From Text to Vector

How High Dimensional Data gets Stored, Searched & Retrieved

Narrated by: Jeevan

April 2026

Why This Talk, Why Now

Every team is adding vector search. Few are planning for what happens next.

- LLM layer is commoditising — the moat isn't the model, it's data
- RAG over internal knowledge — Confluence, SharePoint, emails, codebases
- Better tooling, better dev experience, better customer answers

The pattern we keep seeing:

```
Month 1:  "Let's add semantic search!"
          → 100K vectors, works great ✅

Month 4:  "Scale to all our docs"
          → 10M vectors, still fine ✅

Month 8:  "Enterprise rollout"
          → 100M vectors, RAM bill explodes 💸

Month 9:  "Add per-tenant filtering"
          → recall silently drops to 40% 🤷

Month 10: "Maybe we need a vector DB?"
          → now syncing two databases forever
🔄
```

This talk gives the mental model to make these decisions before month 8.

Our Journey Today

By the end — how vector search works under the hood, and what breaks at production scale.

1. The Comparison Problem

How computers understand meaning

2. Searching at Scale

Why brute force breaks and what replaces it

3. Two Index Strategies

IVFFlat vs HNSW — when to use which

4. The Scale Wall

RAM math, quantization, disk-based indexes

5. Production Reality

Filtered search, hybrid search, decision matrix

🟦 Chapter 1: How Do Computers Compare Text?

The fundamental problem: computers don't understand words.

```
"I love fries"   vs   "Fries are great"
```

👴 (Human) instantly similar. 🖥️ (Computer) two different strings.

How do we bridge that gap?

🟦 Embeddings: Text → Numbers That Capture Meaning

The problem: Computers compare numbers easily (32°C vs 30°C = 2°C) but can't compare text.

The trick: Turn text into numbers that capture meaning.

```
"I love fries"      → [0.2, 0.8, 0.1, ... 384 numbers]
"Fries are great"   → [0.3, 0.7, 0.2, ... 384 numbers]
"The sky is blue"   → [0.9, 0.1, 0.8, ... 384 numbers]
```

The magic: Similar meanings → similar numbers!

Analogy: 📍 GPS coordinates for meaning

Just like GPS turns "India Gate" into (28.61, 77.22) and "Rashtrapati Bhavan" into (28.61, 77.19) — close on a map because they're close in Delhi —

an embedding model places "fries" and "great snack" close together in a 384-dimensional "meaning space."

Demo: See It In Action

Similar meaning → similar numbers. Let's prove it:

```
python scripts/compare.py
```

Try these:

```
Text 1: I love fries
Text 2: Fries are great
```

```
Text 1: I love fries
Text 2: The stock market crashed
```

How Do We Measure "Close"?

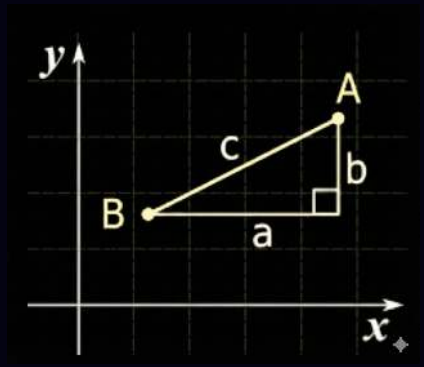
Distance between two points — already familiar:

```
Point A = (x₁, y₁)    Point B = (x₂, y₂)

Distance = √((x₂-x₁)² + (y₂-y₁)²)
```

Same idea, just more dimensions:

```
Vector A = [0.2, 0.8, 0.1, ... 384 nums]
Vector B = [0.3, 0.7, 0.2, ... 384 nums]

Distance = √((0.3-0.2)² + (0.7-0.8)² + ...)
```

That's Euclidean distance. Better options exist for text...



Three Ways to Measure Distance

Method	Formula	Meaning	Best For
Euclidean (L2)	$\sqrt{(\Sigma(a_i - b_i)^2)}$	How far apart?	Images, spatial
Cosine ★	$1 - (a \cdot b)/(\|a\|\|b\|)$	Same direction?	Text (most common)
Inner Product	$-(a \cdot b)$	How aligned?	Normalized vectors

For text search, cosine is king. Cares about direction (meaning), not magnitude (length).

