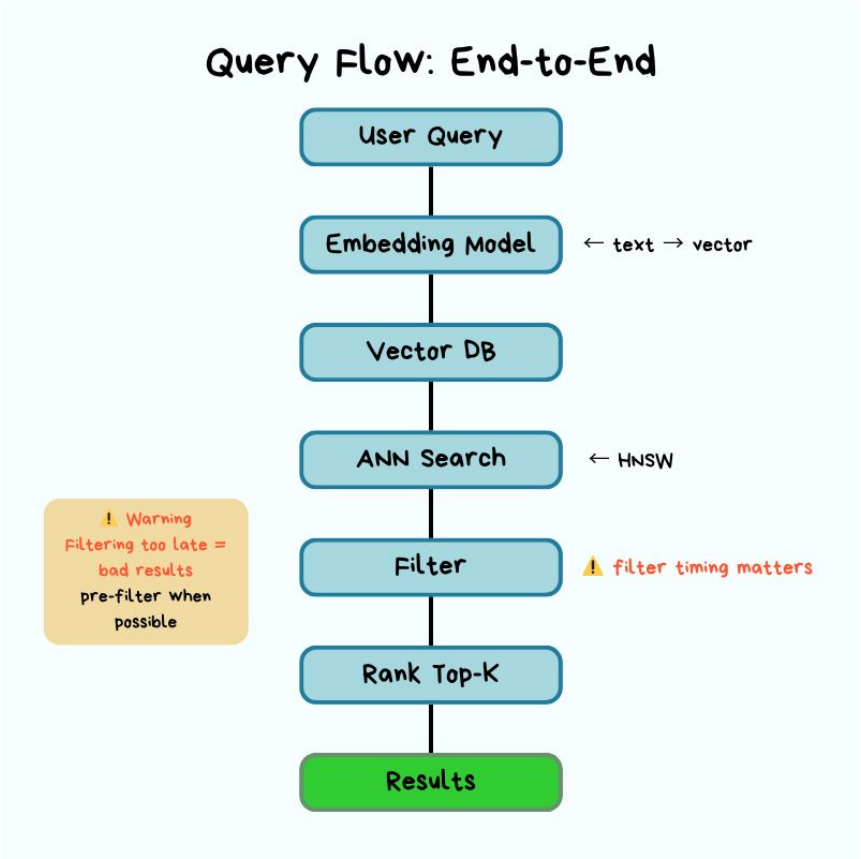
Vector DB Architecture



Offline Query Pipeline



Online Query Pipeline



Latency Budget Example

Stage	Example Time
Embed Query	20ms
Route to Shards	2ms
Local ANN Search	25ms
Network + Merge	8ms
Rerank top candidates	40ms
Total before LLM	95ms

Distributed design is a latency-recall-cost trade-off

Why Rerank?

- First ranking is cheap/scalable; the re-ranking is expensive but more accurate.

- Query: “lightweight waterproof running shoes”. The vector index retrieves:

Product	Vector Similarity
A: Lightweight running shoe	0.94
B: Waterproof hiking shoe	0.92
C: Waterproof running shoe	0.90
D: Lightweight sneaker	0.88

Why Rerank?

- The embedding search ranks them: A > B > C > D

- A stronger re-ranker can look more carefully at the query and product text:
  - A = not waterproof
  - B = not a running shoe
  - C = matches all 3 concepts

- Reranking them gives: C > A > B > D

Common Failure Modes

- **Routing miss:** The correct region or shard was not searched.
- **Local ANN miss:** Local index failed to surface true neighborhood.
- **Merge/rerank issue:** Good candidates were pruned or reranked poorly.
- Approximation can enter at multiple points…

