PuigFlows

Guia premium | puigflows.com

pgvector en Produccion: HNSW, Busqueda con Filtros, Cuantizacion y Recall Medido de Verdad

De un prototipo de 5.000 filas a millones de vectores sin sorpresas: tipos de columna, HNSW por dentro, medicion real del recall, filtros e iterative scans, cuantizacion, memoria, busqueda hibrida, multi-tenant y operacion en el tiempo.

Tipo: Guia | Dificultad: Intermedio | Categoria: IA
Tiempo de lectura: ~44 min | Edicion ampliada: 21 de agosto de 2026
Version en espanol --- English version starts on page 24

Cuando la búsqueda vectorial deja de escalar

El prototipo siempre funciona. Cargas 5.000 fragmentos de texto en una tabla de PostgreSQL con una columna `vector(1536)`, lanzas un `ORDER BY embedding <=> $1 LIMIT 5` y la respuesta llega en doce milisegundos. Ni siquiera necesitas un índice: recorrer 5.000 filas es trivial para cualquier máquina.

Seis meses después hay cuatro millones de fragmentos, la misma consulta tarda seis segundos, alguien crea un índice HNSW con los parámetros por defecto y la latencia vuelve a 20 ms. Problema resuelto. Hasta que producto añade un filtro por idioma y el buscador empieza a devolver dos resultados donde antes devolvía diez. No hay excepción, no hay timeout, no salta ninguna alerta. El RAG simplemente responde peor, y tardas semanas en darte cuenta de por qué.

Esa distancia entre *funciona* y *funciona bien* es de lo que trata esta guía: elegir el tipo de columna antes de que el disco te lo recuerde, entender qué hacen de verdad los dos índices de pgvector, medir el recall en lugar de suponerlo, y sobre todo aprender por qué mezclar filtros con búsqueda aproximada es la primera causa de resultados incorrectos que nadie detecta.

Todo lo que sigue asume pgvector 0.8 o superior sobre PostgreSQL 16+. Varias de las piezas clave (los *iterative scans*, la estimación de coste mejorada) no existen en versiones anteriores, así que lo primero es comprobar qué tienes instalado.

```
SELECT extname, extversion FROM pg_extension WHERE extname = 'vector';
-- Si ves 0.7.x o menos, actualiza antes de seguir:
ALTER EXTENSION vector UPDATE;
```

1. El tipo de columna decide tu factura antes que el índice

pgvector no ofrece un solo tipo de dato, y la diferencia entre ellos no es cosmética. Un `vector` guarda cada dimensión como un float de 4 bytes, más 8 bytes de cabecera. Un `halfvec` usa 2 bytes por dimensión. Un `bit`

usa un bit. Con 1536 dimensiones, la aritmética queda así:

```
SELECT pg_column_size('[1,2,3]'::vector)   AS vector_3d,    -- 20 bytes  (4*3 + 8)
       pg_column_size('[1,2,3]'::halfvec)  AS halfvec_3d,   -- 14 bytes  (2*3 + 8)
       pg_column_size('101'::bit(3))       AS bit_3d;       -- 9 bytes

-- Para 1536 dimensiones y 4 millones de filas, sólo la columna:
--   vector(1536)   -> 6.152 B/fila  ->  ~23 GB
--   halfvec(1536)  -> 3.080 B/fila  ->  ~11 GB
--   bit(1536)      ->   200 B/fila  ->  ~0,7 GB
```

Hay un detalle que casi nadie tiene en cuenta y que se paga en cada consulta: PostgreSQL mueve fuera de la fila cualquier valor que haga crecer la tupla por encima de unos 2 KB. Un `vector(1536)` siempre acaba en TOAST, así que cada vez que lees el embedding real (por ejemplo, al reordenar candidatos) haces una lectura extra a la tabla TOAST. Un `halfvec(768)` ocupa 1.544 bytes y cabe dentro de la fila. No es un detalle académico: en reranking sobre cientos de candidatos, esa lectura adicional se nota.

La regla práctica: guarda el vector de máxima precisión que tu modelo produce, pero **indexa** una versión reducida. Los índices son los que tienen que caber en memoria, no la columna.

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE documents (
    id          bigserial   PRIMARY KEY,
    tenant_id   bigint      NOT NULL,
    title       text        NOT NULL,
    body        text        NOT NULL,
    lang        text        NOT NULL DEFAULT 'es',
    published   boolean     NOT NULL DEFAULT true,
    deleted_at  timestamptz,
    embedding   vector(1536) NOT NULL,
    created_at  timestamptz NOT NULL DEFAULT now()
);
```

Un límite que conviene conocer antes de elegir modelo: el tipo `vector` se puede indexar hasta 2.000 dimensiones, `halfvec` hasta 4.000 y `bit` hasta 64.000. Si tu modelo devuelve 3.072 dimensiones, no puedes crear un índice HNSW directamente sobre la columna `vector`; tendrás que indexar la expresión `embedding::halfvec(3072)`. Muchos modelos modernos soportan además Matryoshka: puedes truncar el vector a 512 o 1024 dimensiones y perder muy poco. Compruébalo con tus propios datos antes de asumirlo.

2. HNSW e IVFFlat: qué hace realmente cada uno

Ambos índices resuelven el mismo problema (evitar comparar tu consulta contra cada fila) con filosofías opuestas.

IVFFlat divide el espacio en `lists` celdas mediante k-means, guarda cada vector en la celda de su centroide más cercano y, en tiempo de consulta, sólo mira las `ivfflat.probes` celdas más prometedoras. Se construye rápido y ocupa poco, pero tiene un defecto operativo serio: los centroides se calculan una vez, con los datos que había en ese momento. Si la distribución de tus embeddings cambia (idiomas nuevos, un dominio nuevo, otro modelo), el índice se degrada y hay que reconstruirlo. Además, necesita datos ya cargados para entrenar: crear un IVFFlat sobre una tabla vacía produce un índice inútil.

HNSW construye un grafo navegable de varias capas. Las capas superiores son atajos de larga distancia; las inferiores, vecindarios densos. Una búsqueda entra por arriba, desciende y explora. Es más lento de construir y ocupa bastante más, pero da mejor recall a la misma latencia, acepta inserciones incrementales sin degradarse de golpe y no necesita entrenamiento previo. En 2026, para la mayoría de casos de producción, HNSW es la opción por defecto y IVFFlat la excepción justificada (conjuntos enormes, RAM escasa, tolerancia a

reconstrucciones periódicas).

Los tres parámetros que importan en HNSW:

- `m` (por defecto 16): número de conexiones por nodo. Más conexiones, mejor recall y más memoria. Subir a 32 o 48 tiene sentido en espacios de alta dimensionalidad donde el recall se resiste; por debajo de 16 casi nunca compensa.
- `ef_construction` (por defecto 64): cuántos candidatos se examinan al insertar cada nodo. Afecta a la calidad del grafo y al tiempo de construcción, no al tamaño. Para embeddings de 1536 dimensiones, 128–200 es un punto de partida mucho más sensato que el valor por defecto.
- `hnsw.ef_search` (por defecto 40): cuántos candidatos se exploran *en cada consulta*. Es el único de los tres que puedes cambiar sin reconstruir nada, y es tu mando principal para negociar recall contra latencia.

Ese 40 por defecto es el origen de la mitad de los artículos que afirman que «pgvector es lento» o «pgvector tiene mal recall». No es pgvector: es que nadie tocó el mando.

3. Construir el índice sin perder la tarde

Crear un HNSW sobre millones de filas puede tardar horas o veinte minutos, y la diferencia está en dos ajustes que casi nadie configura.

El primero es `maintenance_work_mem`. pgvector construye el grafo en memoria si cabe; si no cabe, pasa a un modo en disco que es un orden de magnitud más lento. Que se te desborde la memoria no da error, sólo aparece un aviso en el log y la construcción se arrastra. El segundo es el paralelismo: pgvector soporta construcción paralela desde 0.6, controlada por `max_parallel_maintenance_workers`.

