1 Vector Databases

1.1 Two kinds of "find me something"

Traditional databases are built for *exact* questions:

- `WHERE customer_id = 42`
- `WHERE price BETWEEN 10 AND 20`
- `WHERE title LIKE '%refund%'`

B-trees, hash indexes and inverted (full-text) indexes make these fast. But consider:

> *"How do I get my money back after my order arrived broken?"*

The best answer in the knowledge base might be titled *"Return and reimbursement policy for damaged goods."* The two texts share almost no words. Keyword search fails: the *vocabulary mismatch problem*. What we need is search by **meaning**.

Analogy: Library catalogue vs. a knowledgeable librarian

A keyword index is a card catalogue: you must guess the exact subject heading. A librarian who *understands* your question walks you to the right shelf even if you use different words. Vector databases give software that librarian's sense of "nearness in meaning."

1.2 The core idea in one sentence

Key idea

Convert every piece of content into a point in a high-dimensional space (an *embedding*) such that *similar meaning* $\Rightarrow$ *nearby points*. Then "search by meaning" becomes "find the nearest points to the query point." A **vector database** stores these points and finds nearest neighbors *fast*.

1.3 Why LLMs make this urgent

LLMs have three limitations that vector search directly addresses:

1. **Frozen knowledge**: training data has a cutoff date.
2. **No private data**: the model has never seen your company wiki, contracts or tickets.
3. **Finite context window**: you cannot paste a 10,000-page corpus into every prompt.

Solution: *retrieve* only the few most relevant passages at question time, and give them to the model. Retrieval = nearest-neighbor search over embeddings.

2 Embeddings and Similarity

2.1 What is an embedding?

An embedding model (e.g. a transformer encoder) maps text (or images, audio, code, ...) to a vector

$$f: \text{ text} \longrightarrow \mathbf{x} \in \mathbb{R}^d, \qquad d \in \{384, 768, 1024, 1536, 3072, \ldots\}.$$

The model is trained (typically with contrastive learning) so that texts with related meaning get nearby vectors. Individual coordinates are *not* human-interpretable; only the geometry (distances and angles) matters.

Analogy: GPS coordinates for ideas

Latitude/longitude turn a physical place into numbers so that "near on Earth" means "near in numbers." An embedding model does the same for meaning, except the "planet" has 1536 dimensions instead of 2. *Kitten* and *puppy* live in the same "neighborhood"; *truck* lives across town.

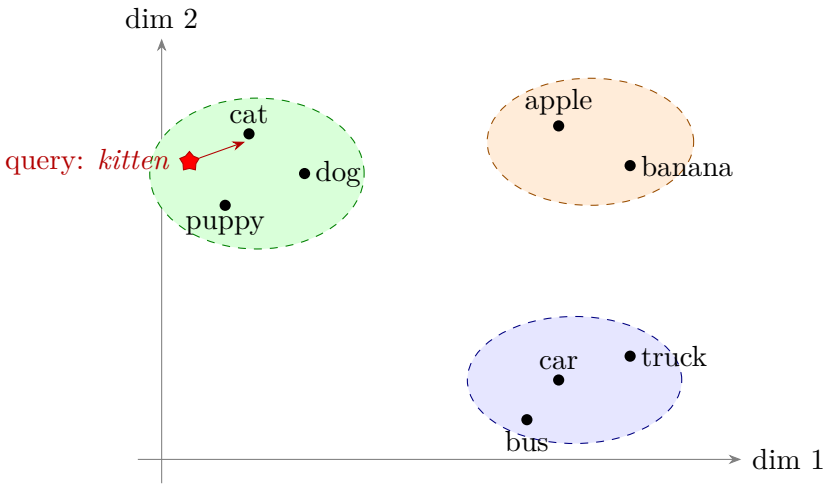


Figure 1: Toy 2-D picture of embedding space. A query is embedded into the same space; its nearest neighbors are semantically related items.

2.2 Similarity metrics

For vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$:

$$\begin{aligned} \text{Euclidean (L2):} \quad & \|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{\textstyle\sum_{i=1}^{d} (a_i - b_i)^2} \\ \text{Dot product:} \quad & \mathbf{a} \cdot \mathbf{b} = \textstyle\sum_{i=1}^{d} a_i b_i \\ \text{Cosine similarity:} \quad & \cos(\mathbf{a}, \mathbf{b}) = \dfrac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|} \end{aligned}$$

Cosine distance is $1 - \cos(\mathbf{a}, \mathbf{b})$.