RAG: Retrieval as a System Concept

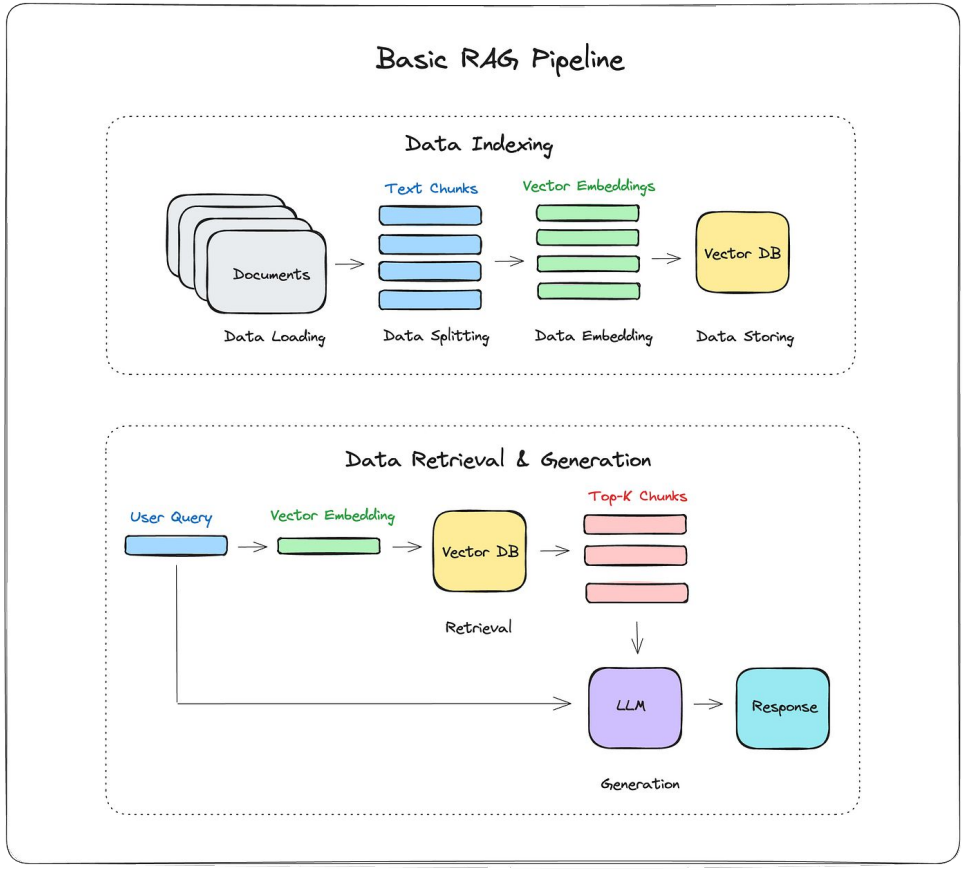
What RAG Adds

- **Vector search alone:** Return similar records to a query.

- **RAG:** *“technique that improves AI language model accuracy by searching an external database for relevant facts and adding them to the user's prompt before generating a response”*

- The retrieval system now affects answer quality, cost, and latency.

RAG Pipeline



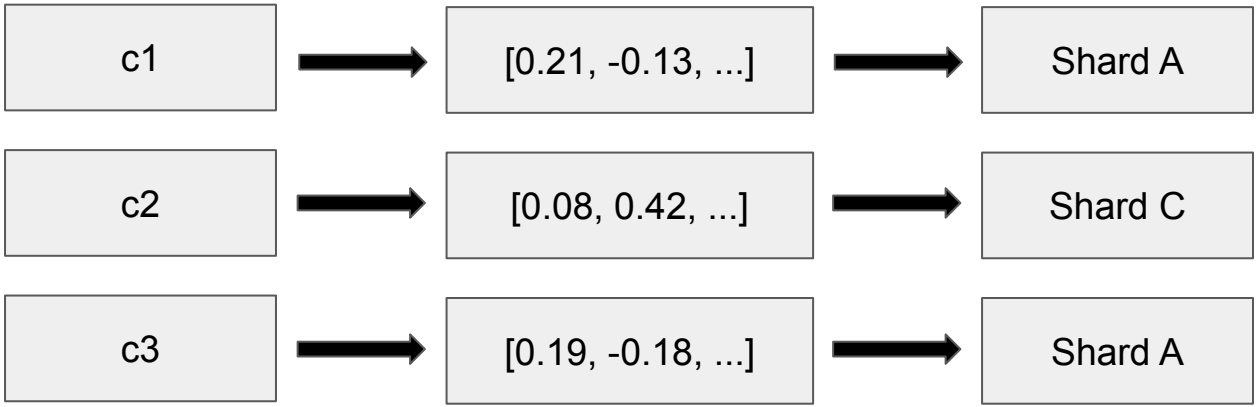
Chunking Example

- Document: “Spark uses lineage to recover lost partitions. RDDs record transformations. When data is lost, Spark recomputes from its lineage.”

Chunk ID	Text
c1	Spark uses lineage to recover lost partitions.
c2	RDDs record transformations.
c3	When data is lost, Spark recomputes from its lineage.

Chunk size affects retrieval quality: Too small loses context; too large wastes tokens.

Embedding and Indexing Chunks



Metadata travels with each vector: Document ID, section, timestamp, etc.

Query-time Retrieval Example

Query

“How does Spark recover lost RDD partitions?”