```
-- Sólo para esta sesión; no toques postgresql.conf por un build puntual.
SET maintenance_work_mem = '8GB';
SET max_parallel_maintenance_workers = 7;   -- 7 workers + el proceso líder = 8

CREATE INDEX documents_embedding_hnsw
    ON documents
    USING hnsw (embedding vector_cosine_ops)
    WITH (m = 16, ef_construction = 128);
```

Hay una trampa importante aquí: `CREATE INDEX CONCURRENTLY` **no usa trabajadores paralelos**. Es una limitación de PostgreSQL, no de pgvector. Tienes que elegir entre construir rápido bloqueando las escrituras sobre la tabla, o construir en línea pagando el precio en tiempo. En una tabla de cuatro millones de vectores, la diferencia puede ser entre veinte minutos con lock y tres horas sin él. Si tienes ventana de mantenimiento, úsala; si no, lanza el `CONCURRENTLY` y ten paciencia.

```
-- Vigila el progreso desde otra sesión
SELECT phase,
       round(100.0 * blocks_done  / NULLIF(blocks_total, 0), 1)  AS pct_blocks,
       round(100.0 * tuples_done  / NULLIF(tuples_total, 0), 1)  AS pct_tuples
FROM pg_stat_progress_create_index;
```

Dos avisos de operación. Si corres PostgreSQL en Docker o Kubernetes, la construcción paralela usa memoria compartida: `--shm-size` (o un volumen `emptyDir` con `medium: Memory`) debe ser al menos tan grande como tu `maintenance_work_mem`, o el build falla a mitad de camino. Y si vas a cargar millones de filas desde cero, carga primero y crea el índice después: insertar cuatro millones de vectores contra un HNSW existente es mucho más lento que construir el grafo de una vez.

4. Recall: medirlo, no creerlo

Un índice ANN es una aproximación. La pregunta relevante no es «¿es rápido?» sino «¿cuántos de los diez vecinos reales me está devolviendo?». Y esa pregunta tiene una respuesta numérica que puedes calcular en veinte minutos, así que no hay excusa para adivinarla.

El método es directo: para un conjunto de consultas de muestra, calcula los K vecinos exactos desactivando el índice, calcula los K vecinos que devuelve el índice con distintos `ef_search`, y compara los conjuntos.

```
import statistics

import psycopg
from pgvector.psycopg import register_vector

DSN = "postgresql://app@localhost/rag"
K = 10        # el top-K que sirve tu API
SAMPLE = 200  # consultas de prueba

conn = psycopg.connect(DSN)
register_vector(conn)

# Usa embeddings reales como consultas: los vectores sintéticos mienten.
with conn.cursor() as cur:
    cur.execute(
        "SELECT embedding FROM documents TABLESAMPLE SYSTEM (1) LIMIT %s",
        (SAMPLE,),
    )
    queries = [row[0] for row in cur.fetchall()]


def top_k(cur, q):
    cur.execute(
        "SELECT id FROM documents ORDER BY embedding <=> %s LIMIT %s", (q, K)
    )
    return {row[0] for row in cur.fetchall()}


# Verdad de referencia: una sola vez, es lo caro.
ground_truth = []
for q in queries:
    with conn.transaction(), conn.cursor() as cur:
        cur.execute("SET LOCAL enable_indexscan = off")
        ground_truth.append(top_k(cur, q))

for ef in (20, 40, 80, 160, 320):
    scores = []
    for q, exact in zip(queries, ground_truth):
        with conn.transaction(), conn.cursor() as cur:
            # SET no admite parámetros: interpola un entero validado.
            cur.execute(f"SET LOCAL hnsw.ef_search = {int(ef)}")
            approx = top_k(cur, q)
        scores.append(len(exact & approx) / K)
    print(f"ef_search={ef:<4} recall@{K} = {statistics.mean(scores):.3f}")
```

Una salida típica sobre embeddings de 1536 dimensiones y unos pocos millones de filas se parece a esto:

```
ef_search=20   recall@10 = 0.842
ef_search=40   recall@10 = 0.913
ef_search=80   recall@10 = 0.961
ef_search=160  recall@10 = 0.984
ef_search=320  recall@10 = 0.993
```

Los números concretos dependerán de tus datos, pero la forma de la curva casi siempre es la misma: crece deprisa y luego se aplana. Ahí está la decisión de producto. Para un chatbot de soporte, 0,96 con 80 candidatos probablemente sobra. Para búsqueda legal o médica, quizá necesites 0,99 y estés dispuesto a

pagar la latencia. Lo importante es que sea una decisión consciente y no el valor por defecto de una librería.

Añade esta medición a tu CI. Un cambio de modelo de embeddings, una migración de datos o un ajuste de `m` pueden mover el recall sin mover ni la latencia ni ninguna métrica que estés vigilando.

```
-- Ajuste por consulta, no global: cada endpoint puede tener su compromiso.
BEGIN;
SET LOCAL hnsw.ef_search = 100;
SELECT id, title FROM documents ORDER BY embedding <=> :q LIMIT 10;
COMMIT;
```

5. El fallo silencioso: filtros más búsqueda aproximada

Este es el problema que más daño hace y el que menos se detecta. Imagina la consulta más natural del mundo:

```
SET hnsw.ef_search = 40;   -- valor por defecto

SELECT id, title
FROM documents
WHERE lang = 'es' AND published
ORDER BY embedding <=> :q
LIMIT 10;
```

PostgreSQL recorre el índice HNSW pidiendo candidatos y aplica el `WHERE` *después*. Con `ef_search = 40` el índice entrega unos 40 candidatos; si sólo el 10% de tus documentos están en español y publicados, cuatro sobrevivirán al filtro. Has pedido diez resultados y te llegan cuatro. La consulta es correcta, rápida y devuelve menos de lo que pediste sin decirte nada.

pgvector 0.8 introdujo los *iterative scans* precisamente para esto: en lugar de traer un lote fijo y filtrar, el índice sigue pidiendo candidatos hasta reunir suficientes filas que pasen el filtro.

