EcomLabs

STEP-BY-STEP GUIDE

Semantic product search with pgvector

The table and its HNSW index, the script that creates and refreshes the embeddings, the similarity query and a ready-to-copy hybrid query that fuses full-text search with RRF. All of it tested, with real output and a checklist.

FOR	Stores on Postgres that want search to understand sentences and synonyms
TESTED WITH	Node 22.22.3 · PGlite 0.5.10 (PostgreSQL 18.6) · pgvector 0.8.7 · Postgres.js 3.4.9 · Windows 11
UPDATED	November 11, 2026
DOCS	github.com/pgvector/pgvector
ARTICLE	ecomlabs.dev/g/embeddings-en
AUTHOR	Luis Morales · @luismoralesdev

Contents

Before you start	What you will build
Step 01	The table and its indexes
Step 02	The script that creates the embeddings
Step 03	The ready queries
Step 04	What we tested
Step 05	Cost, and when to re-embed
When you finish	Checklist

BEFORE YOU START

What you will build

A `products` table with an HNSW index, a script that embeds only what changed, and two queries: similarity, and hybrid (full-text plus meaning, fused with Reciprocal Rank Fusion). We tested it on November 11, 2026 with Node 22.22.3, Postgres.js 3.4.9 (`npm install postgres`) and PGlite 0.5.10 (PostgreSQL 18.6 in WebAssembly) with its pgvector 0.8.7 extension.

> **CAUTION**
>
> The tests called no embeddings provider: the `embed` function was a stub returning toy 8-dimensional vectors, built by hand from about forty words. The SQL and the script are tested; search quality with a real model is not. Measure it with real searches from your store.

STEP 01

The table and its indexes

Check that your Postgres has pgvector with `SELECT * FROM pg_available_extensions WHERE name = 'vector'`. Then save this as `schema.sql` and run it as a migration, with `1024` replaced by your model's dimension (HNSW indexes `vector` up to 2,000; beyond that, `halfvec` up to 4,000):

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE products (
  id            bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  handle        text NOT NULL UNIQUE,
  title         text NOT NULL,
  description   text NOT NULL DEFAULT '',
  search        tsvector GENERATED ALWAYS AS
                  (to_tsvector('english', title || ' ' || description)) STORED,
  content_hash  text GENERATED ALWAYS AS (md5(title || ' ' || description)) STORED,
  embedding     vector(1024),
  embedded_hash text,
  embedded_with text
);

CREATE INDEX products_search_idx ON products USING gin (search);
CREATE INDEX products_embedding_idx ON products USING hnsw (embedding vector_cosine_ops);
```

STEP 02

The script that creates the embeddings

Save `index-products.mjs`:

```
// Embeds every product whose text or model changed since its last embedding.
// embed(texts) calls your provider and returns one vector per text, in order.
export async function indexProducts(sql, embed, { model, dimensions, batchSize = 64 }) {
  let total = 0;
  for (;;) {
    const rows = await sql`
      SELECT id, title || ' ' || description AS text, content_hash
      FROM products
      WHERE embedded_hash IS DISTINCT FROM content_hash
         OR embedded_with IS DISTINCT FROM ${model}
      ORDER BY id
      LIMIT ${batchSize}`;
    if (rows.length === 0) return total;

    const vectors = await embed(rows.map((r) => r.text));
    if (vectors.length !== rows.length) {
      throw new Error(`expected ${rows.length} vectors, got ${vectors.length}`);
    }
    for (const v of vectors) {
      if (v.length !== dimensions || !v.every(Number.isFinite)) {
        throw new Error(`expected ${dimensions} finite numbers per vector`);
      }
    }

    await sql.begin(async (tx) => {
      for (const [i, row] of rows.entries()) {
        await tx`
          UPDATE products
          SET embedding = ${JSON.stringify(vectors[i])}::vector,
              embedded_hash = ${row.content_hash}, embedded_with = ${model}
          WHERE id = ${row.id}`;
      }
    });
    total += rows.length;
  }
}
```

Call it from a worker on your job queue, never inside the request that saves the product: `await indexProducts(sql, embed, { model: "your-model-name", dimensions: 1024 })`.

- `embed(texts)` takes an array of texts and returns an array of vectors in the same order. Wire it to your provider's SDK or API following its docs; if it separates documents from queries, this is where the "document" type goes.
- **The validation** rejects the wrong number of vectors, the wrong dimension or non-finite numbers before writing. **The stored hash** is that of the text sent: if the product changes while the provider answers, the next loop redoes it.
- **Changing** `model` re-embeds the whole catalog: vectors from two models cannot be compared.

STEP 03

The ready queries

Save `search.mjs`. `embed` is the same function as in step 02; if your provider asks, this is where the "query" type goes:

```
export async function semanticSearch(sql, embed, q, limit = 5) {
  const [vector] = await embed([q]);
  const v = JSON.stringify(vector);
  return sql`
    SELECT handle, title, round((1 - (embedding <=> ${v}::vector))::numeric, 3) AS similarity
    FROM products
    WHERE embedding IS NOT NULL
    ORDER BY embedding <=> ${v}::vector
    LIMIT ${limit}`;
}

export async function hybridSearch(sql, embed, q, limit = 5) {
  const [vector] = await embed([q]);
  const v = JSON.stringify(vector);
  return sql`
    WITH semantic AS (
      SELECT id, rank() OVER (ORDER BY embedding <=> ${v}::vector) AS rank
      FROM products
      WHERE embedding IS NOT NULL
      ORDER BY embedding <=> ${v}::vector
      LIMIT 20
    ),
    keyword AS (
      SELECT id, rank() OVER (ORDER BY ts_rank_cd(search, query) DESC) AS rank
      FROM products, websearch_to_tsquery('english', ${q}) query
      WHERE search @@ query
      ORDER BY ts_rank_cd(search, query) DESC
      LIMIT 20
    )
    SELECT p.handle, semantic.rank AS by_meaning, keyword.rank AS by_text,
           round(COALESCE(1.0 / (60 + semantic.rank), 0)
               + COALESCE(1.0 / (60 + keyword.rank), 0), 4) AS score
    FROM semantic
    FULL OUTER JOIN keyword ON keyword.id = semantic.id
    JOIN products p ON p.id = COALESCE(semantic.id, keyword.id)
    ORDER BY score DESC, p.id
    LIMIT ${limit}`;
}
```

- `<=>` is the cosine distance that `vector_cosine_ops` speeds up. **The hybrid query** adds `1 / (60 + position)` for every list a product appears in: whatever ranks well by keywords and by meaning wins.
- **If filters leave you short of results**, raise `hnsw.ef_search` (40 by default) with `SET LOCAL` or turn on `SET hnsw.iterative_scan = strict_order` (pgvector 0.8.0 or later).

STEP 04

What we tested

With `node --test`, all 9 tests passed:

- The first indexing run processed 10 products and the second, 0. One edited product was re-embedded alone, a new model re-embedded all 10 and one edited mid-call was redone.
- Answers with 2-number vectors or one vector missing threw without writing anything; with `enable_seqscan` off, `EXPLAIN` used the HNSW and GIN indexes.

"something warm for the snow": keywords, 0 rows; meaning, jacket (0.920), beanie (0.901) and socks (0.780). "boots": semantic search ranked the running shoes first; hybrid search, the boots:

```
-- hybridSearch('boots')
 handle        | by_meaning | by_text | score
---------------+------------+---------+--------
 rain-boots    | 3          | 1       | 0.0323
 running-shoes | 1          |         | 0.0164
 thermal-socks | 2          |         | 0.0161
 down-jacket   | 4          |         | 0.0156
 wool-beanie   | 4          |         | 0.0156
(5 rows)
```

The last two rows have a similarity of 0: vector search never answers "no results." Cut at a minimum similarity you calibrate against your model.

STEP 05

Cost, and when to re-embed

Cost is tokens × your provider's price per token: indexing the catalog once, plus one short query embedding per search. Each 1024-dimension vector takes 4,104 bytes (4 × dimensions + 8, measured on PGlite). Re-embed when the text you embed or the model changes; not when price or stock changes, which you filter with `WHERE`.

WHEN YOU FINISH

Checklist

- [ ] pgvector is enabled and the `vector(n)` column has your model's exact dimension.
- [ ] `indexProducts` runs in the background, from a queue, and validates the provider's answer.
- [ ] Documents and queries use the same model, with the right input type if your provider asks for one.
- [ ] Search blends keywords and meaning, cuts off distant results and falls back to keywords if the provider fails.
- [ ] You tested with real searches that return nothing today.