Practical shortcut

If vectors are normalized to unit length ($\|\mathbf{a}\| = 1$), then

$$\|\mathbf{a} - \mathbf{b}\|_2^2 = 2 - 2\,\mathbf{a} \cdot \mathbf{b} = 2\,(1 - \cos(\mathbf{a}, \mathbf{b})).$$

So L2, dot product and cosine all induce the *same ranking*. Many embedding APIs return unit vectors already; this lets databases use the cheapest operation (inner product).

Analogy: Compass bearing vs. distance

Cosine similarity asks "are we pointing in the same *direction*?" (a short and a long document about cats still point toward "cats"). Euclidean distance additionally cares about *how far apart* the tips of the arrows are.

2.3 The nearest-neighbor problem

Given a collection $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\} \subset \mathbb{R}^d$ and a query $\mathbf{q}$, find the top-k:

$$\mathrm{NN}_k(\mathbf{q}) = \text{the } k \text{ points of } X \text{ minimizing } \mathrm{dist}(\mathbf{q}, \mathbf{x}_i).$$

3 The Brute-Force Wall

3.1 Cost of exact search

Exact ("flat") search compares $\mathbf{q}$ with every stored vector:

$$\text{time per query} \;=\; O(N \cdot d), \qquad \text{memory} \;=\; O(N \cdot d).$$

Quantity (example)	Value
Vectors N	10^6
Dimensions d (e.g. 1536)	1536
Storage as `float32`	$10^6 \times 1536 \times 4 \approx 6.1$ GB
Multiply-adds per query	$\approx 1.5 \times 10^9$
Same at $N = 10^9$	≈ 6 TB, 1.5×10^{12} ops per query

A million vectors is feasible (tens to hundreds of ms on one core); a billion vectors with hundreds of concurrent users is not.

3.2 Why not use a classic index?

- **B-tree**: needs a total order on one key. There is no meaningful ordering of 1536-D points.
- **k-d tree / R-tree**: work well in low dimension (say $d \lesssim 20$). In high dimension they degrade to scanning nearly everything.

Curse of dimensionality. In high dimensions, distances *concentrate*: the nearest and farthest points from a query have almost the same distance, so partitioning space with axis-aligned cuts rarely lets you rule out a region.

Analogy: Phone book for "people who feel like Alice"

A phone book is sorted by last name, which makes "find Smith" instant. But "find the 10 people most similar in personality to Alice" cannot be answered by flipping pages: personality has no alphabetical order, and similarity depends on hundreds of traits at once. You'd have to interview everyone.

The central trade

We give up *exactness* (we may occasionally miss a true neighbor) to gain orders of magnitude in speed. This is **Approximate Nearest Neighbor (ANN)** search. Quality is measured by

$$\text{recall@}k = \frac{|\text{ANN top-}k \ \cap \ \text{exact top-}k|}{k}.$$

Typical production targets are recall 0.95–0.99.

4 How a Vector Database Avoids Brute Force

All ANN methods share one strategy: **spend effort at index-build time to organize the data so that, at query time, most vectors are never touched.** We study three building blocks that are used in practice (and are available in PostgreSQL's `pgvector` or related extensions).

4.1 Method 1: Inverted File Index (IVF) — "partition and probe"

Build. Run k-means on the data to obtain n_{list} centroids. Assign each vector to its nearest centroid; each centroid owns a *list* (cell) of vectors.

Query.

1. Compare $\mathbf{q}$ with the n_{list} centroids (cheap).
2. Pick the n_{probe} closest centroids.
3. Scan *only* the vectors in those cells; return the best k.

Analogy: Supermarket aisles

Instead of walking every shelf for "something like oat milk," you read the aisle signs (centroids), pick the two most promising aisles (probes), and search only there. Fast—unless oat milk was misfiled in an aisle you skipped.

