

# Vector Databases

Storing, indexing, and querying embeddings at scale for RAG, semantic search, and similarity lookups.

Last reviewed 28 Sep 2026 ·

**Prerequisites:** [Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) ·

**Related:** [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) · [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

## Overview

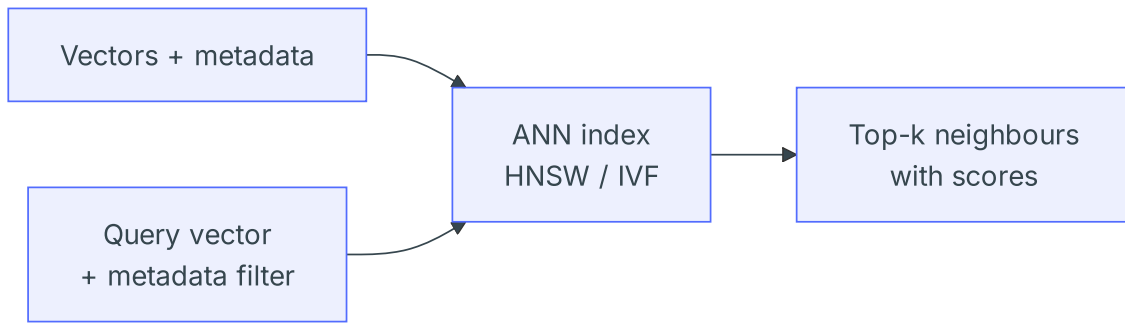
**Challenge:** Once documents are converted to embeddings, applications must quickly find the vectors closest to a query. Comparing the query with every stored vector is feasible for thousands of vectors but too slow for millions, and results usually also need filtering by source, date, or tenant.

**Solution:** a vector database stores vectors with their metadata and builds specialized indexes (such as HNSW graphs) that find *approximate* nearest neighbors in milliseconds without scanning every vector. It also provides database fundamentals: upserts and deletes, metadata filtering, access control, replication, and scaling.

```
query: "why did the orders DAG fail?" → embed → [0.12, -0.44, ...]
                                     |
                                     ANN index (HNSW graph): jump to the right
                                     neighborhood, walk a few hops
                                     |
                                     top-5 nearest chunks + filter: source =
                                     'runbooks', env = 'prod'
```

**When a dedicated database is needed:** often not initially. A relational database with a vector extension (such as PostgreSQL with pgvector) covers many use cases, and most warehouses and lakehouses now include vector search. Dedicated vector databases are

justified at large scale, with heavy filtering, many tenants, or strict latency targets.



## On this page

**Basic** - [Why a Vector Database](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#why-a-vector-database) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#why-a-vector-database) - [Core Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#core-concepts) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#core-concepts) - [pgvector \(PostgreSQL\)](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#pgvector-postgresql) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#pgvector-postgresql)

**Intermediate** - [Pinecone](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#pinecone) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#pinecone) - [Chroma](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#chroma) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#chroma) - [Weaviate](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#weaviate) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#weaviate)

**Advanced** - [Index Types & Trade-offs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#index-types--trade-offs) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#index-types--trade-offs) - [Filtering at Scale](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#filtering-at-scale) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#filtering-at-scale) - [Choosing the Right Vector DB](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#choosing-the-right-vector-db) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#choosing-the-right-vector-db) - [Production Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#production-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#production-patterns)

**Reference** - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/#further-reading)

## Why a Vector Database

A vector database stores embeddings and lets you find the N most similar ones to a query vector in milliseconds — even across millions of vectors.

```
Plain NumPy (brute force):  O(n) per query  - fine for < 100k vectors
```

Vector DB (ANN index):  $O(\log n)$  per query – millions of vectors, sub-10ms

**ANN = Approximate Nearest Neighbor** — trades tiny accuracy loss for massive speed gain.

Vector DBs also provide: - Metadata storage alongside each vector (doc title, source, date, tags) - Metadata filtering (search only vectors where `type="docs"`) - Namespaces/collections to separate indexes - Upsert semantics (re-index updated documents without rebuilding) - Managed scaling (no infrastructure to run)