```
-- 'strict_order': resultados estrictamente ordenados por distancia.
SET hnsw.iterative_scan = 'strict_order';

-- Techo de trabajo para que un filtro muy selectivo no barra la tabla entera.
SET hnsw.max_scan_tuples = 20000;      -- por defecto 20000
SET hnsw.scan_mem_multiplier = 2;      -- por defecto 1; súbelo si el scan se queda corto

SELECT id, title
FROM documents
WHERE lang = 'es' AND published
ORDER BY embedding <=> :q
LIMIT 10;
```

Existe un segundo modo, `relaxed_order`, más rápido porque no garantiza que las filas salgan perfectamente ordenadas por distancia. Si te importa el orden exacto (y en un RAG te importa, porque normalmente recortas por posición), envuélvelo y reordena fuera:

```
SET hnsw.iterative_scan = 'relaxed_order';

WITH candidates AS MATERIALIZED (
    SELECT id, title, embedding <=> :q AS distance
    FROM documents
    WHERE lang = 'es' AND published
    ORDER BY embedding <=> :q
    LIMIT 10
)
SELECT id, title, distance
```

```
FROM candidates
ORDER BY distance;
```

El `MATERIALIZED` no es decorativo: obliga a PostgreSQL a ejecutar la subconsulta tal cual, en lugar de aplanarla y perder el efecto del reordenado.

Para IVFFlat el equivalente es `ivfflat.iterative_scan = 'relaxed_order'` junto con `ivfflat.max_probes`; IVFFlat no ofrece `strict_order`.

Cuando el filtro es fijo, no lo pagues en tiempo de consulta

Los iterative scans arreglan la corrección, pero siguen costando trabajo. Si un filtro aparece en el 100% de tus consultas, mételo en el índice:

```
CREATE INDEX CONCURRENTLY documents_embedding_live_hnsw
    ON documents
    USING hnsw (embedding vector_cosine_ops)
    WITH (m = 16, ef_construction = 128)
    WHERE published AND deleted_at IS NULL;
```

El índice parcial sólo contiene filas vivas, así que el grafo es más pequeño, cabe mejor en RAM y no desperdicia candidatos en documentos que vas a descartar. Para que el planificador lo use, el `WHERE` de tu consulta debe implicar el del índice.

Y si tu filtro dominante es el tenant en una aplicación multi-tenant, la respuesta correcta no es un índice parcial por cliente (no escala a mil clientes) sino particionar la tabla:

```
CREATE TABLE documents (
    id        bigserial,
    tenant_id bigint       NOT NULL,
    embedding vector(1536) NOT NULL,
    -- ...
    PRIMARY KEY (id, tenant_id)
) PARTITION BY HASH (tenant_id);

CREATE TABLE documents_p0 PARTITION OF documents FOR VALUES WITH (MODULUS 8, REMAINDER 0);
-- ... p1 a p7

-- Un HNSW por partición: grafos más pequeños y builds paralelizables.
CREATE INDEX ON documents_p0 USING hnsw (embedding vector_cosine_ops);
```

La condición para que esto funcione es que **toda** consulta incluya `tenant_id`, de modo que el planificador pode las particiones. Verifícalo con `EXPLAIN`: si ves las ocho particiones en el plan, no estás podando y acabas de multiplicar el trabajo por ocho.

6. Cuantización: dividir el índice entre 2 o entre 32

Un índice HNSW rinde cuando cabe en memoria. Si el grafo ocupa 40 GB y tu instancia tiene 32 GB de RAM, cada consulta acaba leyendo del disco y la latencia se dispara de forma errática: rápida cuando hay suerte con la caché, lentísima cuando no. La cuantización ataca ese problema por la raíz reduciendo lo que se indexa.

halfvec: mitad de tamaño, prácticamente el mismo recall

Pasar de float32 a float16 divide el índice por dos y, con embeddings normalizados de modelos actuales, la pérdida de recall suele ser de décimas de punto. Es el cambio con mejor relación beneficio/riesgo de toda la guía. No conviertas la columna: indexa la expresión.

```
CREATE INDEX CONCURRENTLY documents_embedding_half_hnsw
    ON documents
    USING hnsw ((embedding::halfvec(1536)) halfvec_cosine_ops)
    WITH (m = 16, ef_construction = 128);

-- La consulta DEBE repetir la expresión exacta, o el índice se ignora.
SELECT id, title
FROM documents
ORDER BY embedding::halfvec(1536) <=> (:q)::halfvec(1536)
LIMIT 10;
```

Esta es una de esas cosas que se rompen en silencio: si en la consulta escribes `embedding <=> :q` en lugar de la expresión con el cast, PostgreSQL no puede usar el índice de expresión y cae a un escaneo secuencial. Rápido en desarrollo, catastrófico en producción. Compruébalo siempre con `EXPLAIN`.

Cuantización binaria más reranking: el patrón de dos fases

La cuantización binaria convierte cada dimensión en un solo bit según su signo: positivo es 1, el resto 0. Suena brutal, y lo es: un vector de 1536 dimensiones pasa de 6 KB a 192 bytes, 32 veces menos. Por sí sola pierde demasiada precisión para servir resultados finales, pero es excelente como *primer filtro*. La receta es recuperar muchos candidatos con el índice binario y reordenarlos con los vectores completos.

```
CREATE INDEX CONCURRENTLY documents_embedding_bit_hnsw
    ON documents
    USING hnsw ((binary_quantize(embedding)::bit(1536)) bit_hamming_ops);