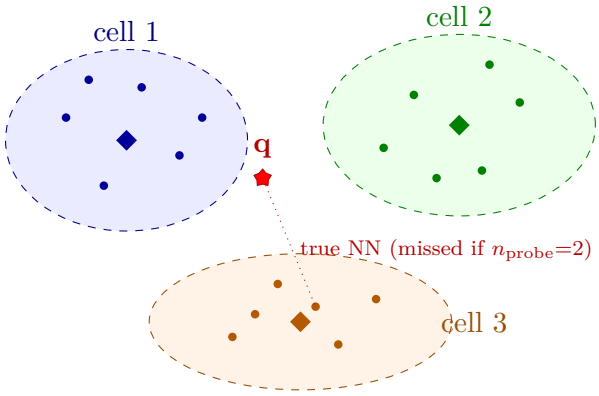


Figure 2: IVF: data is partitioned into cells around centroids (diamonds). The query is compared with centroids first, and only the nearest cells are scanned. Neighbors lying in an unprobed cell are missed, hence recall < 1.

Cost. With $n_{\text{list}} \approx \sqrt{N}$:

$$\text{cost} \approx \underbrace{n_{\text{list}}\, d}_{\text{centroids}} + \underbrace{n_{\text{probe}} \tfrac{N}{n_{\text{list}}} d}_{\text{scanned vectors}} = \sqrt{N}\, d\, (1 + n_{\text{probe}}).$$

For $N = 10^6$, $d = 1536$, $n_{\text{probe}} = 10$: about 1.7×10^7 multiply-adds versus 1.5×10^9 for brute force — roughly a **90**× reduction.

Knobs. More lists ⇒ smaller cells but more boundary effects. More probes ⇒ higher recall, slower queries.

Weakness. Needs training data for k-means (build the index *after* loading data), and cells drift as data changes.

4.2 Method 2: HNSW — "navigate a small-world graph"

Hierarchical Navigable Small World graphs are the default high-performance index in most vector databases.

Build. Each vector becomes a node connected to its approximate M nearest neighbors. Nodes are randomly promoted to higher layers (exponentially fewer per layer, like a skip list). Top layers are sparse "highways"; layer 0 contains every node.

Query.

1. Enter at the top layer. Greedily hop to whichever neighbor is closer to **q** until no improvement.
2. Drop one layer, continue greedy search from that node.
3. At layer 0, run a best-first search keeping a candidate list of size ef_{search}; return the top k.

Expected cost grows roughly **logarithmically** in N.

Analogy: Zooming a map: flight → highway → street

To reach a specific café in a foreign city you don't walk every street. You fly to the right country (top layer: few long links), drive on highways to the right district (middle layer), then walk local streets (layer 0: dense short links). Each stage narrows the area dramatically.

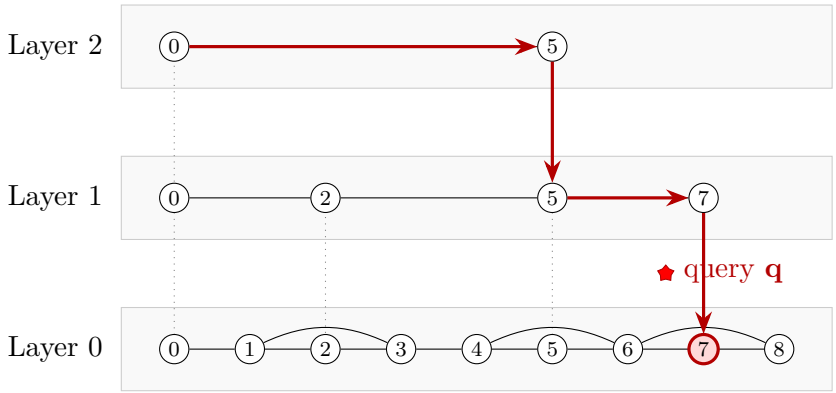


Figure 3: HNSW search: enter at the sparse top layer, hop greedily toward the query, descend, and refine in the dense bottom layer. Red path = nodes visited; the vast majority of nodes are never touched.

Knobs.

Parameter	Effect
M	Links per node. Larger: better recall and more memory.
$ef_{\text{construction}}$	Candidate list size while building. Larger: better graph, slower build.
ef_{search}	Candidate list size at query time. Larger: higher recall, higher latency.

Strengths. Excellent recall/latency trade-off; no training step; supports incremental inserts. **Weaknesses.** Memory hungry (vectors *plus* graph links); slower, heavier builds; deletes leave "holes" that need maintenance.