## Core Concepts

| Concept                       | Definition                                                                   |
|-------------------------------|------------------------------------------------------------------------------|
| <b>Vector / Embedding</b>     | A list of floats representing a document's meaning                           |
| <b>Index</b>                  | The data structure that enables fast <u>ANN</u> search (HNSW, IVFFlat, etc.) |
| <b>Namespace / Collection</b> | A logical partition of the index (e.g., separate namespaces per customer)    |
| <b>Upsert</b>                 | Insert if new, update if ID already exists                                   |
| <b>k-NN query</b>             | "Find the k most similar vectors to this query vector"                       |
| <b>Metadata</b>               | Key-value pairs stored alongside each vector, used for filtering             |
| <b>Similarity metric</b>      | How to measure distance: cosine, dot product, or Euclidean                   |

## pgvector (PostgreSQL)

Best choice when you already have PostgreSQL and your dataset is < a few million vectors.

```
# Install extension
pip install pgvector psycopg2-binary
```

```
-- Enable the extension
CREATE EXTENSION IF NOT EXISTS vector;

-- Create a table with an embedding column
CREATE TABLE documents (
```

```

    id          SERIAL PRIMARY KEY,
    source      TEXT NOT NULL UNIQUE,    -- one row per
source: the upsert below relies on this constraint
    doc_type    TEXT,
    content     TEXT NOT NULL,
    embedding   vector(1536),           -- 1536 dims for text-
embedding-3-small
    created_at  TIMESTAMPTZ DEFAULT NOW()
);

-- IVFFlat index: smaller and faster to build than HNSW, with
a weaker speed-recall trade-off.
-- Create it AFTER the table has data (it learns its lists
from existing rows).
-- pgvector suggests lists = rows / 1000 up to 1M rows, and
sqrt(rows) beyond that;
-- queries look in ivfflat.probes lists (1 by default): raise
it for better recall
CREATE INDEX ON documents USING ivfflat (embedding
vector_cosine_ops)
WITH (lists = 100);

-- HNSW index: better recall, more memory, faster queries
(recommended for most cases)
CREATE INDEX ON documents USING hnsw (embedding
vector_cosine_ops)
WITH (m = 16, ef_construction = 64);

```

```

import psycopg2
from psycopg2.extras import execute_values
from pgvector.psycopg2 import register_vector
import numpy as np
from openai import OpenAI

openai_client = OpenAI()
conn =
psycopg2.connect("postgresql://user:pass@localhost:5432/mydb"
)
register_vector(conn)

def embed_batch(texts: list[str]) -> list[list[float]]:
    response = openai_client.embeddings.create(input=texts,

```

```

model="text-embedding-3-small")
    return [r.embedding for r in response.data]

# — Insert


---




---


def upsert_documents(documents: list[dict]):
    """documents = [{source, doc_type, content}, ...]"""
    texts = [d["content"] for d in documents]
    vecs = embed_batch(texts)

    with conn.cursor() as cur:
        rows = [(d["source"], d.get("doc_type"),
d["content"], v)
                for d, v in zip(documents, vecs)]
        execute_values(
            cur,
            "INSERT INTO documents (source, doc_type,
content, embedding) VALUES %s "
            "ON CONFLICT (source) DO UPDATE SET
content=EXCLUDED.content, embedding=EXCLUDED.embedding",
            rows
        )
        conn.commit()

# — Search


---




---


def search(query: str, k: int = 5, doc_type: str = None) ->
list[dict]:
    q_vec = embed_batch([query])[0]

    filter_clause = "WHERE doc_type = %s" if doc_type else ""

    sql = f"""
        SELECT source, doc_type, content,
               1 - (embedding <=> %s::vector) AS score
        FROM documents
        {filter_clause}
        ORDER BY embedding <=> %s::vector
        LIMIT %s
    """
    # The query vector appears twice (the score and the ORDER

