EcomLabs

GUÍA PASO A PASO

Búsqueda semántica de productos con pgvector

La tabla con su índice HNSW, el script que genera y actualiza los embeddings, la consulta de similitud y la consulta híbrida con texto completo y RRF, lista para copiar. Todo probado, con la salida real y la checklist.

PARA	Tiendas con Postgres que quieren un buscador que entienda frases y sinónimos
PROBADO CON	Node 22.22.3 · PGlite 0.5.10 (PostgreSQL 18.6) · pgvector 0.8.7 · Postgres.js 3.4.9 · Windows 11
ACTUALIZADA	11 de noviembre de 2026
DOCUMENTACIÓN	github.com/pgvector/pgvector
ARTÍCULO	ecomlabs.dev/g/embeddings
AUTOR	Luis Morales · @luismoralesdev

Índice

Antes de empezar	Qué vas a montar
Paso 01	La tabla y los índices
Paso 02	El script que genera los embeddings
Paso 03	Las consultas listas
Paso 04	Lo que probamos
Paso 05	Costo y cuándo regenerar
Al terminar	Checklist

ANTES DE EMPEZAR

Qué vas a montar

Una tabla `products` con su índice HNSW, un script que genera los embeddings solo de lo que cambió y dos consultas: la de similitud y la híbrida (texto completo + significado, con Reciprocal Rank Fusion). Lo probamos el 11 de noviembre de 2026 con Node 22.22.3, Postgres.js 3.4.9 (`npm install postgres`) y PGlite 0.5.10 (PostgreSQL 18.6 en WebAssembly) con su extensión pgvector 0.8.7.

> **CUIDADO**
>
> En las pruebas no llamamos a ningún proveedor de embeddings: la función `embed` fue un stub con vectores de juguete de 8 dimensiones, hechos a mano a partir de unas cuarenta palabras. El SQL y el script están probados; la calidad de la búsqueda con un modelo real, no. Mídela con búsquedas reales de tu tienda.

PASO 01

La tabla y los índices

Comprueba que tu Postgres tenga pgvector con `SELECT * FROM pg_available_extensions WHERE name = 'vector'`. Luego guarda esto como `schema.sql` y córrelo como migración, con `1024` cambiado por la dimensión de tu modelo (HNSW indexa `vector` hasta 2,000; más allá, `halfvec` hasta 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('spanish', 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);
```

PASO 02

El script que genera los embeddings

Guarda `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;
  }
}
```

Llámalo desde un worker de tu cola de trabajos, nunca dentro del request que guarda el producto: `await indexProducts(sql, embed, { model: "nombre-del-modelo", dimensions: 1024 })`.

- `embed(texts)` recibe un arreglo de textos y devuelve un arreglo de vectores en el mismo orden. Conéctala al SDK o la API de tu proveedor según su documentación; si distingue documento y consulta, aquí va el tipo «documento».
- **La validación** rechaza otra cantidad de vectores, otra dimensión o números no finitos antes de escribir. **El hash guardado** es el del texto enviado: si el producto cambia mientras el proveedor responde, la siguiente vuelta lo rehace.
- **Cambiar** `model` regenera todo el catálogo: los vectores de dos modelos no se comparan.

PASO 03

Las consultas listas

Guarda `search.mjs`. `embed` es la misma función del paso 02; si tu proveedor lo pide, aquí va el tipo «consulta»:

```
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('spanish', ${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}`;
}
```

- `<=>` es la distancia coseno, la que acelera `vector_cosine_ops`. **La híbrida** suma `1 / (60 + posición)` por cada lista donde aparece el producto: gana el que sale bien por texto y por significado.
- **Si faltan resultados** con filtros, sube `hnsw.ef_search` (40 por defecto) con `SET LOCAL` o activa `SET hnsw.iterative_scan = strict_order` (pgvector 0.8.0 o superior).

PASO 04

Lo que probamos

Con `node --test`, las 9 pruebas pasaron:

- La primera indexación procesó 10 productos y la segunda, 0. Un producto editado se regeneró solo, un modelo nuevo regeneró los 10 y uno editado durante la llamada se rehízo.
- Respuestas con vectores de 2 números o con uno de menos lanzaron error sin escribir nada; con `enable_seqscan` apagado, `EXPLAIN` usó los índices HNSW y GIN.

«algo calientito para la nieve»: texto, 0 filas; semántica, chamarra (0.920), gorro (0.901) y calcetines (0.780). «botas»: la semántica puso primero los tenis; la híbrida, las botas:

```
-- hybridSearch('botas')
 handle              | by_meaning | by_text | score
---------------------+------------+---------+--------
 botas-lluvia        | 3          | 1       | 0.0323
 tenis-correr        | 1          |         | 0.0164
 calcetines-termicos | 2          |         | 0.0161
 chamarra-plumas     | 4          |         | 0.0156
 gorro-lana          | 4          |         | 0.0156
(5 rows)
```

Las dos últimas filas tienen similitud 0: la búsqueda por vectores nunca responde «sin resultados». Corta por una similitud mínima que calibres con tu modelo.

PASO 05

Costo y cuándo regenerar

El costo es tokens × el precio por token de tu proveedor: indexar el catálogo una vez, más una búsqueda corta por cada consulta. Cada vector de 1024 dimensiones ocupa 4,104 bytes (4 × dimensiones + 8, medido en PGlite). Regenera cuando cambia el texto que conviertes en vector o el modelo; no cuando cambian el precio o el stock, que se filtran con `WHERE`.

AL TERMINAR

Checklist

- [ ] pgvector está activado y la columna `vector(n)` tiene la dimensión exacta de tu modelo.
- [ ] `indexProducts` corre en segundo plano, desde una cola, y valida la respuesta del proveedor.
- [ ] Documentos y consultas usan el mismo modelo, con el tipo de entrada correcto si tu proveedor lo pide.
- [ ] La búsqueda combina texto y significado, corta los resultados lejanos y, si el proveedor falla, responde solo con texto.
- [ ] Probaste con búsquedas reales que hoy no dan resultados.