🟦 Putting It All Together

The complete flow:

```
1. User asks: "best crispy fries"
                        ↓
2. Embedding model: [0.72, 0.82, 0.08, ...]
                        ↓
3. Compare against every stored vector using cosine distance
                        ↓
4. Return closest matches:
   "Golden crunchy french fries"       → distance: 0.08  ✅
   "Crispy potato wedges with dip"     → distance: 0.12  ✅
   "How to fix a flat tire"            → distance: 0.95  ❌
```

Key insight: Lower distance = more similar. Always.

This is semantic search. No keyword matching. Pure meaning.

Demo: Semantic Search End-to-End

Pre-seed (run once before the talk):

```
python scripts/seed_demo_docs.py
```

Step 1: What's in the table?

```
SELECT id, content FROM docs_demo;
```

Step 2: Embed a query → get the vector:

```
python scripts/embed_query.py "how to find and fix slow queries"
```

Step 3: Paste vector into SQL → semantic search:

```
SELECT content,
       embedding <=> '<paste_vector_here>'::vector AS distance
FROM docs_demo
ORDER BY distance
LIMIT 3;
```

🟦 Chapter 2: The Scale Problem

Solved search, right? Just compare and return the closest?

...Not quite.

Every query compares against every stored vector:

```
10 docs      ▏                                  instant
1K docs      ▎                                  fast
100K docs    ▉                                  ~okay
1M docs      ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉               😰 seconds
100M docs    ▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉▉ 💀
```

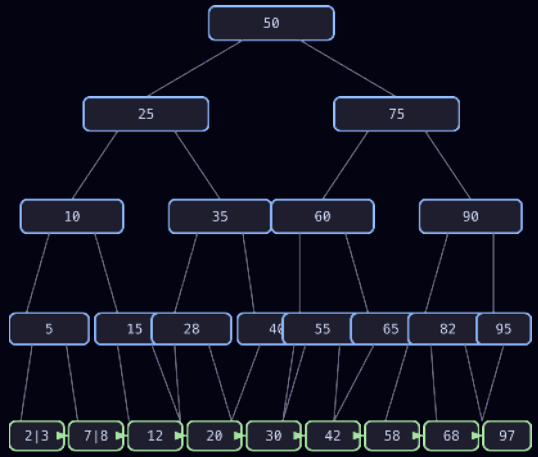
10 docs → 1M docs → 100M docs = 10,000,000x more work. Same query.

Why Traditional Indexes Can't Help

Regular database:

```
SELECT * FROM users WHERE id = 42;
```

B-tree index → binary search. Fast.



B+ Tree — sorted keys, O(log n) lookup
Leaf-linked for range scans. Binary search works because there's a natural order.

Why Traditional Indexes Can't Help (cont.)

But for vectors:

```
SELECT * FROM docs
ORDER BY distance(embedding, query)
LIMIT 10;
```

B-trees need linear ordering. Vectors don't.

```
[0.23, -0.89, 0.45, 0.12, -0.67, ...]
[0.91, 0.03, -0.44, 0.78, 0.15, ...]
[0.17, 0.62, -0.33, -0.51, 0.88, ...]

Sort these? By which number? 🤷
```

If not exact... what if we search approximately?

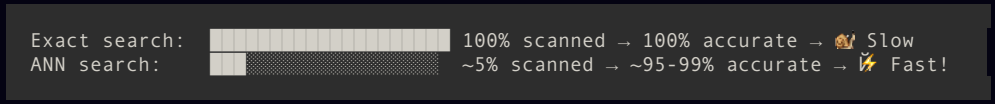
We need a different kind of index entirely.

🟦 The Key Insight: Approximate Is Good Enough

What if we don't need the exact top 10?

What if finding 9 out of 10 true best matches is acceptable?

This is Approximate Nearest Neighbor (ANN) search.



Analogy: 📦 Finding a delivery address in a city

- Exact search: Check every house in every street → hours 🐌
- With pincode: Go to the right area, check nearby streets → minutes ⚡
- The catch: Might miss a house on the border of two pincodes

Chapter 3: Two Index Strategies

Two dominant approaches to ANN indexing.