```

```

BY), and the filter value sits between them
    params = (q_vec, q_vec, k) if not doc_type else (q_vec,
doc_type, q_vec, k)

    with conn.cursor() as cur:
        cur.execute(sql, params)
        return [{"source": r[0], "doc_type": r[1], "content":
r[2], "score": r[3]}
                for r in cur.fetchall()]

results = search("orders table schema", k=3)
for r in results:
    print(f"{r['score']:.3f} {r['source']}")

```

## Pinecone

Managed vector database — no infrastructure, scales to billions of vectors.

```

pip install pinecone          # the old package name
pinecone-client is deprecated

```

```

import os
from pinecone import Pinecone, ServerlessSpec
from openai import OpenAI

pc      = Pinecone(api_key=os.environ["PINECONE_API_KEY"])
client = OpenAI()

# — Create an index
—
pc.create_index(
    name="de-knowledge-base",
    dimension=1536,          # must match your embedding model
    metric="cosine",
    spec=ServerlessSpec(cloud="aws", region="us-east-1")
)

index = pc.Index("de-knowledge-base")

```

## # — Upsert vectors

---

```
documents = [
    {"id": "orders-schema", "text": "The orders table has
order_id, customer_id, amount, status, created_at", "type":
"schema"},
    {"id": "kafka-intro", "text": "Kafka is a distributed
event streaming platform", "type": "docs"},
]

def embed(texts):
    r = client.embeddings.create(input=texts, model="text-
embedding-3-small")
    return [x.embedding for x in r.data]

vecs = embed([d["text"] for d in documents])

vectors = [
    {
        "id": doc["id"],
        "values": vec,
        "metadata": {"text": doc["text"], "type":
doc["type"]}
    }
    for doc, vec in zip(documents, vecs)
]
index.upsert(vectors=vectors, namespace="default")
```

## # — Query

---

```
query_vec = embed(["what columns does the orders table
have?"])[0]

results = index.query(
    vector=query_vec,
    top_k=3,
    include_metadata=True,
    namespace="default",
    filter={"type": {"$eq": "schema"}} # metadata filter
)
```

```

for match in results.matches:
    print(f"{match.score:.3f}  {match.id}
{match.metadata['text'][:80]}")

# — Update / Delete

—

# Update a vector (upsert with same ID)
index.upsert(vectors=[{"id": "orders-schema", "values":
new_vec, "metadata": {...}}])

# Delete by ID
index.delete(ids=["orders-schema"], namespace="default")

# Delete by metadata filter
index.delete(filter={"type": {"$eq": "stale"}},
namespace="default")

# Stats
print(index.describe_index_stats())

```

## Chroma

Open-source, embeds natively in Python — great for local development and smaller production workloads.

```
pip install chromadb
```

```

import chromadb
from chromadb.utils import embedding_functions

# — Local (in-memory or persistent)

—

client = chromadb.Client() # in-
memory
client = chromadb.PersistentClient(path="./chroma_db") #
persists to disk

# — Use OpenAI embeddings

—

```

```

oai_ef = embedding_functions.OpenAIEmbeddingFunction(
    api_key="YOUR_OPENAI_API_KEY",
    model_name="text-embedding-3-small"
)

collection = client.create_collection(
    name="de_knowledge_base",
    embedding_function=oai_ef,
    metadata={"hnsw:space": "cosine"}
)

# — Add documents (Chroma embeds automatically)


---


collection.add(
    ids=["doc1", "doc2", "doc3"],
    documents=[
        "The orders table has order_id, customer_id, amount, status",
        "Airflow is a workflow orchestration tool using DAGs",
        "Kafka is a distributed event streaming platform",
    ],
    metadatas=[
        {"type": "schema", "team": "data"},
        {"type": "docs", "team": "data"},
        {"type": "docs", "team": "platform"},
    ]
)

# — Query


---


results = collection.query(
    query_texts=["orders table columns"],
    n_results=3,
    where={"type": {"$eq": "schema"}}, # metadata filter
)

for doc, meta, dist in zip(results["documents"][0],
                           results["metadatas"][0],
                           results["distances"][0]):
    print(f"dist={dist:.3f} type={meta['type']} {doc[:80]}")