-- Fase 1: 200 candidatos con distancia de Hamming sobre el índice binario.
-- Fase 2: reordenar esos 200 con la distancia coseno real.
SELECT id, title, distance
FROM (
    SELECT id, title, embedding <=> (:q)::vector(1536) AS distance
    FROM documents
    ORDER BY binary_quantize(embedding)::bit(1536)
             <~> binary_quantize((:q)::vector(1536))::bit(1536)
    LIMIT 200
) AS candidates
ORDER BY distance
LIMIT 10;
```

El parámetro que hay que calibrar es el `LIMIT` de la fase 1, el llamado factor de sobremuestreo. Con 10 resultados finales, empezar por 200 candidatos (20x) es razonable; mide el recall con el mismo script de la sección 4 y ajusta. Si con 100 ya llegas a tu objetivo, ahorra el trabajo.

Un aviso honesto: la cuantización binaria funciona bien con modelos entrenados o evaluados para resistirla (Cohere Embed v3, mxbai-embed-large, Jina v3, entre otros) y puede degradarse mucho con modelos que no lo están. Nunca la adoptes sin medir el recall sobre tus propios datos y tus propias consultas.

Resumen de compromisos con 1536 dimensiones:

- **vector** — referencia. Máxima precisión, máximo tamaño, índice grande.
- **halfvec** — mitad de índice, pérdida de recall casi nula. Adóptalo por defecto salvo que midas lo contrario.
- **bit + reranking** — índice 32 veces menor a cambio de una segunda fase y de dependencia del modelo. Reserva esta opción para cuando el índice ya no cabe en RAM.

7. Consultas que sí usan el índice

Un índice vectorial sólo entra en juego bajo condiciones bastante estrictas, y salirse de ellas no produce ningún aviso.

- **El operador debe coincidir con la clase de operadores del índice.** Un índice creado con `vector_cosine_ops` sólo sirve para `<=>`. Si consultas con `<->` (L2) tendrás un escaneo secuencial. Es el error más común y el más fácil de cometer al copiar código de un tutorial.
- **Necesitas `ORDER BY` con `LIMIT`.** Sin límite, PostgreSQL tiene que ordenar la tabla entera y el índice no aporta nada.
- **Nada de umbrales en el `WHERE`.** Un `WHERE embedding <=> :q < 0.4` no es una consulta de vecinos más cercanos. Recupera primero y filtra después.

```
-- MAL: no es una búsqueda de k vecinos, no usa el índice ANN.
SELECT id FROM documents WHERE embedding <=> :q < 0.4;

-- BIEN: recupera k, luego aplica el umbral.
SELECT id, distance
FROM (
    SELECT id, embedding <=> :q AS distance
    FROM documents
    ORDER BY embedding <=> :q
    LIMIT 100
) t
WHERE distance < 0.4;
```

Y verifica siempre el plan. Un `Index Scan using documents_embedding_hnsw` es lo que buscas; un `Seq Scan` seguido de `Sort` significa que algo no encaja.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id FROM documents ORDER BY embedding <=> :q LIMIT 10;
```

Una nota sobre distancias: si tus embeddings están normalizados a norma 1 (lo están los de OpenAI, Cohere y la mayoría de modelos de sentence-transformers), la similitud coseno y el producto interno ordenan igual, y `vector_ip_ops` con el operador `<#>` es algo más barato de calcular. Ten en cuenta que `<#>` devuelve el producto interno *negado*, para que «menor es mejor» siga cumpliéndose. Si no estás seguro de que tus vectores estén normalizados, quédate con coseno: es la opción segura.

8. Operar el índice en el tiempo

El día uno todo va bien. Lo interesante empieza al sexto mes.

El grafo se degrada con la rotación. HNSW acepta inserciones y borrados sin reconstruirse, pero cada actualización de un embedding es en realidad un borrado más una inserción (MVCC), y los nodos muertos dejan huecos en el grafo. En tablas con mucha rotación, el recall baja lentamente y el índice crece. La solución es periódica y sin downtime:

```
REINDEX INDEX CONCURRENTLY documents_embedding_hnsw;
```

Vigila el tamaño frente a tu RAM. No hay una fórmula fiable para predecir el tamaño de un HNSW: depende de `m`, de las dimensiones y de la distribución de tus datos. Constrúyelo sobre una muestra del 5%, mide y extrapola.

```
SELECT indexrelname,
       pg_size_pretty(pg_relation_size(indexrelid)) AS index_size,
       idx_scan
FROM pg_stat_user_indexes
WHERE relname = 'documents'
ORDER BY pg_relation_size(indexrelid) DESC;
```

Calienta la caché tras un reinicio. El primer minuto después de un failover o un despliegue de la base de datos es el peor: el índice está frío y cada consulta lee de disco. `pg_prewarm` convierte ese problema en un paso de arranque.

```
CREATE EXTENSION IF NOT EXISTS pg_prewarm;
SELECT pg_prewarm('documents_embedding_hnsw');
```

Pon un techo por consulta. Con iterative scans activados, una consulta con un filtro patológicamente selectivo puede trabajar mucho más de lo que esperas. Un `statement_timeout` local es la red de seguridad.

Así queda una función de búsqueda que reúne todo lo anterior:

```
from dataclasses import dataclass

SEARCH_SQL = """
WITH candidates AS MATERIALIZED (
    SELECT id, title, body, embedding <=> %(q)s AS distance
    FROM documents
    WHERE tenant_id = %(tenant)s
      AND lang = %(lang)s
      AND published
      AND deleted_at IS NULL
    ORDER BY embedding <=> %(q)s
    LIMIT %(k)s
)
SELECT id, title, body, distance
FROM candidates
WHERE distance < %(max_distance)s
ORDER BY distance
"""


@dataclass(frozen=True)
class Hit:
    id: int
    title: str
    body: str
    distance: float


async def search(conn, q, *, tenant, lang, k=10, ef_search=100, max_distance=0.45):
    """Búsqueda vectorial filtrada, ordenada y acotada en tiempo."""
    async with conn.transaction():
        # SET LOCAL: los ajustes mueren con la transacción y no contaminan
        # la siguiente petición que reutilice esta conexión del pool.
        await conn.execute(f"SET LOCAL hnsw.ef_search = {int(ef_search)}")
        await conn.execute("SET LOCAL hnsw.iterative_scan = 'relaxed_order'")
        await conn.execute("SET LOCAL statement_timeout = '2s'")

        cur = await conn.execute(
            SEARCH_SQL,
            {
                "q": q,
                "tenant": tenant,
                "lang": lang,
                "k": k,
                "max_distance": max_distance,
            },
        )
        return [Hit(*row) for row in await cur.fetchall()]
```

El detalle del `SET LOCAL` merece un párrafo propio. Si usas `SET` a secas sobre una conexión de un pool, el ajuste sobrevive a tu petición y afecta a la siguiente, que puede ser de otro endpoint con otras necesidades. Con PgBouncer en modo *transaction* el resultado es aún más divertido de depurar, porque el reparto de conexiones cambia entre ejecuciones. `SET LOCAL` dentro de una transacción explícita elimina la clase entera de

problema.

