

pgvector

Version pgvector 0.8.7 · PostgreSQL 18 · verified 2026-11-02

Vector search in Postgres — types, distance operators, exact vs HNSW vs IVFFlat, the post-filter trap, hybrid search, upserts and sizing.

Contents

Mental model 2

The vector type and distance operators 2

Exact search and index use 4

HNSW 5

IVFFlat 7

Filtering: the post-filter trap 8

Hybrid search with full-text 10

Chunks, upserts and reindexing 12

Sizing memory 13

Further reading 14

Mental model

A vector index trades recall for speed: once you add one, a query can silently miss true nearest neighbours, and every knob — `m`, `ef_search`, `lists`, `probes` — moves you along that curve. So decide how much recall you can lose, measure it against exact search, and tune from that budget rather than shipping the defaults. Everything else is ordinary PostgreSQL: rows, transactions, `WHERE` clauses and B-tree indexes beside the vectors.

The vector type and distance operators

Operator	Distance	Index opclass
`<->`	L2 (Euclidean)	`vector_l2_ops`
`<=>`	Cosine	`vector_cosine_ops`
`<#>`	Negative inner product	`vector_ip_ops`
`<+>`	L1 (taxicab)	`vector_l1_ops`

Operator	Distance	Index opclass
`<~>`, `<%>`	Hamming, Jaccard (`bit`)	`bit_hamming_ops`, …

`vector(n)` stores `n` 4-byte floats; `halfvec(n)` stores 2-byte floats; `bit(n)` and `sparsevec(n)` cover binary and sparse embeddings. Pick the operator your embedding model was trained for, and build the index with the matching opclass — an index serves only its own operator.

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE TABLE chunks (
  id        bigint GENERATED ALWAYS AS IDENTITY
            PRIMARY KEY,
  doc_id    bigint NOT NULL,
  content   text NOT NULL,
  embedding vector(3) NOT NULL  -- your model's
dims
);
```

```
INSERT INTO chunks (doc_id, content, embedding)
VALUES (1, 'heat pump sizing', '[1,0,1]'),
       (1, 'battery storage', '[0,1,0]');
```

> **Gotcha:** `<#>` returns the *negative* inner product, so smaller is closer. For cosine similarity compute `1 - (a <=> b)`; for inner product multiply by `-1`.

Exact search and index use

Query shape	Uses an ANN index
`ORDER BY e <=> q LIMIT k`	Yes
`ORDER BY 1 - (e <=> q) DESC`	No
`WHERE e <=> q < 0.3` alone	No
No index at all	Exact: perfect recall

Without an index pgvector scans every row and returns the true nearest neighbours. That is your recall baseline, and on small tables it is also the fastest plan. The

planner uses a vector index only for `ORDER BY` a distance operator, ascending, with a `LIMIT`.

```
SELECT id, content,
       embedding <=> '[1,0,1]' AS distance
FROM chunks
ORDER BY embedding <=> '[1,0,1]'
LIMIT 5;
```

> **Tip:** measure recall by running the same query with `SET LOCAL enable_indexscan = off` inside a transaction (exact) and comparing the ids with the indexed result.

HNSW

Parameter	Default	Raise it for
`m` (build)	16	Recall on hard data; costs memory

Parameter	Default	Raise it for
`ef_construction` (build)	64	Recall; costs build time
`hnsw.ef_search` (query)	40	Recall; costs latency
Max indexed dims	2,000 `vector`, 4,000 `halfvec`	—

HNSW builds a layered graph. It has the better speed-recall trade-off, needs no training data so you can create it on an empty table, but builds slower and uses more memory than IVFFlat. Tune `ef_search` per query with `SET LOCAL`; it also caps how many rows one scan returns.

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

BEGIN;
```

```
SET LOCAL hnsw.ef_search = 100;
SELECT id FROM chunks
ORDER BY embedding <=> '[1,0,1]' LIMIT 10;
COMMIT;
```

IVFFlat

Parameter	Start at
`lists` (≤ 1M rows)	`rows / 1000`
`lists` (> 1M rows)	`sqrt(rows)`
`ivfflat.probes`	`sqrt(lists)` ; default is 1

IVFFlat clusters vectors into `lists` with k-means at build time and searches the `probes` closest lists. It builds faster and smaller than HNSW but has the worse speed-recall trade-off. Setting `probes` equal to `lists` makes the search exact, and the planner stops using the index.

```
CREATE INDEX chunks_ivf ON chunks
  USING ivfflat (embedding vector_cosine_ops)
```

```
  WITH (lists = 100);
SET ivfflat.probes = 10;
```

> **Gotcha:** the clusters are learned from the rows present at build time. An IVFFlat index created on an empty or tiny table returns fewer results than `LIMIT`; drop it and rebuild once the data exists.