```

```
# — Update / Delete
```

---

```
collection.update(ids=["doc1"], documents=["Updated orders  
schema text"])  
collection.delete(ids=["doc2"])  
collection.upsert(ids=["doc4"], documents=["New document"])  
  
print(collection.count()) # number of documents
```

## Weaviate

Open-source, production-grade, supports multi-modal embeddings and GraphQL.

```
pip install weaviate-client  
# Run: docker run -p 8080:8080 -p 50051:50051  
cr.weaviate.io/semitechnologies/weaviate:<version>
```

```
import weaviate  
from weaviate.classes.init import Auth  
  
# Local  
client = weaviate.connect_to_local()  
  
# Cloud  
client = weaviate.connect_to_weaviate_cloud(  
    cluster_url="YOUR_WCS_URL",  
    auth_credentials=Auth.api_key("YOUR_WCS_API_KEY"),  
)  
  
# — Create collection (schema)  
  
from weaviate.classes.config import Configure, Property,  
DataType  
  
client.collections.create(  
    name="Document",  
  
    vectorizer_config=Configure.Vectorizer.text2vec_openai(model=
```

```

"text-embedding-3-small"),
    properties=[
        Property(name="content", data_type=DataType.TEXT),
        Property(name="doc_type", data_type=DataType.TEXT),
        Property(name="source", data_type=DataType.TEXT),
    ]
)

```

```
collection = client.collections.get("Document")
```

```
# — Insert
```

---

```
from weaviate.classes.data import DataObject
```

```

collection.data.insert_many([
    DataObject(properties={"content": "The orders table has
order_id, amount, status",
                        "doc_type": "schema", "source":
"data_dict"}),
    DataObject(properties={"content": "Airflow orchestrates
pipelines as DAGs",
                        "doc_type": "docs", "source":
"wiki"}),
])

```

```
# — Semantic search
```

---

```
from weaviate.classes.query import MetadataQuery, Filter
```

```

results = collection.query.near_text(
    query="orders table columns",
    limit=3,
    filters=Filter.by_property("doc_type").equal("schema"),
    return_metadata=MetadataQuery(certainty=True)
)

```

```

for obj in results.objects:
    print(f"{obj.metadata.certainty:.3f}
{obj.properties['content'][:80]}")

```

```
# — Hybrid search (BM25 + vector)
```

```

results = collection.query.hybrid(
    query="orders table",
    alpha=0.5,    # 0=BM25, 1=vector
    limit=5
)

client.close()

```

## Index Types & Trade-offs

| Index          | Algorithm                    | Speed     | Recall | Memory | Notes                      |
|----------------|------------------------------|-----------|--------|--------|----------------------------|
| <b>Flat</b>    | Brute force                  | Slow      | 100%   | Low    | Only for < 100k vectors    |
| <b>IVFFlat</b> | Inverted file                | Fast      | 95–99% | Medium | Good for > 1M vectors      |
| <b>HNSW</b>    | Hierarchical graph           | Very fast | 95–99% | High   | Best for most use cases    |
| <b>ScaNN</b>   | Google's <a href="#">ANN</a> | Very fast | 99%+   | Medium | Production at Google scale |

```

-- pgvector: HNSW (recommended default)
CREATE INDEX ON documents USING hnsw (embedding
vector_cosine_ops)
WITH (m = 16, ef_construction = 64);
-- m: connections per node (16 is default; higher = better
recall, more memory)
-- ef_construction: build-time quality (64 is default; higher
= slower build, better index)

-- At query time:
SET hnsw.ef_search = 40; -- higher = better recall, slower
query (default 40)

```

## Filtering at Scale

The challenge: an ANN index is built for similarity, not for your `WHERE` clause. Combining the two naively either loses results (filtering *after* the ANN search) or gets slow (scanning every row that matches the filter).