4.3 Method 3: Product Quantization (PQ) — "compress the vectors"

Even if we visit few vectors, storing $N \times d \times 4$ bytes is costly. PQ compresses each vector:

1. Split the d-dim vector into m sub-vectors of dimension d/m.
2. For each sub-space, learn a codebook of 256 centroids with k-means.
3. Replace each sub-vector by the 1-byte ID of its nearest centroid.

A 1536-D `float32` vector (6144 bytes) with $m = 192$ becomes **192 bytes** (32× smaller). Distances are approximated using lookup tables of precomputed query-to-centroid distances.

Analogy: Describing colors with a paint-chip book

Instead of storing the exact RGB value of every wall in a city, you store "chip #17" from a shared 256-color chip book. You lose a little accuracy but save enormous space—and comparing two chip numbers is just a table lookup.

Related options: scalar quantization (float32 → int8, 4×), binary quantization (1 bit per dimension, 32×), and half-precision (`float16`). Common pattern: search quantized vectors quickly, then *re-rank* the top candidates with full-precision vectors.

4.4 Comparing the methods

	Flat (exact)	**IVF**	**HNSW**
Idea	Compare with all	Partition, probe few cells	Greedy graph walk over layers
Query cost	$O(Nd)$	$\approx O(\sqrt{N}\, d\, n_{\text{probe}})$	$\approx O(\log N)$ hops
Recall	100%	Tunable via probes	Tunable via ef_{search}
Build	None	k-means training	Slow graph construction
Memory	Nd	$\approx Nd$ + centroids	Nd + links
Updates	Trivial	OK, centroids go stale	Good; deletes awkward

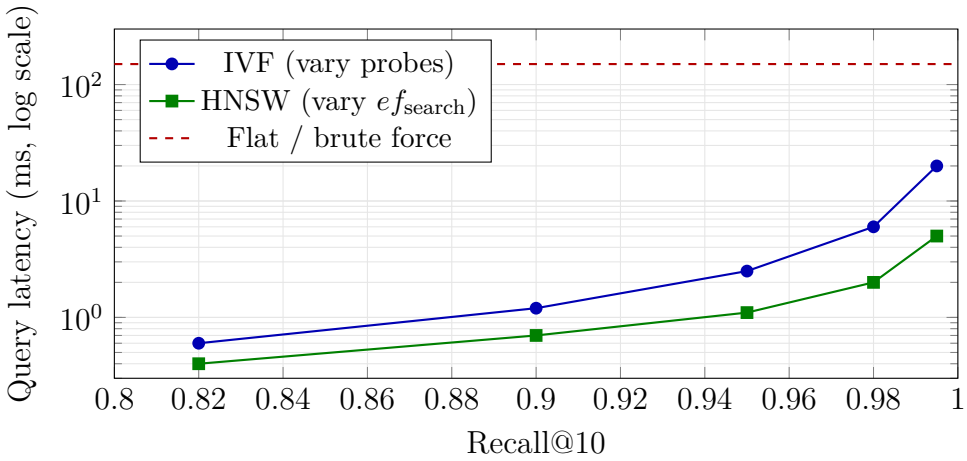


Figure 4: *Schematic* (not measured data): the recall–latency trade-off. Moving right (more recall) costs more time; both ANN families are far below brute force until recall approaches 100%.

5 Text Chunking

5.1 Why chunk?

- Embedding models have an input limit (often 512–8192 tokens).

- One vector per 50-page document *blurs* many topics into one average meaning, so retrieval becomes imprecise.
- LLM context is limited and costly: we want to send only the relevant paragraph, not the whole manual.

Analogy: Index cards for a textbook

Nobody studies by photocopying a whole textbook for every question. You cut it into index cards, one idea each, and pull the right cards. Cards that are too big are unfocused; cards that are too small lose context ("it was lower" — *what* was lower?).