9. Ocho errores que se repiten

1. **Dejar `hnsw.ef_search` en 40 y concluir que el recall es malo.** Mide la curva antes de opinar.
2. **Filtrar con `WHERE` sin activar iterative scans.** Devuelves menos resultados de los pedidos y nadie se entera.
3. **Usar un operador que no corresponde a la clase del índice.** `<->` contra un índice `vector_cosine_ops` es un Seq Scan disfrazado.
4. **Crear un índice de expresión y no repetir la expresión en la consulta.** El índice existe, ocupa disco y jamás se usa.
5. **Construir el índice antes de cargar los datos.** Mucho más lento, y en IVFFlat directamente inútil porque no hay nada con lo que entrenar los centroides.
6. **Olvidar `maintenance_work_mem`.** El build cae al modo en disco y tarda horas sin decir por qué.
7. **Esperar paralelismo con `CREATE INDEX CONCURRENTLY`.** No lo hay; planifica el tiempo en consecuencia.
8. **Adoptar cuantización binaria sin reranking ni medición.** Es un filtro de primera fase, no un sustituto del vector completo.

10. Checklist de producción

- pgvector 0.8+ instalado y verificado con `SELECT extversion FROM pg_extension WHERE extname = 'vector'`.
- Recall@K medido sobre consultas reales y documentado junto al valor de `ef_search` elegido.
- La medición de recall corre en CI y falla el build si cae por debajo del umbral acordado.
- `hnsw.iterative_scan` configurado en toda consulta que combine `WHERE` con orden por distancia.
- Filtros permanentes trasladados a índices parciales; el filtro por tenant, a particiones.
- Ajustes por consulta con `SET LOCAL` dentro de una transacción, nunca con `SET` global.
- `statement_timeout` en el camino de búsqueda.
- Tamaño del índice monitorizado y comparado con la RAM disponible; alerta cuando supere el 70%.
- `pg_prewarm` integrado en el arranque o en el procedimiento post-failover.
- `REINDEX CONCURRENTLY` programado si la tabla tiene rotación alta.
- `EXPLAIN` revisado para cada consulta vectorial nueva antes de desplegarla.

11. El presupuesto de memoria: dimensiona la RAM antes de elegir la máquina

La pregunta que más veces llega tarde es sencilla: ¿cuánta RAM necesita esto? Se responde tarde porque el prototipo cabía en cualquier sitio y nadie hizo la cuenta. La cuenta existe y es aritmética de servilleta.

Un índice HNSW guarda, por cada fila, el vector completo más los enlaces del grafo. Una estimación conservadora que funciona bien en la práctica es `N x D x 4 bytes x 2`: el primer factor es el vector en coma flotante de 32 bits, el segundo cubre el grafo y la sobrecarga de página. Para cinco millones de filas de 1.536 dimensiones eso son unos 60 GB. No es un error de tipeo, y no es el tamaño de la tabla: es sólo el índice.

Ese número cambia tu arquitectura antes que cualquier parámetro. Con `halfvec` pasas a 2 bytes por dimensión y el índice baja a unos 30 GB. Con cuantización binaria y reranking en dos fases, el índice de bits ocupa unos 2 GB y los vectores originales viven en la tabla, donde se leen sólo para las pocas filas finalistas.

Mide en lugar de estimar en cuanto tengas datos reales, aunque sea una muestra:

```
SELECT
  pg_size_pretty(pg_relation_size('documentos'))              AS tabla,
  pg_size_pretty(pg_relation_size('documentos_embedding_idx')) AS indice_hnsw,
  pg_size_pretty(pg_total_relation_size('documentos'))          AS total,
  (SELECT count(*) FROM documentos)                             AS filas;
```

Extrapola linealmente: si 500.000 filas dan 6 GB de índice, diez millones darán unos 120 GB. La linealidad se cumple bien porque el grafo tiene grado acotado por `m`.

Con el tamaño en la mano, la regla operativa es corta: el índice debe caber en memoria. No necesariamente en `shared_buffers`, pero sí en la suma de `shared_buffers` más la caché de página del sistema operativo. Cuando no cabe, cada salto del grafo que falla en caché se convierte en una lectura aleatoria de disco, y una búsqueda HNSW hace decenas o cientos de saltos. Ahí es donde una consulta de 8 ms pasa a 400 ms sin que ningún parámetro haya cambiado.

```
# postgresql.conf en una máquina de 128 GB dedicada a búsqueda vectorial
shared_buffers = 32GB          # 25 % de la RAM; el resto lo gestiona el SO
effective_cache_size = 96GB    # pista al planificador, no reserva memoria
work_mem = 64MB                # por nodo de plan; cuidado con la concurrencia
maintenance_work_mem = 24GB    # sólo para construir indices, ver seccion 3
max_parallel_maintenance_workers = 7
```

Después de un reinicio o de un failover, la caché está vacía y las primeras consultas pagan el precio completo. `pg_prewarm` convierte ese arranque frío en un paso de despliegue en vez de en una alerta:

```
CREATE EXTENSION IF NOT EXISTS pg_prewarm;
SELECT pg_prewarm('documentos_embedding_idx');
```

Comprobar si estás leyendo de disco es directo. Ejecuta la consulta con `EXPLAIN (ANALYZE, BUFFERS)` y mira `shared read`: si es un número alto y persistente después de varias ejecuciones, el índice no cabe y ninguna cantidad de `hnsw.ef_search` lo va a arreglar. Lo que lo arregla es más RAM, `halfvec`, cuantización binaria o particionado.

12. Búsqueda híbrida: el vector solo no encuentra "ORD-2026-88131"

Los embeddings son buenos entendiendo intenciones y malos recordando cadenas literales. Si un usuario busca el identificador de un pedido, el nombre de una función interna o un código de error, la búsqueda semántica devuelve documentos parecidos en tema y ninguno con la cadena exacta. Es el fallo más frecuente en buscadores internos y no se arregla subiendo `hnsw.ef_search`.

La solución no es elegir entre semántica y léxica, sino fusionar las dos listas. PostgreSQL ya trae la mitad léxica con `tsvector`, así que no hay infraestructura nueva que operar.

```
ALTER TABLE documentos
  ADD COLUMN fts tsvector
  GENERATED ALWAYS AS (to_tsvector('spanish', coalesce(titulo,'') || ' ' || coalesce(cuerpo,'')))
STORED;

CREATE INDEX documentos_fts_idx ON documentos USING gin (fts);
```

Para combinar los dos rankings, el método que gana casi siempre es también el más aburrido: Reciprocal Rank Fusion. No compara puntuaciones (que viven en escalas incomparables: una distancia coseno y un `ts_rank` no se pueden sumar), sino posiciones. Cada documento recibe `1 / (k + posición)` en cada lista y se suman los

aportes. La constante `k`, tradicionalmente 60, amortigua la diferencia entre los primeros puestos y evita que un único retriever con mucha confianza domine el resultado.

```
WITH semantica AS (
  SELECT id,
         ROW_NUMBER() OVER (ORDER BY embedding <=> $1) AS pos
  FROM documentos
  ORDER BY embedding <=> $1
  LIMIT 50
),
lexica AS (
  SELECT id,
         ROW_NUMBER() OVER (ORDER BY ts_rank_cd(fts, q) DESC) AS pos
  FROM documentos, websearch_to_tsquery('spanish', $2) AS q
  WHERE fts @@ q
  ORDER BY ts_rank_cd(fts, q) DESC
  LIMIT 50
)
SELECT d.id, d.titulo,
       COALESCE(1.0 / (60 + s.pos), 0) + COALESCE(1.0 / (60 + l.pos), 0) AS rrf
FROM documentos d
LEFT JOIN semantica s ON s.id = d.id
LEFT JOIN lexica    l ON l.id = d.id
WHERE s.id IS NOT NULL OR l.id IS NOT NULL
ORDER BY rrf DESC
LIMIT 10;
```