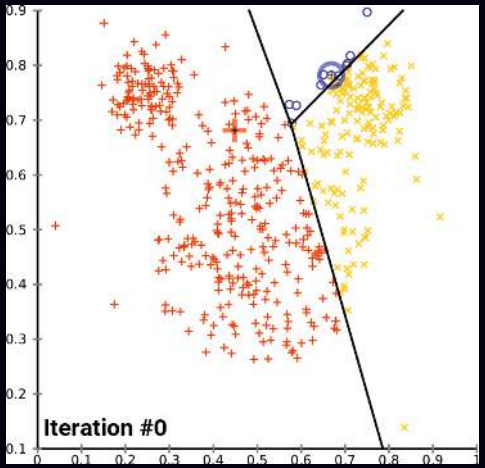
Let's look at each one...

Strategy 1: IVFFlat — Cluster & Search

IVFFlat = Inverted File Index with Flat Storage

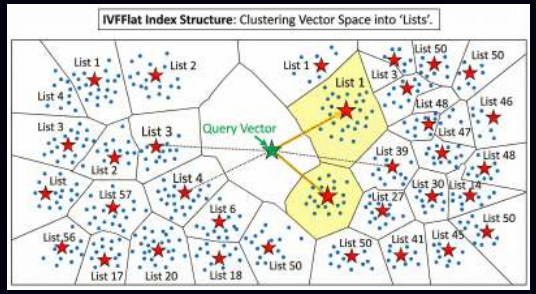
Build: Divide vectors into groups (clusters)

"Supermarket aisles — dairy, snacks, spices"
🛒



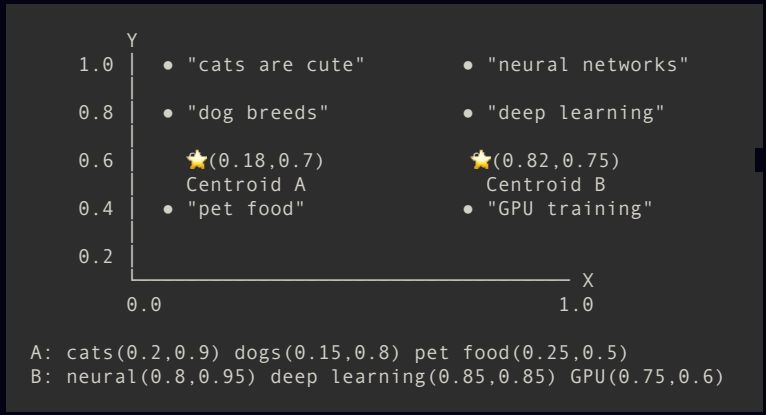
Query: Search only the nearest group(s)

"Need butter? Go to dairy aisle, skip the rest" 🏃



IVFFlat: How Clustering Works

Step 1: Group similar vectors into clusters (k-means)



Step 2: Query → find nearest centroid → search only that cluster

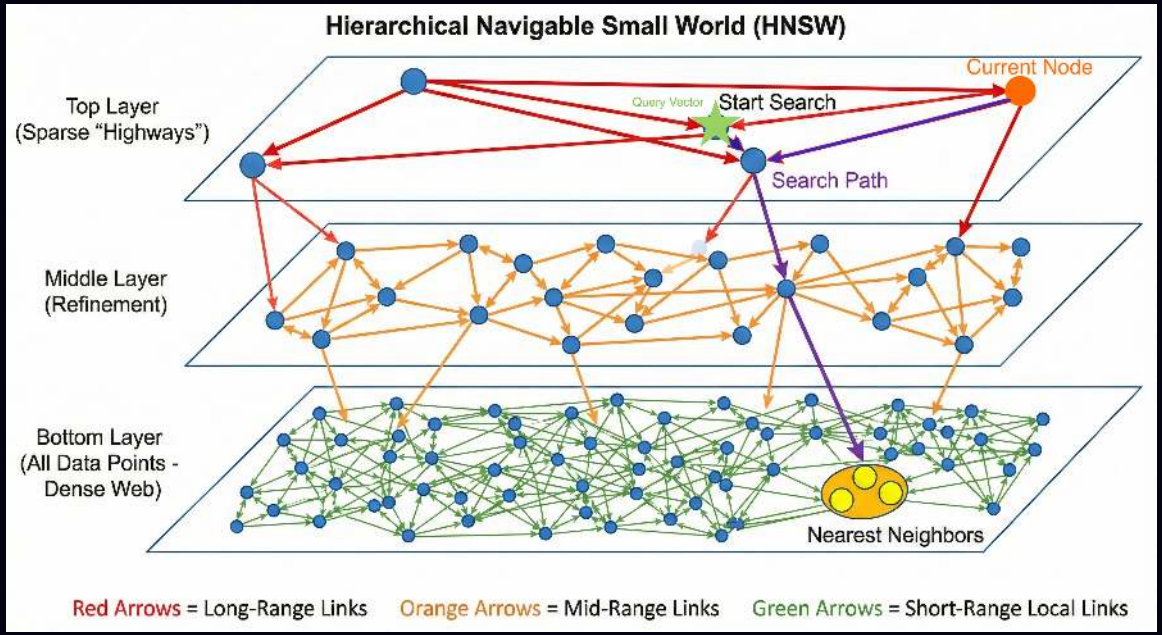
```
Query: "deep learning models" → vector (0.80, 0.78)
  → Nearest centroid: B at (0.82, 0.75)  ✓
  → Search only Cluster B: 3 docs instead of 6!
```