5.2 Chunking strategies

Strategy	How it works	Trade-off
Fixed size	Every T tokens (e.g. 300–500)	Simple, but cuts mid-sentence
Fixed size + overlap	Consecutive chunks share ≈10–20%	Preserves boundary context; more storage
Recursive / structural	Split by section → paragraph → sentence, respecting headings, code blocks, tables	Best default for documents with structure
Semantic	Split where embedding similarity between adjacent sentences drops	Coherent chunks; more compute, less predictable size
Parent–child	Embed small chunks, return the larger parent passage to the LLM	Precise retrieval + rich context

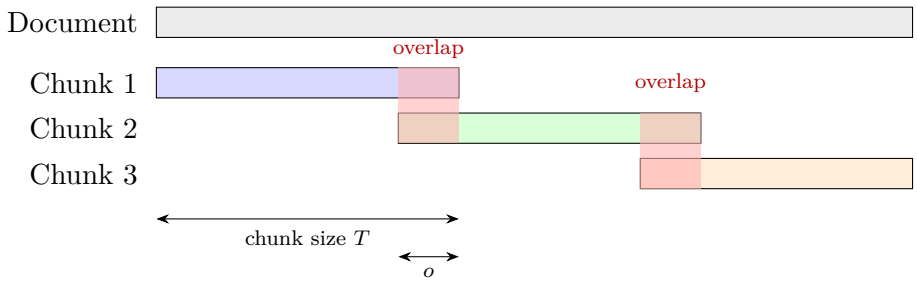


Figure 5: Sliding-window chunking with overlap o. A sentence that falls on a boundary appears whole in at least one chunk.

5.3 Practical guidance

- Start with ∼300–500 tokens and 10–15% overlap; **tune using retrieval evaluation**, not intuition.
- Store *metadata* with each chunk (source document, section title, page, date, access-control labels). It enables filtering and citations.
- Prepend context if needed (e.g. document title or section heading) so a chunk is self-explanatory.

- Match the chunker to the content: code (by function), legal text (by clause), chat logs (by conversation turn), tables (keep rows with headers).

Trade-off to state clearly

Small chunks ⇒ precise matches, but missing context. Large chunks ⇒ rich context, but diluted embeddings and more irrelevant tokens in the prompt. There is no universally correct size.

6 Semantic Search with a GPT: Retrieval-Augmented Generation

6.1 The pipeline

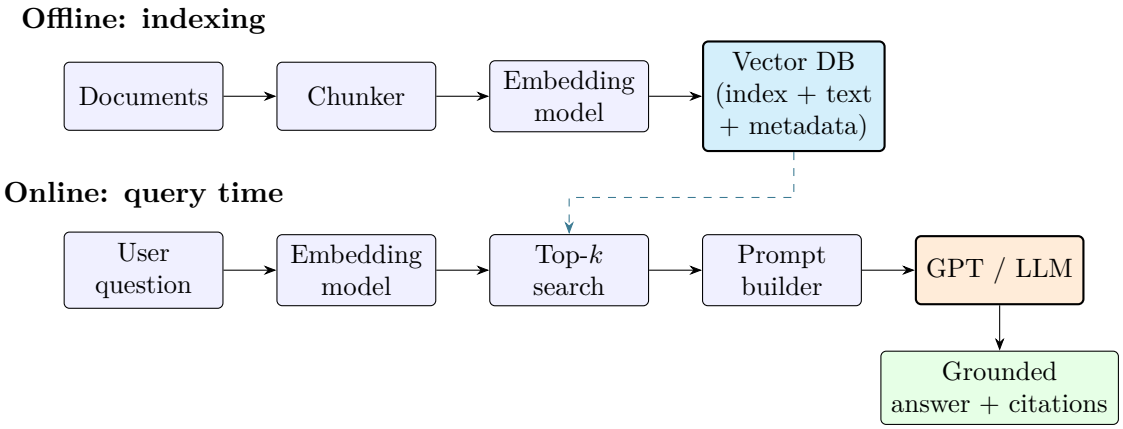


Figure 6: RAG architecture. The same embedding model must be used offline and online.

Analogy: Open-book exam

A bare LLM takes a closed-book exam from memory. RAG hands it the exact textbook pages just before it answers. The vector database is the student's fast, meaning-aware index at the back of the book; the LLM is the student who reads the pages and writes the answer.

6.2 Step by step

Offline (once, then incrementally):

1. Load and clean documents; chunk them.
2. Embed each chunk: $\mathbf{x}_i = f(\text{chunk}_i)$.
3. Insert $(\text{id}, \text{text}, \mathbf{x}_i, \text{metadata})$ into the database; build the ANN index.

Online (per question):

1. Embed the question: $\mathbf{q} = f(\text{question})$ — *same model*.
2. Retrieve top-k chunks (e.g. $k = 5$–20), optionally with metadata filters.