Dos detalles importan más de lo que parece. El primero: cada rama pide 50 candidatos aunque devuelvas 10, porque la fusión necesita profundidad para tener algo que fusionar; con listas de 10 el resultado se parece demasiado a la intersección. El segundo: el `LIMIT` dentro del CTE semántico es lo que permite usar el índice ANN, y si lo quitas para "no perder resultados" acabas con un Seq Scan que calcula distancias sobre toda la tabla.

Si quieres dar más peso a una de las señales, multiplica el aporte de esa rama en lugar de tocar `k`. Un factor de 1,5 en la rama léxica es suficiente para que los códigos exactos suban al primer puesto sin destruir el resto del ranking:

```
RRF_K = 60
PESO_SEMANTICO = 1.0
PESO_LEXICO = 1.5

def fusionar(semanticos: list[str], lexicos: list[str], limite: int = 10):
    """Fusiona dos listas de ids ya ordenadas. Util cuando una de las ramas
    no vive en PostgreSQL (un reranker externo, un motor de busqueda, etc.)."""
    puntos: dict[str, float] = {}
    for peso, lista in ((PESO_SEMANTICO, semanticos), (PESO_LEXICO, lexicos)):
        for posicion, doc_id in enumerate(lista, start=1):
            puntos[doc_id] = puntos.get(doc_id, 0.0) + peso / (RRF_K + posicion)
    return sorted(puntos, key=puntos.get, reverse=True)[:limite]
```

La mejora es medible y suele ser grande. En conjuntos de evaluación internos es habitual pasar de una precisión de recuperación en torno al 60 % con vectores solos a más del 80 % con la fusión, y la ganancia se concentra justo donde más duele: consultas con nombres propios, referencias y códigos.

13. Cambiar de modelo de embeddings sin downtime

Antes o después vas a querer cambiar de modelo: sale uno mejor, uno más barato, o el proveedor deprecia el que usas. El problema es que los vectores de dos modelos distintos no son comparables. No hay conversión, no hay migración incremental de valores: hay que recalcular todo. Y mientras recalculas, el buscador tiene que seguir funcionando.

El patrón es el mismo expand-contract que usarías para cambiar el tipo de una columna, con la diferencia de que aquí el backfill cuesta dinero de API.

```
-- Expand: la columna nueva convive con la vieja
ALTER TABLE documentos ADD COLUMN embedding_v2 halfvec(3072);

-- Sin indice todavia: construirlo antes del backfill significa
-- pagar el coste de insercion en el grafo fila a fila.
```

El backfill va por lotes, con reintentos y sin bloquear escrituras. La clave es avanzar por clave primaria en lugar de usar `OFFSET`, y hacer commit por lote para que un fallo a mitad no obligue a empezar de cero:

```
import time
import psycopg
from openai import OpenAI

TAMANO_LOTE = 256
cliente = OpenAI()

def backfill(dsn: str) -> None:
    with psycopg.connect(dsn, autocommit=False) as conn:
        ultimo_id = "00000000-0000-0000-0000-000000000000"
        while True:
            with conn.cursor() as cur:
                cur.execute(
                    """SELECT id, cuerpo FROM documentos
                       WHERE id > %s AND embedding_v2 IS NULL
                       ORDER BY id LIMIT %s""",
                    (ultimo_id, TAMANO_LOTE),
                )
                filas = cur.fetchall()

            if not filas:
                break

            textos = [cuerpo for _, cuerpo in filas]
            respuesta = cliente.embeddings.create(
                model="text-embedding-3-large", input=textos
            )
            vectores = [d.embedding for d in respuesta.data]

            with conn.cursor() as cur:
                cur.executemany(
                    "UPDATE documentos SET embedding_v2 = %s WHERE id = %s",
                    [(v, fila[0]) for v, fila in zip(vectores, filas)],
                )
            conn.commit()
            ultimo_id = filas[-1][0]
            time.sleep(0.05)  # respeta el rate limit del proveedor
```

Con el backfill terminado se construye el índice, ya sobre datos completos y por tanto mucho más rápido que si se hubiera ido llenando fila a fila:

```
SET maintenance_work_mem = '24GB';
SET max_parallel_maintenance_workers = 7;
CREATE INDEX CONCURRENTLY documentos_embedding_v2_idx
  ON documentos USING hnsw (embedding_v2 halfvec_cosine_ops)
  WITH (m = 16, ef_construction = 128);
```

Antes de cambiar el tráfico, compara. Este es el paso que casi todo el mundo se salta y el que evita descubrir en producción que el modelo nuevo es peor para tu dominio concreto. Ejecuta tu conjunto de consultas de evaluación contra las dos columnas y mira recall y solapamiento de resultados. Un modelo con mejores números en un benchmark público puede rendir peor con tu vocabulario.

El corte en sí es una bandera de funcionalidad, no un despliegue. Mantén las dos columnas durante unos días con el porcentaje de tráfico que quieras en cada una, vigila latencia y métricas de calidad, y sólo entonces:

```
-- Contract: cuando el modelo nuevo lleva dias estable
DROP INDEX CONCURRENTLY documentos_embedding_idx;
ALTER TABLE documentos DROP COLUMN embedding;
ALTER TABLE documentos RENAME COLUMN embedding_v2 TO embedding;
```

Un aviso sobre el coste: recalcular embeddings de diez millones de fragmentos no es gratis ni instantáneo. Haz la cuenta con el precio por millón de tokens del proveedor antes de aprobar la migración, y considera si de verdad necesitas 3.072 dimensiones o si el modelo admite reducir dimensionalidad en la propia llamada a la API, que es la forma más barata de recortar índice y factura a la vez.

14. Escrituras: lo que las actualizaciones le hacen a tu grafo

Casi toda la literatura sobre HNSW asume un índice estático. Tu tabla no lo es: llegan documentos nuevos, se reindexan documentos editados y se borran los que caducan. Merece la pena entender qué pasa por dentro, porque el modo de fallo es lento y silencioso.