Retrieve top chunks

Rank	Chunk	Score
1	c1	0.93
2	c3	0.89
3	c2	0.71

Answers uses context

Spark recomputes lost partitions using lineage; the transformations that produced each RDD partition

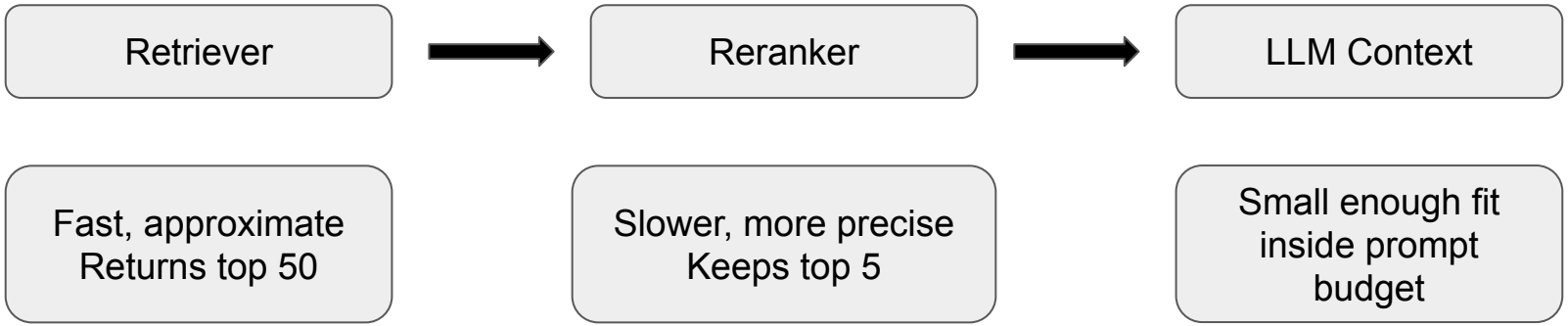
The retrieval result determines what the LLM is able to ground on.

Metadata Filters in RAG

Query	Correct Behavior
“What did the latest assignment say about triangle counting?” Filters: Course = CS6240 Doc_type = HWs Semester = Summ26	Search only documents the user can access. Prefer recent course material. Still rank by semantic similarity.

RAG needs vector similarity + traditional database predicates.

Reranking: Top-K is Often Two-Stage



This mirrors distributed top-k: use a cheap broad stage, then an expensive narrow stage.

Context Assembly

Problem	System Response
Duplicate chunks	Deduplicate by document/section
Too many chunks	Use budgeted top-k and reranking
Chunks out of order	Reconstruct document order when needed
Conflicting sources	Preserve source metadata
Sensitive documents	Apply permissions before generation

Final prompt is a data product assembled by retrieval, ranking, filtering, and formatting.

Evaluating RAG Systems

Layer	System Response
Retriever	Recall@k, latency
Reranker	Latency, cost
Generator	Answer correctness, citation quality
System	Latency, throughput, memory, index freshness

Do not evaluate only the final answer; failures can originate in retrieval, routing, ranking, or generation.

Architecture Checklist

Question	Design implication
Does the index fit in RAM?	single-node vs sharded serving
Is query traffic skewed?	replication and hot-shard splitting
Are filters important?	metadata-aware indexes and routing
Is recall more important than latency?	larger fanout, more local candidates, reranking

Takeaways

- Exact search is the quality baseline but scans everything.
- ANN trades recall for latency and memory efficiency.
- IVF is “learned” partitioning: HNSW is graph search.
- Distributed vector search is local top-k + global merge.

Takeaways

- HNSW uses a randomized sparse hierarchy plus similarity-based neighbor links.
- Memory is a first-order systems constraint for vector search.
- Sharding determines which machines store and search which parts of the index.
- Distributed top-k reduces network traffic by merging local candidates.
- RAG turns vector search into one stage in a larger data-processing pipeline.

References and Cool Papers

- Malkov & Yashunin (2018). HNSW [https://arxiv.org/pdf/1603.09320]
- Johnson, Douze, Jégou (2017). FAISS [https://arxiv.org/pdf/1702.08734]
- Jégou, Douze, Schmid (2010). Product Quantization [https://ieeexplore.ieee.org/document/5432202]
- Guo et al. (2019). ScaNN [https://arxiv.org/pdf/1908.10396]
- ANN-Benchmarks [https://ann-benchmarks.com/]
- DataMListic. (2026). HNSW [https://www.youtube.com/watch?v=77QH0Y2PYKg]

References and Cool Papers

- Javaid (2024). Building a RAG System. [https://medium.com/@aminajavaid30/building-a-rag-system-the-data-ingestion-pipeline-d04235fd17ea]
- Meurer & Kim (2026). What is a Vector Database? [https://newsletter.systemdesign.one/p/what-is-a-vector-database]
- DataQuarry (2023). Understanding Vector Database Internals [https://thedataquarry.com/blog/vector-db-2/]