### Solutions:

```
# 1. Filtered search inside the engine (preferred)
# Pinecone, Weaviate, Qdrant, Chroma, and Milvus apply
# metadata filters during the ANN search.
# pgvector 0.8+ uses iterative index scans to keep fetching
# until enough rows pass the filter:
#   SET hnsw.iterative_scan = relaxed_order;

# 2. Post-filter (search k*10, then filter down to k) –
# simple, but can return fewer than k
#   results when the filter is selective
def search_with_post_filter(query: str, k: int, doc_type:
str, overfetch: int = 10):
    candidates = search(query, k=k * overfetch)
    filtered    = [c for c in candidates if c["doc_type"] ==
doc_type]
    return filtered[:k]

# 3. Namespaces – separate index per filter value
# e.g., one Pinecone namespace per team or customer
index.query(vector=q_vec, top_k=5, namespace="team_data")
index.query(vector=q_vec, top_k=5, namespace="team_platform")
```

## Choosing the Right Vector DB

|                                         | pgvector      | Chroma                | Pinecone   | Weaviate     |
|-----------------------------------------|---------------|-----------------------|------------|--------------|
| <b>Managed</b>                          | No            | No                    | Yes        | Yes/No       |
| <b>Scale</b> (rough guide, not a limit) | Millions      | Up to about a million | Very large | Very large   |
| <b>Setup</b>                            | Need Postgres | Python-native         | API key    | Docker/Cloud |

|                      | pgvector                                              | Chroma                | Pinecone                     | Weaviate                   |
|----------------------|-------------------------------------------------------|-----------------------|------------------------------|----------------------------|
| <b>Cost</b>          | Postgres infra                                        | Free                  | Paid                         | Free/Paid                  |
| <b>Best for</b>      | Existing PG, a few million vectors                    | Local dev, small prod | Serverless, large scale      | Multi-modal, graph queries |
| <b>Hybrid search</b> | Manual (combine with <code>tsvector</code> full-text) | Limited               | Yes (sparse + dense vectors) | Yes (BM25 + vectors)       |
| <b>Multi-modal</b>   | No                                                    | No                    | No                           | Yes                        |

**Decision guide:** - Already have PostgreSQL → **pgvector** - Local development / prototyping  
→ **Chroma** - Production, don't want to manage infra → **Pinecone** - Need built-in hybrid search or multi-modal → **Weaviate** (or Pinecone's sparse + dense) - Other strong options: **Qdrant**, **Milvus**, **LanceDB**, and the vector search built into Databricks, Snowflake, Elasticsearch/OpenSearch, and MongoDB Atlas

## Production Patterns

### Incremental indexing

```
import hashlib

def get_content_hash(text: str) -> str:
    return hashlib.sha256(text.encode()).hexdigest()[:16]

def sync_documents(new_docs: list[dict], index, existing_ids: set):
    """Only upsert documents that are new or changed."""
    to_upsert = []
    for doc in new_docs:
        doc_id = f"{doc['source']}_{get_content_hash(doc['content'])}"
        if doc_id not in existing_ids:
            to_upsert.append({**doc, "id": doc_id})

    if to_upsert:
        texts = [d["content"] for d in to_upsert]
        vecs = embed_batch(texts)
        index.upsert(vectors=
```

```

        {"id": d["id"], "values": v, "metadata":
{"content": d["content"], "source": d["source"]}}
        for d, v in zip(to_upsert, vecs)
    ])
    print(f"Synced {len(to_upsert)} new/changed documents")

```

## Namespace-per-tenant (multi-tenancy)

```

def get_user_index(user_id: str):
    """Each user/team gets their own namespace – complete
    isolation."""
    return lambda **kwargs:
index.query(namespace=f"user_{user_id}", **kwargs)

def upsert_for_user(user_id: str, documents: list[dict]):
    index.upsert(vectors=vectors,
namespace=f"user_{user_id}")

```

## Cache frequent queries

```

import hashlib
from functools import lru_cache

@lru_cache(maxsize=1000)
def cached_search(query: str, k: int = 5) -> tuple:
    results = search(query, k=k)
    return tuple(frozenset(r.items()) for r in results)

