Vector Databases (for RAG)

The core problem RAG solves

When a user asks a question, you want to fetch the **most relevant chunks** from your knowledge base.

Two big requirements:

- **Semantic match** (meaning, not exact keywords)
- **Fast retrieval** at scale

Vector DBs are built for this.

What is a vector (embedding)?

A chunk of text → converted into a list of numbers:

- `embedding = [0.12, -0.03, 0.88, ...]`
- Each chunk becomes a point in a high-dimensional space
- Similar meaning → points are closer

Retrieval becomes: **find nearest vectors** to the query vector.

What is a Vector Database?

A database optimized to store and query:

- **Vectors (embeddings)**
- **IDs**
- **Metadata (JSON fields)**
- Sometimes the **raw text** too

Main feature:

- **ANN search** (Approximate Nearest Neighbors)
- Returns top-K most similar vectors quickly

What vector search returns

Given:

- query embedding `q`
- stored embeddings `{v1, v2, v3...}`

Vector search returns:

- `top_k` matches
- each with a **similarity score** (cosine / dot / L2)
- plus associated metadata + chunk text

Why “Approximate” Nearest Neighbor?

Exact nearest neighbor search is expensive at scale.

Vector DBs use indexing structures:

- HNSW (graph-based)
- IVF (cluster-based)
- PQ (compression)

Trade-off:

- Slight loss in recall
- Huge gain in speed + cost

Why not SQL?

SQL is great for:

- Exact matches
- Range queries
- Joins
- Aggregations

But SQL is NOT designed for:

- Finding the 20 most semantically similar chunks to this sentence
- Efficient ANN indexes
- Similarity search over high-dimensional float arrays at scale

What about PostgreSQL's pgvector

True. You *can* do vector search in SQL using extensions (like pgvector).

When it works well:

- Small/medium datasets
- You want strong relational joins + vectors
- You want fewer moving parts

When dedicated vector DB wins:

- Very large scale
- Heavy ANN tuning
- High throughput retrieval
- Hybrid search + filtering performance

What a typical RAG record looks like

Each chunk you store usually needs:

- `id`: unique chunk ID
- `vector`: embedding array
- `text`: the actual chunk content
- `metadata`: fields for filtering + traceability

Example metadata fields:

- `source` (pdf, web, doc)
- `doc_id`
- `title`
- `page`
- `chunk_index`
- `section`
- `created_at`
- `tenant_id` (multi-user systems)

Metadata storage (why it matters)

Vector similarity alone is not enough.

Metadata helps you:

- Filter to correct scope (e.g., a single course / client)
- Maintain citations / provenance (page numbers, URLs)
- Debug retrieval (“why did this chunk appear?”)
- Support access control (RBAC, per-user documents)

In RAG, metadata is not optional.

Filtering + retrieval (the common pattern)

You rarely want:

- “Search across everything”

You usually want:

- “Search within the right subset”

Examples:

- Only this user's docs: `tenant_id = X`
- Only policies: `doc_type = policy`
- Only recent docs: `created_at > date`
- Only relevant tags: `topic IN (...)`

So retrieval becomes:

vector search + metadata filter

Example: retrieval request (conceptual)

Inputs:

- query text
- filter conditions
- top_k

Process:

1. Embed the query → `q`
2. ANN search in vector index
3. Apply metadata filters (pre or post, depends on DB)
4. Return top_k chunks + metadata

Pre-filter vs Post-filter

Two ways vector DBs handle filters:

Pre-filter (preferred)

- Apply metadata constraints during ANN search
- Faster and more accurate for constrained scopes

Post-filter

- Search broadly first, then filter results
- Can miss good matches if filtered set is small

For RAG with strict scopes (tenant/user), pre-filter matters.

The RAG pipeline (where vector DB fits)

1. **Ingest**
    - load documents
    - clean text
    - chunk
2. **Embed**
    - generate embeddings for each chunk
3. **Store**
    - vector + text + metadata in vector DB
4. **Retrieve**
    - embed query
    - search + filter
5. **Generate**
    - send retrieved chunks to LLM with prompt + citations

Chunking (required for good RAG)

You do NOT embed whole PDFs.

You embed chunks:

- 200–800 tokens per chunk (common range)
- with overlap (e.g., 10–20%) to avoid cutting context

Chunking must preserve:

- section headings
- page info
- document identity
- ordering info

Bad chunking = bad retrieval.

What “good retrieval” means

A good retriever gets:

- **relevant** chunks (high precision)
- **complete** coverage (high recall)
- **from the right scope** (filters)
- **with traceable sources** (metadata)

RAG quality is often bottlenecked by retrieval, not the LLM.

Similarity metrics (quick)

Common options:

- **Cosine similarity**: compares direction (common for normalized embeddings)
- **Dot product**: similar to cosine if vectors are normalized
- **L2 distance**: Euclidean distance

Most modern embedding workflows:

- normalize vectors
- use cosine or dot

Hybrid search (very important for RAG)

Vector search is semantic, but sometimes you need exact terms:

- names, IDs, error codes
- legal section numbers
- version strings

Hybrid search = combine:

- **keyword search** (BM25)
- **vector search**

Benefits:

- better for “needle” queries
- better for rare proper nouns
- better grounding for technical/legal RAG

Re-ranking (often necessary)

ANN gives you candidates, not perfect ordering.

Common pattern:

1. Retrieve `top_k = 50`
2. Re-rank with a stronger model (cross-encoder / LLM rerank)
3. Keep top `k_final = 5–10`

This usually boosts answer quality a lot.

Storing text: in vector DB or elsewhere?

Two approaches:

Store text in vector DB

- simpler system
- easy retrieval

Store text in object store / SQL

- vector DB stores only IDs + metadata
- fetch text by ID after retrieval

Pick based on:

- scale
- cost
- system complexity

Multi-tenant RAG (must-have in products)

If many users upload documents:

Always store:

- `tenant_id`
- `user_id` (optional)
- access rules

And always filter retrieval by tenant/user scope.

If you don’t, you leak data across users.

Versioning + updates

Real systems have evolving docs.

Plan for:

- `doc_version`
- “soft delete” old chunks
- re-embedding on model change
- rebuild indexes when needed

Also store:

- embedding model name used (critical for migrations)

What to log for debugging RAG

At query time, log:

- query text
- filters applied
- retrieved chunk IDs + scores
- doc/page/section metadata
- final prompt context

This is how you diagnose:

- wrong chunks
- missing chunks
- over-filtering
- noisy embeddings

Common failure modes

- No metadata → cannot filter properly
- Wrong chunk size → either too vague or too fragmented
- No hybrid → fails on exact identifiers
- No rerank → weak ordering
- No scope filters → cross-user leakage
- Poor ingestion → garbage in, garbage out

Minimal “RAG-ready” vector DB checklist

You should be able to do:

- Upsert vectors with metadata
- ANN search with top_k
- Metadata filtering (ideally pre-filter)
- Delete / soft-delete by doc_id
- Multi-tenant isolation
- Optional: hybrid search + rerank support

Summary

- Vector DB = storage + fast similarity search for embeddings
- SQL is not optimized for high-dimensional ANN similarity search
- Metadata is required for filtering, traceability, and access control
- RAG quality improves with:
  - chunking + overlap
  - hybrid search
  - reranking
  - dedup/diversity
  - good logging