Strategy 2: HNSW — Multi-Layer Graph

HNSW = Hierarchical Navigable Small World

Build a navigable graph with layers. Top = express highways. Bottom = local streets.

"GPS navigation — highways first, then local roads to the destination."



IVFFlat vs HNSW: The Trade-offs

	IVFFlat	HNSW
Speed	Fast (50-100x)	Faster (70-150x)
Accuracy	~90-95% recall	~95-99% recall
Build time	Fast (~30-60s for 50K)	Slower (~2-5 min for 50K)
Index size	Smaller	Larger (graph overhead)
Inserts	Degrades over time	Handles well

Rule of thumb:

- HNSW for production, user-facing apps (search APIs, chatbots, RAG)
- IVFFlat for batch jobs, analytics, write-heavy workloads

⚠️ Critical: Always Use LIMIT!

Without LIMIT, the database can't use ANN indexes.

```
-- ❌ BAD: No target K → can't use ANN index
SELECT * FROM docs
ORDER BY distance(embedding, query)

-- ✅ GOOD: LIMIT = top K → ANN index kicks in
SELECT * FROM docs
ORDER BY distance(embedding, query)
LIMIT 10
```

Why? ANN indexes find the top K nearest — they stop early once they have K candidates. No K = no early stopping = no index.

LIMIT isn't pagination — it tells the algorithm "only need the best 10."

Forgetting LIMIT is like ordering everything on the menu just to pick one dish 🤦

Rule of 37 (from "Algorithms to Live By"): Know the number 37?

Chapter 4: The Scale Wall

ANN indexes are fast. But they assume vectors live in RAM.

Fine at 1M vectors. At 100M? Let's do the math.

The RAM Math

...just the vectors 🤯

```
Per vector:  1536 dims × 4 bytes = 6 KB
```

Scale	Raw Vectors	+ Index Overhead	Approx. RAM Cost
1M	6 GB	~9 GB	~$50/mo
10M	61 GB	~92 GB	~$500/mo
100M	614 GB	~920 GB	~$5,000+/mo
1B	6.1 TB	~9.2 TB	💀

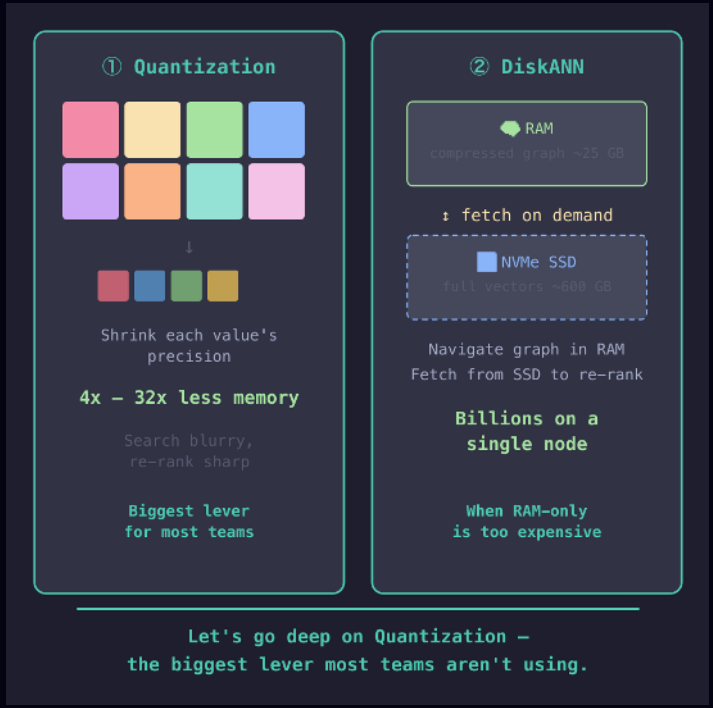
The cliff isn't linear. Going from 64 GB → 920 GB means jumping from a single machine to a distributed cluster.