Cuando insertas una fila, pgvector recorre el grafo para encontrarle vecinos y la enlaza. Es una operación de búsqueda completa por cada inserción, y por eso cargar un millón de filas con el índice ya creado es órdenes de magnitud más lento que crear el índice al final. Regla práctica: en cargas masivas, borra el índice, carga y reconstruye.

Cuando borras o actualizas una fila, PostgreSQL marca la versión antigua como muerta y el índice conserva su entrada hasta que `VACUUM` pasa por ahí. Mientras tanto, esas entradas siguen ocupando espacio en el grafo y siguen visitándose durante la búsqueda, sólo para descartarse al comprobar la visibilidad en la tabla. Es trabajo pagado sin resultado: la búsqueda hace los mismos saltos pero devuelve menos candidatos útiles, lo que se manifiesta como recall que baja poco a poco a lo largo de semanas.

En una tabla vectorial con rotación alta conviene ser más agresivo que con los valores por defecto:

```
ALTER TABLE documentos SET (
  autovacuum_vacuum_scale_factor = 0.02,   -- 2 % en lugar del 20 % por defecto
  autovacuum_vacuum_cost_limit   = 2000,   -- mas presupuesto de E/S
  autovacuum_analyze_scale_factor = 0.05
);
```

Aun así, `VACUUM` recupera espacio pero no reconstruye el grafo. Los enlaces que apuntaban a nodos eliminados se reparan de forma local, y con suficiente rotación la calidad del grafo se degrada: los caminos se vuelven menos directos y hacen falta más visitas para el mismo recall. La única cura real es reconstruir:

```
REINDEX INDEX CONCURRENTLY documentos_embedding_idx;
```

¿Cada cuánto? No hay una respuesta universal, pero hay una señal fiable: mide el recall periódicamente con el script de la sección 4 y reconstruye cuando caiga por debajo de tu umbral. Como orientación, una tabla en la que se reescribe el 20 % de las filas al mes suele pedir una reconstrucción trimestral; una tabla que sólo crece no la necesita casi nunca.

Vigila también el crecimiento del índice frente al de la tabla. Si el índice crece más rápido que las filas, tienes bloat:

```
SELECT
  pg_size_pretty(pg_relation_size('documentos_embedding_idx')) AS tamano_indice,
  n_live_tup, n_dead_tup,
```

```
  round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS pct_muertas,
  last_autovacuum
FROM pg_stat_user_tables
WHERE relname = 'documentos';
```

Un último patrón que ahorra mucho dolor: no reescribas el embedding si el texto no ha cambiado. Guarda un hash del contenido y compara antes de llamar a la API y antes de tocar la columna vectorial. Es una comprobación de una línea que elimina la mayor parte de las escrituras en pipelines de reindexación periódica.

```
ALTER TABLE documentos ADD COLUMN cuerpo_hash text;

UPDATE documentos SET embedding = $1, cuerpo_hash = $2
WHERE id = $3 AND cuerpo_hash IS DISTINCT FROM $2;
```

15. Observabilidad: qué medir en un buscador vectorial

Un buscador vectorial falla de una forma incómoda: sigue devolviendo diez resultados, con buena pinta, en pocos milisegundos. No hay excepción, no hay error 500, no hay alerta. Simplemente los resultados son peores. Si tu instrumentación sólo mide latencia y tasa de error, no vas a enterarte.

Hay tres capas que merecen métricas y son distintas entre sí.

Latencia por etapa. Una consulta RAG no es una consulta: es generar el embedding, buscar, opcionalmente reordenar y luego llamar al modelo. Medir el total esconde cuál de las cuatro se ha degradado. En más de una investigación la culpable ha resultado ser la llamada a la API de embeddings, no la base de datos.

```
import time
from contextlib import contextmanager
from prometheus_client import Histogram, Gauge, Counter

LATENCIA = Histogram(
    "busqueda_etapa_segundos", "Latencia por etapa", ["etapa"],
    buckets=(0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5),
)
RECALL = Gauge("busqueda_recall_at_10", "Recall@10 medido por el canario")
VACIAS = Counter("busqueda_sin_resultados_total", "Consultas que no devolvieron nada")

@contextmanager
def medir(etapa: str):
    inicio = time.perf_counter()
    try:
        yield
    finally:
        LATENCIA.labels(etapa=etapa).observe(time.perf_counter() - inicio)

def buscar(texto: str, k: int = 10):
    with medir("embedding"):
        vector = generar_embedding(texto)
    with medir("ann"):
        filas = consultar_pgvector(vector, k)
    if not filas:
        VACIAS.inc()
    with medir("rerank"):
        return reordenar(texto, filas)
```

Recall en producción. El recall que mediste en el portátil describe el índice de aquel día. Monta un canario: un trabajo programado que cada hora coge una muestra de consultas reales, ejecuta la búsqueda aproximada y la exacta (con `enable_indexscan = off` o con un `ORDER BY` sin índice sobre una muestra) y publica el solapamiento como métrica. Es barato, corre fuera de la ruta crítica y es la única forma de detectar una degradación gradual antes que tus usuarios.

```
def canario_recall(consultas: list[str], k: int = 10) -> float:
    aciertos = 0
    for texto in consultas:
        vector = generar_embedding(texto)
        aproximados = {f.id for f in consultar_pgvector(vector, k)}
        exactos = {f.id for f in consultar_exacto(vector, k)}
        aciertos += len(aproximados & exactos)
    valor = aciertos / (len(consultas) * k)
    RECALL.set(valor)
    return valor
```

Lo que dice PostgreSQL. `pg_stat_statements` te da el p95 real por consulta normalizada y, más útil todavía, la relación entre bloques leídos y bloques encontrados en caché. Un salto en `shared_blks_read` sobre la consulta de búsqueda es la firma inequívoca de un índice que ha dejado de caber en memoria.

```
SELECT
  calls,
  round(mean_exec_time::numeric, 2)   AS media_ms,
  round((total_exec_time / 1000)::numeric, 1) AS total_s,
  shared_blks_hit, shared_blks_read,
  round(100.0 * shared_blks_hit / NULLIF(shared_blks_hit + shared_blks_read, 0), 2) AS pct_cache
FROM pg_stat_statements
WHERE query ILIKE '%<=>%'
ORDER BY total_exec_time DESC
LIMIT 5;
```

Con esas tres capas, tres alertas bastan para cubrir casi todo: recall del canario por debajo del umbral, porcentaje de caché de la consulta vectorial cayendo, y proporción de búsquedas que devuelven menos de `k` resultados subiendo (la firma del sobrefiltrado de la sección 5).