```

## Monitor index health

```

def index_health_check(index) -> dict:
    stats = index.describe_index_stats()
    return {
        "total_vectors": stats.total_vector_count,
        "namespaces": list(stats.namespaces.keys()),
        "index_fullness_pct": stats.index_fullness * 100, #
meaningful for pod-based indexes only; serverless reports 0
    }

# Alert if index is >80% full (Pinecone pods have limits)

```

```
health = index_health_check(index)
if health["index_fullness_pct"] > 80:
    print("WARNING: index is >80% full – consider scaling")
```

## Common Pitfalls

| Pitfall                                                       | Symptom                                                   | Fix                                                                                                         |
|---------------------------------------------------------------|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| Adopting a dedicated vector DB before you need one            | Another system to secure, sync, and pay for               | Start with pgvector or your platform's built-in vector search; move when scale or features demand it        |
| Index metric doesn't match the embedding model                | Poor recall with no error                                 | Use the metric the model was trained for (usually cosine); be consistent at write and query time            |
| Never measuring recall of the <u>ANN</u> index                | Silently missing relevant results                         | Compare <u>ANN</u> results with exact search on a sample; tune <code>ef_search</code> / <code>nprobe</code> |
| Post-filtering with a selective filter                        | Fewer than k results, or none                             | Filtering inside the engine, partitions/namespaces, or pgvector iterative scans                             |
| Building IVFFlat on an empty or tiny table (pgvector)         | Terrible recall once data grows                           | Build IVFFlat after loading representative data, or use HNSW                                                |
| No stable IDs for chunks                                      | Duplicates on re-index; deletes can't find what to remove | Deterministic IDs ( <code>doc_id#chunk_n</code> ) and upserts                                               |
| Forgetting deletes                                            | Deleted or expired documents keep showing up in answers   | Propagate source deletes ( <u>CDC</u> ); periodic reconciliation against the source                         |
| One index shared by all tenants with only a prompt-level rule | Data leaks between customers                              | Namespace/collection per tenant, or a mandatory tenant filter enforced in code                              |
| Changing the embedding model in place                         | Old and new vectors mixed in one index                    | Build a new index, backfill, switch reads, then drop the old one                                            |

# Cheat Sheet

## pgvector

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE TABLE chunks (id text PRIMARY KEY, doc_id text,
content text, metadata jsonb,
                        embedding vector(1024));
CREATE INDEX ON chunks USING hnsw (embedding
vector_cosine_ops);

-- top 5 by cosine distance (<=> cosine, <-> L2, <#> negative
inner product)
SELECT id, content, 1 - (embedding <=> $1) AS similarity
FROM chunks
WHERE metadata->>'source' = 'runbooks'
ORDER BY embedding <=> $1
LIMIT 5;