3. (Optional) **Re-rank** candidates with a stronger cross-encoder model.
4. Build the prompt: instructions + retrieved chunks + question.
5. Call the LLM; return answer with source citations.

A typical prompt template:

```
You are a support assistant. Answer ONLY from the context below.
If the answer is not in the context, say you don't know.

# Context
[1] <chunk text 1> (source: returns_policy.pdf, p.3)
[2] <chunk text 2> (source: faq.html#damaged)

# Question
How do I get my money back after my order arrived broken?
```

6.3 What the vector database contributes (and what it doesn't)

- **Does:** finds semantically related passages in milliseconds across millions of chunks; keeps knowledge fresh (update the database, not the model); allows per-user access control via filters; enables citations.
- **Does not:** guarantee the retrieved chunk actually contains the answer, nor that the LLM uses it faithfully. RAG *reduces* hallucination; it does not eliminate it.

6.4 Common improvements

- **Hybrid search**: combine vector similarity with keyword (BM25 / full-text) ranking; helps with exact identifiers, names and rare terms where embeddings are weak.
- **Re-ranking**: retrieve 50 cheaply, re-score with a cross-encoder, keep 5.
- **Query rewriting**: have the LLM rewrite vague or conversational questions before embedding.
- **Evaluation**: measure recall@k of retrieval *separately* from answer quality.

7 PostgreSQL in Practice: `pgvector`

7.1 Why PostgreSQL?

`pgvector` is an extension that adds a `vector` type, distance operators, and ANN indexes (IVFFlat and HNSW) to PostgreSQL. The pitch: **keep embeddings next to your relational data**, with transactions, backups, joins, row-level security and SQL you already know, instead of operating a separate system.

Distance operators in pgvector

Operator	Meaning
`<->`	Euclidean (L2) distance
`<=>`	Cosine distance ($1 - \cos$)
`<#>`	Negative inner product (negated so `ORDER BY ... ASC` works)

7.2 Schema and ingestion

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE documents (
  id BIGSERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  source_url TEXT,
  created_at TIMESTAMPTZ DEFAULT now()
);

CREATE TABLE chunks (
  id BIGSERIAL PRIMARY KEY,
  document_id BIGINT REFERENCES documents(id) ON DELETE CASCADE,
  chunk_index INT NOT NULL,
  content TEXT NOT NULL,
  metadata JSONB DEFAULT '{}',
  embedding VECTOR(1536) NOT NULL -- must equal the model's dimension
);

-- Insert (the embedding comes from your application / embedding API):
INSERT INTO chunks (document_id, chunk_index, content, embedding)
VALUES (1, 0, 'Damaged items can be reimbursed within 30 days...',
        '[0.012, -0.034, ... ]');
```

7.3 Exact search first

With no index, PostgreSQL does a sequential scan — an exact, brute-force top-k (recall is 100%):

```
-- $1 is the query embedding produced by the same model
SELECT id, content, embedding <=> $1 AS cosine_distance
FROM chunks
ORDER BY embedding <=> $1
LIMIT 5;
```

This is perfectly fine up to tens or hundreds of thousands of rows, and is the **ground truth** to compare approximate indexes against.

7.4 Adding an ANN index

```
-- HNSW (graph index)
```

```sql
CREATE INDEX chunks_hnsw_idx ON chunks
  USING hnsw (embedding vector_cosine_ops)
  WITH (m = 16, ef_construction = 64);

-- Query-time recall/speed knob:
SET hnsw.ef_search = 100;

-- Alternative: IVFFlat (partition-and-probe)
CREATE INDEX chunks_ivf_idx ON chunks
  USING ivfflat (embedding vector_cosine_ops) WITH (lists = 1000);
SET ivfflat.probes = 10;
```

Notes for students:

- The *operator class* (`vector_cosine_ops`) must match the operator used in `ORDER BY` (`<=>`), otherwise the planner will not use the index.
- IVFFlat should be built *after* loading representative data, since k-means needs it; `lists` $\approx \sqrt{N}$ for large tables is a common starting point.
- Use `EXPLAIN ANALYZE` to confirm an index scan is used.
- Check `pgvector` documentation for current limits (e.g. maximum indexable dimensions, half-precision and binary-quantized types), which evolve between releases.

7.5 Semantic search + metadata + keywords in one SQL statement