That's not 15x cost — it's 30-50x operational complexity.

Two Ways Through the Wall

Levers to pull:

1. Quantization: Compress vectors without losing much accuracy
2. Disk-based indexes: Move the index to SSD, keep only compressed data in RAM



Quantization: Compress Smartly

Core idea: Full precision isn't needed for searching. Only for the final ranking of top candidates.

Scalar Quantization

```
FP32 → INT8
[0.2341, -0.8912, 0.4563]
          ↓
[60, -228, 117]
```

4x compression. Like JPEG for vectors.

Binary Quantization

```
Positive → 1, Negative → 0
[0.23, -0.89, 0.45, -0.12]
          ↓
[1, 0, 1, 0]
```

32x compression. Lightning fast (XOR + count).

The production pattern:

```
Query arrives
  ↓
1. Search compressed index (RAM)
   → top 1000 candidates (fast!)
  ↓
2. Fetch full-precision vectors
   for those 1000 only (disk)
  ↓
3. Re-rank with exact distances
   → return true top 10
```

Search blurry, rank sharp.

The blurry copy finds the neighborhood. The sharp original picks the winner.

Demo: Quantization — 32x Compression, High Recall


```
python scripts/quantization_demo.py
```

Beyond RAM: Disk-Based Indexes

When even quantization isn't enough, move the index to SSD.

DiskANN (used in Search Engines for that AI answer section):

```
RAM:  Compressed graph + tiny codes
      (~10-25 GB for 100M vectors)
SSD:  Full-precision vectors
      (fetched only for final candidates)
```

But aren't disk reads slow?

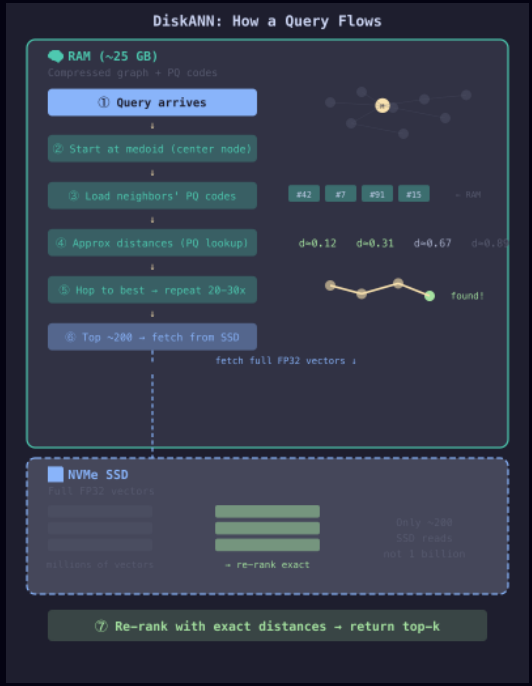
Based on "Six Degrees of Separation" Concept

7B people, ~6 hops to reach anyone

🧓 → 🔵 → 🔵 → 🔵 → 🔵 → 🔵 → ✅

Doubling data = ~1 extra hop, not 2x work

Factor	HNSW	Q-HNSW	DiskANN
RAM (100M)	~920 GB	~30-60 GB	~10-25 GB
Latency	1-5ms	2-8ms	5-15ms
Cost	~$5K/mo	~$500/mo	~$200/mo



The trade-off: Slightly higher latency, massively lower cost.

🔀 Hybrid Search: Best of Both Worlds

Vector search finds meaning. Keyword search finds exact terms.