Filtering: the post-filter trap

Approach	When
B-tree on the filter column	Filter matches few rows (exact)
`hnsw.iterative_scan` (0.8.0+)	Filter matches many rows
Partial ANN index `WHERE …`	A few distinct filter values
`PARTITION BY LIST`	Many values, tenant isolation

With an approximate index the `WHERE` clause is applied *after* the index scan. The README's numbers: a filter matching 10% of rows with `ef_search` at 40 returns about 4 rows on average, not 10. Iterative scans keep reading the index until `LIMIT` is met or a cap is hit (`hnsw.max_scan_tuples`, 20,000 by default).

```
SET hnsw.iterative_scan = relaxed_order;
SELECT id FROM chunks
WHERE doc_id = 1
ORDER BY embedding <=> '[1,0,1]' LIMIT 10;

CREATE INDEX ON chunks
  USING hnsw (embedding vector_cosine_ops)
  WHERE (doc_id = 1);
```

> **Gotcha:** a shared index across tenants lets one tenant's vectors crowd out another's — "vectors from one tenant can affect recall". Partition by tenant, or use a table per tenant.

Hybrid search with full-text

Piece	Role
`tsvector` column + GIN index	Keyword match
`ts_rank_cd(tsv, query)`	Keyword rank
Vector `ORDER BY … LIMIT`	Semantic rank
`1.0 / (k + rank)`, `k = 60`	Reciprocal Rank Fusion

Embeddings miss exact terms such as part numbers and error codes; full-text search misses paraphrase. Run both, take the top 20 of each, and fuse by rank rather than by raw score, since the two scores are on unrelated scales.

```
ALTER TABLE chunks ADD COLUMN tsv tsvector
  GENERATED ALWAYS AS
  (to_tsvector('english', content)) STORED;
WITH s AS (SELECT id, rank() OVER (ORDER BY
    embedding <=> '[1,0,1]') AS r FROM chunks
  ORDER BY embedding <=> '[1,0,1]' LIMIT 20),
k AS (SELECT id, rank() OVER (ORDER BY
    ts_rank_cd(tsv, q) DESC) AS r
  FROM chunks, plainto_tsquery('english', 'pump')
q
  WHERE tsv @@ q LIMIT 20)
SELECT coalesce(s.id, k.id) AS id,
  coalesce(1.0 / (60 + s.r), 0)
  + coalesce(1.0 / (60 + k.r), 0) AS score
FROM s FULL JOIN k ON s.id = k.id
ORDER BY score DESC LIMIT 5;
```

Chunks, upserts and reindexing

Task	Do
One row per chunk	Keep `doc_id` and chunk position
Re-embed a changed document	Delete its chunks, insert new ones
Bulk load	`COPY … (FORMAT BINARY)`, index afterwards
Index in production	`CREATE INDEX CONCURRENTLY`
New embedding model	New column or table, then swap

Each chunk is a row with its own vector, so a document edit is a transaction that replaces that document's chunks. Vectors from different models are not comparable, even at the same dimension; never mix them in one indexed column.

```
BEGIN;
DELETE FROM chunks WHERE doc_id = 1;
INSERT INTO chunks (doc_id, content, embedding)
VALUES (1, 'heat pump sizing v2', '[1,0,0.9]'),
       (1, 'battery storage v2', '[0,1,0.1]');
COMMIT;
REINDEX INDEX CONCURRENTLY chunks_hnsw;
```

> **Warning:** HNSW vacuum is slow; the README's advice is `REINDEX INDEX CONCURRENTLY` first, then `VACUUM`. Heavy delete and re-insert churn leaves dead tuples that also shrink result counts.

Sizing memory

Type	Bytes per value
`vector(d)`	`4 * d + 8`
`halfvec(d)`	`2 * d + 8`
`bit(d)`	`d / 8 + 8`

An index need not fit in memory, but approximate search is fast only when it does. HNSW builds "significantly faster when the graph fits into `maintenance_work_mem`", and Postgres prints a notice when it no longer does. To shrink the working set, store `halfvec`, or index `binary_quantize(embedding)` and re-rank the top candidates with the full vectors.

```
SET maintenance_work_mem = '2GB';
SET max_parallel_maintenance_workers = 7;
CREATE INDEX CONCURRENTLY chunks_hnsw2 ON chunks
  USING hnsw (embedding vector_cosine_ops);
SELECT pg_size_pretty(
  pg_relation_size('chunks_hnsw2'));
```

Further reading

- pgvector README
- pgvector — Iterative index scans
- pgvector-python — hybrid search examples
- PostgreSQL 18 — Text Search Controls