```sql
-- Filtered semantic search (e.g. only recent policy documents)
SELECT c.content, d.title, c.embedding <=> $1 AS dist
FROM chunks c
JOIN documents d ON d.id = c.document_id
WHERE d.created_at > now() - interval '1 year'
  AND c.metadata->>'type' = 'policy'
ORDER BY c.embedding <=> $1
LIMIT 5;
```

Caveat: with ANN indexes, filters are often applied *after* the index returns its candidates, so a very selective filter can return fewer than k rows. Mitigations: raise `ef_search`, use iterative index scans (available in recent `pgvector` versions), partition the table, or add a B-tree/partial index on the filter column.

```sql
-- Hybrid: combine full-text rank and vector similarity (sketch)
WITH sem AS (
  SELECT id, row_number() OVER (ORDER BY embedding <=> $1) AS r
  FROM chunks ORDER BY embedding <=> $1 LIMIT 50),
kw AS (
  SELECT id, row_number() OVER (
          ORDER BY ts_rank(to_tsvector('english', content),
                           plainto_tsquery('english', $2)) DESC) AS r
  FROM chunks
  WHERE to_tsvector('english', content) @@ plainto_tsquery('english', $2)
  LIMIT 50)
SELECT id, sum(1.0 / (60 + r)) AS rrf_score -- Reciprocal Rank Fusion
FROM (SELECT * FROM sem UNION ALL SELECT * FROM kw) t
GROUP BY id ORDER BY rrf_score DESC LIMIT 5;
```

8 Benefits, Downsides, and Wrap-up

8.1 Benefits

- **Meaning-based retrieval**: handles synonyms, paraphrases, typos, and even cross-lingual queries.
- **Scale and speed**: sub-linear query time (ms over millions of vectors).
- **Fresh, private knowledge for LLMs** without retraining; supports citations and access control.
- **Multimodal**: the same machinery searches images, audio, code, products, and recommendations.
- **Cost control**: sends only relevant context to the LLM, cutting token usage.

8.2 Downsides and risks

- **Approximate results**: ANN can miss true neighbors; recall must be *measured*.
- **Memory cost**: high-dimensional float vectors and graph indexes are RAM-hungry; quantization trades accuracy for space.
- **Embedding-model lock-in**: changing the model changes the geometry; you must *re-embed everything* and rebuild the index. Query and corpus embeddings must come from the same model.
- **Weak on exact matching**: part numbers, names, negation ("not covered") and numeric constraints are poorly captured; hybrid search is usually needed.
- **Chunking and quality sensitivity**: bad chunks $\Rightarrow$ bad retrieval $\Rightarrow$ confidently wrong answers ("garbage in, garbage out").
- **Operational complexity**: index build times, tuning, filtered search behavior, updates/deletes, ingestion pipelines.
- **Opacity**: "why did this rank first?" is hard to explain compared with keyword matching.
- **Security**: embeddings can leak information about their source text; retrieved text can contain prompt-injection content that the LLM may follow.

8.3 PostgreSQL/`pgvector` versus a dedicated vector database

	PostgreSQL + `pgvector`	**Dedicated vector DB**
Strengths	One system; ACID transactions; joins and filters; familiar tooling	Purpose-built scaling, sharding, advanced ANN/quantization options
Limits	Single-node scaling model; fewer index variants; tuning is on you	Another system to run; data duplication; weaker relational features
Good fit	Up to millions of vectors; apps already on Postgres	Very large (100M–billions) or latency-critical workloads

8.4 Summary

1. Embeddings turn *meaning* into *geometry*; search becomes nearest-neighbor lookup.
2. Exact search costs $O(Nd)$ and does not scale; classic indexes fail in high dimension.

3. ANN indexes (IVF, HNSW, PQ) restructure data *in advance* so queries touch only a tiny fraction of it, trading a little recall for orders-of-magnitude speed.
4. Chunking determines what a "unit of meaning" is; it often matters more than the choice of database.
5. RAG = embed $\rightarrow$ retrieve top-k $\rightarrow$ stuff into prompt $\rightarrow$ generate. The vector DB is the memory; the LLM is the reasoner.
6. PostgreSQL with `pgvector` is a pragmatic first choice; measure recall and latency before scaling out.