Query: "how to handle user authentication timeout"

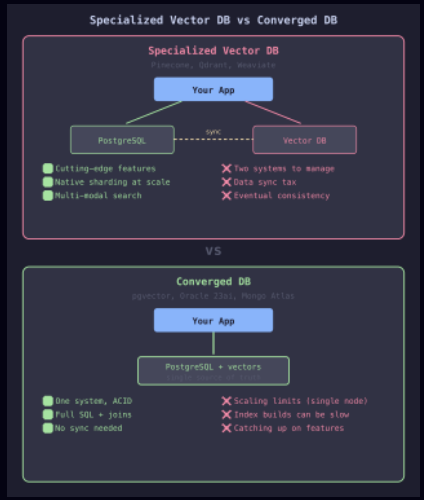
```
Keyword (BM25):   "Authentication timeout error handling guide"
                  → exact match on keywords

Vector search:    "Session expiry and token refresh best practices"
                  → semantically related, different words

Combined → Better recall than either alone
```

Combine with Reciprocal Rank Fusion (RRF):

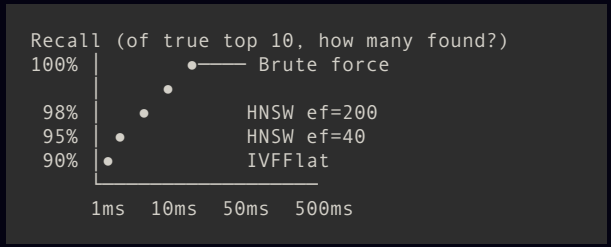
- If two friends both recommend the same restaurant, it's probably good 🍽️

Decision Matrix: What to Use When

Situation	What to Do
< 1M docs	HNSW on existing DB
1-10M, need filters	Partial indexes / partitioning
Cost wall at 10-50M	Quantization (2x-32x)
50M-1B	Disk-based (DiskANN)
Keyword + semantic	Hybrid (BM25 + vectors)
Sub-5ms at 100M+	Specialized vector DB
Already on Mongo/Elastic	Use native vector support



Golden rule: Start with the existing DB. Migrate only when it's outgrown.



Pick the trade-off the product needs.

Key Takeaways

1. Embeddings are GPS coordinates for meaning. Text → numbers. Similar meaning → nearby numbers. That's the whole trick.

2. Approximate search is the unlock. Exact results aren't needed. 95-99% accuracy at 100x speed is the right trade-off.

3. Do the RAM math early. 100M × 1536d = 920 GB. Quantization and disk indexes are the escape hatches.

4. Start with the existing DB. Migrate to a specialized vector DB only when it's outgrown.

5. Measure recall, not just latency. A fast wrong answer is worse than a slightly slower right answer.

6. Filtered search is the hidden production trap. Test real query patterns with filters. Don't assume the index handles it.

The End

Vectors aren't special. Architecture decisions are. 🚀

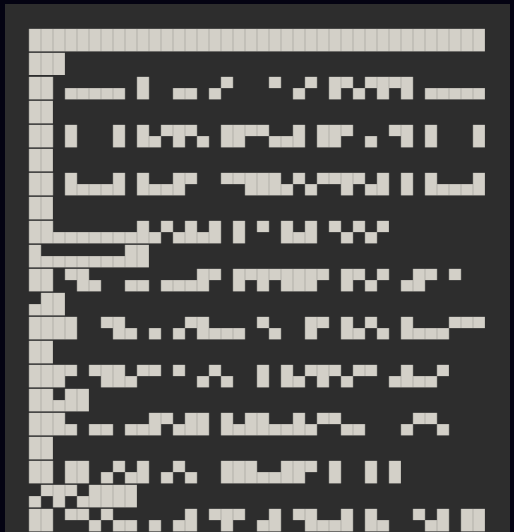
Questions?

📫 Get in touch: jeevan.dc24@alumni.iimb.ac.in

🌐 I write at noobj.me

👷 Part 2: `vector_storage_at_scale.md` in the same repo (Goes deeper: production tuning, architecture decisions, hybrid search patterns — recording coming soon)

Slides & Code:

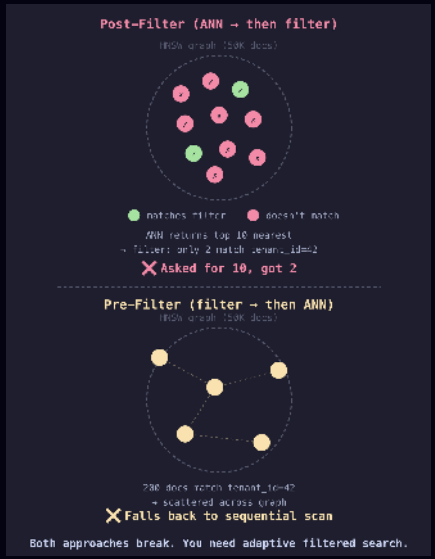


If We Have Time: The Filtered Search Trap

In production, almost never searching the entire database:

```
SELECT * FROM products
WHERE tenant_id = 42 AND category = 'electronics'
ORDER BY distance(embedding, query)
LIMIT 10
```

The vector index is blind to `tenant_id` and `category`.