SET hnsw.ef_search = 100; -- recall vs. speed
SET hnsw.iterative_scan = relaxed_order; -- pgvector 0.8+:
better results with filters
```

| Operation | Pinecone                                                            | Chroma                                                                          | Weaviate (v4 client)                                                 |
|-----------|---------------------------------------------------------------------|---------------------------------------------------------------------------------|----------------------------------------------------------------------|
| Connect   | <code>Pinecone(api_key=...<br/>).Index("idx")</code>                | <code>chromadb.PersistentClie<br/>nt(path)</code>                               | <code>weaviate.connect_to_<br/>local()</code>                        |
| Upsert    | <code>index.upsert(vectors<br/>=[(id, vec, meta)])</code>           | <code>col.upsert(ids,<br/>embeddings, metadatas,<br/>documents)</code>          | <code>col.data.insert_many<br/>([...])</code>                        |
| Query     | <code>index.query(vector=q<br/>, top_k=5, filter=<br/>{...})</code> | <code>col.query(query_embeddi<br/>ngs=[q], n_results=5,<br/>where={...})</code> | <code>col.query.near_vecto<br/>r(q, limit=5,<br/>filters=...)</code> |
| Delete    | <code>index.delete(ids=<br/>[...])</code>                           | <code>col.delete(ids=[...])</code>                                              | <code>col.data.delete_by_i<br/>d(uuid)</code>                        |

| Index type   | Good for                                              | Main knobs |
|--------------|-------------------------------------------------------|------------|
| Flat (exact) | Under ~100k vectors; ground truth<br>for recall tests | —          |

| Index type                               | Good for                                  | Main knobs                                                                |
|------------------------------------------|-------------------------------------------|---------------------------------------------------------------------------|
| HNSW                                     | Default choice: fast, high recall         | <code>m</code> , <code>ef_construction</code> ,<br><code>ef_search</code> |
| IVF / IVFFlat                            | Large collections, lower memory           | <code>lists</code> , <code>probes</code>                                  |
| Quantized (PQ / binary / half-precision) | Very large collections on a memory budget | Compression level vs. recall                                              |

**Sizing:** 1M vectors × 1,024 dims × 4 bytes ≈ 4 GB of raw vectors, before index overhead (HNSW adds roughly 1.5–2×)

## Interview Questions

**Q: What is approximate nearest neighbor (ANN) search and why is it used?** A: Exact nearest-neighbor search compares the query with every vector, which is  $O(n)$  and too slow at millions of vectors. ANN indexes trade a little recall for huge speed-ups: HNSW builds a layered proximity graph and walks it greedily toward the query; IVF clusters vectors and only searches the closest clusters. You tune the trade-off ( `ef_search` , `nprobe` ) and measure recall against exact search on a sample.

**Q: How does HNSW work at a high level?** A: HNSW (Hierarchical Navigable Small World) builds a multi-layer graph where each vector links to its nearest neighbors. Upper layers are sparse, with long-range links, and lower layers are dense. A search starts at the top, greedily moves toward the query, then drops a layer and repeats, finishing with a local search at the bottom layer. It gives excellent recall and latency, at the cost of memory and slower index builds.

**Q: pgvector or a dedicated vector database — how do you decide?** A: pgvector is a great default if you already run Postgres: vectors live next to relational data, you get transactions, joins, and existing backups and permissions, and it handles millions of vectors well with HNSW. A dedicated vector database makes sense for very large collections, many tenants, heavy filtered search, strict latency at high QPS, or built-in hybrid search and managed scaling. Decide with a benchmark on your data and filters, not a feature list.

**Q: How do you handle metadata filtering in vector search?** A: Store filterable attributes with each vector and apply filters inside the engine during the ANN search, so results aren't lost the way they are with post-filtering. For strict isolation (tenants, environments), use separate namespaces, collections, or partitions. In pgvector, combine `WHERE` clauses with the HNSW index and enable iterative scans for selective filters, or partition the table by the filter key.

**Q: How would you keep a vector index in sync with a changing source system?** A: Treat the index as a derived table: capture changes from the source (CDC or change timestamps), re-chunk and re-embed only changed documents, upsert by deterministic chunk IDs, and delete vectors for removed documents. Store the source version or hash and the embedding model version in metadata, monitor index freshness, and run periodic reconciliation to catch drift.

---

## Further Reading

- [pgvector README](https://github.com/pgvector/pgvector) (https://github.com/pgvector/pgvector) — indexes, operators, and tuning
  - [Pinecone documentation](https://docs.pinecone.io/) (https://docs.pinecone.io/)
  - [Weaviate documentation](https://docs.weaviate.io/) (https://docs.weaviate.io/)
  - [Chroma documentation](https://docs.trychroma.com/) (https://docs.trychroma.com/)
  - [ANN Benchmarks](https://ann-benchmarks.com/) (https://ann-benchmarks.com/) — recall/speed comparisons of ANN algorithms
  - *Efficient and robust approximate nearest neighbor search using HNSW graphs* — Malkov & Yashunin (the HNSW paper)
- 

**Previous:** [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) · **Next:** [AI Agents](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026