Post-filter:

```
ANN top 100 → filter → only 2 match!
Asked for 10. Got 2. 😬
```

Pre-filter:

```
Filter to 200 docs → ANN useless
Falls back to brute force.
```

Solutions: iterative scanning, partial indexes, partitioning — but this is the #1 gotcha in production.

```sql
-- Setup metadata on our docs
UPDATE docs_demo SET metadata = jsonb_build_object(
  'category', CASE (id % 3)
    WHEN 0 THEN 'performance' WHEN 1 THEN 'storage' ELSE 'concurrency' END
);
```

```sql
-- Approach 1: iterative_scan (works for any filter)
SET hnsw.iterative_scan = relaxed_order;

EXPLAIN ANALYZE
SELECT content,
  embedding <=> (SELECT embedding FROM docs_demo WHERE id = 1) AS dist
FROM docs_demo
WHERE metadata->>'category' = 'performance'
ORDER BY embedding <=> (SELECT embedding FROM docs_demo WHERE id = 1)
LIMIT 3;
```

Appendix: HNSW — How the Graph Works

The structure — zoom levels:

```
Layer 2 (Express):  A ═══════════ D
                    (2 nodes, long jumps)

Layer 1 (Main):     A ─── C ─── D ─── F
                    (4 nodes, medium
links)

Layer 0 (All):      A — B — C — D — E — F
                    (all 6 nodes, dense)
```

Search for "deep learning":

```
Layer 2: Start at A
  A ═══════════ D
  dist(A)=0.9    dist(D)=0.3 ✓ jump!

Layer 1: At D
  D ─── F
  dist(D)=0.3    dist(F)=0.4
  D still best. Stay.

Layer 0: At D
  D — E — F
  dist(D)=0.3  dist(E)=0.05 🎯
  → E = best match!
```

Like GPS: highway to the right area, then local roads to the exact address.

Appendix: The Restaurant Analogy (Complete)

Sequential Scan (Brute Force):

Walk every street, check every building
🏠 → 🏠 → 🏠 → 🏠 → 🏠 → 🏠

"Is this Italian? No.
Is this Italian? No..."
Check ALL restaurants one by one

Keyword Search (BM25 / TF-IDF):

Look up in phone book
📖 "Italian" → [Addr1, Addr2, Addr3]
🚗 → 🍽️ (direct jump)

Only visit restaurants
labeled "Italian"

IVFFlat (Cluster Search):

Go to the right aisle
🏪 → [Dairy] → 🧀 → 🥛 → 🧈

Search everything in that section
(even yogurt when looking for butter)

HNSW (Graph Navigation):

Highway → Avenue → Street
🛣️ → 🚗 → 🏠 → 🏠

Start at a famous restaurant
Follow recommendations to closer ones

Appendix: Recall vs Latency

Recall = "of the true top 10 results, how many did we actually find?"

The question isn't "best index?" — it's "what does my product need?"

- Internal chatbot / RAG: 95% recall, < 100ms is fine
- Customer-facing search: 90% recall, < 20ms required
- Fraud / compliance: 99%+ recall, latency doesn't matter

Appendix: Quantization Comparison

Method	Compression	Recall (w/ re-rank)	Speed	Training?
FP32 (baseline)	1x	100%	1x	No
FP16 (half)	2x	~99.9%	~1.5x	No
Scalar INT8	4x	~98-99%	~2-3x	No
Product (PQ)	8-64x	~95-99%	~5-10x	Yes
Binary (BQ)	32x	~92-96%	~15-30x	No

Start with FP16 (free 2x win, zero recall loss). Graduate to BQ + re-rank when 32x compression is needed.

Appendix: Vector Index Comparison

	Brute Force	IVFFlat	HNSW	DiskANN
Recall	100% (exact)	90-98%	95-99.9%	95-99%
Latency (1M)	50-500ms	2-10ms	1-5ms	5-15ms
RAM (100M, 1536d)	614 GB	614 GB	~920 GB	~10-25 GB
Build time (1M)	None	~30s	~5 min	~2 min
Incremental insert	N/A	Degrades	Yes	Limited
Best for	< 10K vectors	Batch/analytics	Production	50M-1B