16. Seguridad y multi-tenant: dos cosas que la mayoría deja para el final

Empecemos por la parte aburrida y urgente: **actualiza la extensión**. pgvector 0.8.2, publicado en febrero de 2026, corrige un desbordamiento de búfer en la construcción paralela de índices HNSW (CVE-2026-3172) que puede filtrar datos de otras relaciones o tumbar el servidor. Si construyes índices con trabajadores en paralelo —y la sección 3 te recomienda hacerlo— estás justo en el camino afectado.

```
SELECT extversion FROM pg_extension WHERE extname = 'vector';
-- tras actualizar el paquete del sistema o la imagen:
ALTER EXTENSION vector UPDATE;
```

El aislamiento entre inquilinos tiene un giro específico de la búsqueda aproximada que sorprende a mucha gente. Row-Level Security es la forma correcta de garantizar que un inquilino no vea filas de otro, pero desde el punto de vista del índice ANN una política RLS *es un filtro*, con exactamente el mismo problema de sobrefiltrado de la sección 5: el índice devuelve los candidatos más cercanos globalmente y la política descarta los que no son del inquilino. Con muchos inquilinos, la consulta puede quedarse sin resultados aunque el inquilino tenga miles de documentos relevantes.

```
ALTER TABLE documentos ENABLE ROW LEVEL SECURITY;
ALTER TABLE documentos FORCE ROW LEVEL SECURITY;

CREATE POLICY aislamiento_inquilino ON documentos
  USING (tenant_id = (SELECT current_setting('app.tenant_id', true)::uuid));
```

Fíjate en el `(SELECT ...)` alrededor de `current_setting`: convierte la llamada en un InitPlan que se evalúa una vez por consulta en lugar de una vez por fila. Sin ese paréntesis, la política se vuelve el cuello de botella antes que el propio índice.

Para que la búsqueda siga siendo correcta bajo RLS tienes dos opciones buenas, y conviene elegir según el reparto de tamaños. Si tienes pocas decenas de inquilinos grandes, un índice parcial por inquilino da resultados exactos dentro de su partición del grafo. Si tienes miles de inquilinos, particiona por hash y deja que el pruning haga el trabajo:

```
-- Miles de inquilinos: particionado por hash sobre tenant_id
CREATE TABLE documentos (
  id uuid PRIMARY KEY,
  tenant_id uuid NOT NULL,
  cuerpo text,
  embedding halfvec(1536)
) PARTITION BY HASH (tenant_id);

CREATE TABLE documentos_p0 PARTITION OF documentos FOR VALUES WITH (MODULUS 8, REMAINDER 0);
-- ... p1 a p7

-- Un indice HNSW por particion; el planificador poda por tenant_id
CREATE INDEX ON documentos_p0 USING hnsw (embedding halfvec_cosine_ops);
```

La tercera opción, y la más simple si tu escala lo permite, es activar los iterative scans de la sección 5 con `relaxed_order` y un `max_scan_tuples` generoso. Funciona, pero paga latencia proporcional a lo pequeño que sea el inquilino.

Un detalle de operación que rompe cosas en producción y nunca en local: `SET hnsw.ef_search` y `set_config('app.tenant_id', ...)` a nivel de sesión se filtran entre peticiones cuando hay un pool en modo transacción, como PgBouncer. Usa siempre la variante transaccional:

```
with conn.transaction():
    cur.execute("SELECT set_config('app.tenant_id', %s, true)", (tenant_id,))
    cur.execute("SET LOCAL hnsw.ef_search = 100")
    cur.execute(CONSULTA_ANN, (vector, k))
```

El tercer parámetro `true` de `set_config` y el `LOCAL` de `SET` son lo que ata la configuración a la transacción. Sin ellos, una petición del inquilino A puede acabar ejecutándose con el contexto del inquilino B.

17. Un banco de pruebas en CI que impide regresiones de recall

Todo lo anterior se pierde si nadie lo vuelve a comprobar. Los cambios que degradan un buscador vectorial no son los dramáticos: es alguien que baja `ef_search` para arreglar la latencia, alguien que añade un `WHERE` a la consulta, alguien que cambia el modelo de embeddings en un fichero de configuración. Ninguno rompe un test funcional.

Lo que sí los detecta es un test que mide recall contra un conjunto fijo y falla si baja. Necesita tres piezas: un dataset pequeño y reproducible, un conjunto de consultas doradas y un umbral.

```
import numpy as np
import psycopg
import pytest

DIMENSIONES = 384
N_FILAS = 20_000
UMBRAL_RECALL = 0.95

@pytest.fixture(scope="session")
def db(postgresql_dsn):
    """Carga un corpus sintetico determinista y construye el indice."""
    rng = np.random.default_rng(seed=42)
    vectores = rng.normal(size=(N_FILAS, DIMENSIONES)).astype(np.float32)
    vectores /= np.linalg.norm(vectores, axis=1, keepdims=True)
```

```
    with psycopg.connect(postgresql_dsn, autocommit=True) as conn:
        conn.execute("CREATE EXTENSION IF NOT EXISTS vector")
        conn.execute("DROP TABLE IF EXISTS bench")
        conn.execute(f"CREATE TABLE bench (id int PRIMARY KEY, v vector({DIMENSIONES}))")
        with conn.cursor().copy("COPY bench (id, v) FROM STDIN") as copy:
            for i, v in enumerate(vectores):
                copy.write_row((i, "[" + ",".join(map(str, v)) + "]"))
        conn.execute("SET maintenance_work_mem = '2GB'")
        conn.execute(
            "CREATE INDEX ON bench USING hnsw (v vector_cosine_ops)"
            " WITH (m = 16, ef_construction = 64)"
        )
        conn.execute("ANALYZE bench")
        yield conn, vectores

def recall_medido(conn, vectores, ef_search: int, k: int = 10) -> float:
    rng = np.random.default_rng(seed=7)
    consultas = vectores[rng.choice(len(vectores), size=100, replace=False)]
    total = 0
    for q in consultas:
        literal = "[" + ",".join(map(str, q)) + "]"
        conn.execute(f"SET LOCAL hnsw.ef_search = {ef_search}")
        aprox = {r[0] for r in conn.execute(
            "SELECT id FROM bench ORDER BY v <=> %s::vector LIMIT %s", (literal, k))}
        conn.execute("SET LOCAL enable_indexscan = off")
        exacto = {r[0] for r in conn.execute(
            "SELECT id FROM bench ORDER BY v <=> %s::vector LIMIT %s", (literal, k))}
        conn.execute("RESET enable_indexscan")
        total += len(aprox & exacto)
    return total / (len(consultas) * k)

def test_recall_por_defecto_no_regresiona(db):
    conn, vectores = db
    assert recall_medido(conn, vectores, ef_search=100) >= UMBRAL_RECALL

@pytest.mark.parametrize("ef,minimo", [(40, 0.80), (100, 0.95), (200, 0.98)])
def test_curva_de_recall(db, ef, minimo):
    conn, vectores = db
    assert recall_medido(conn, vectores, ef_search=ef) >= minimo
```

La prueba parametrizada es la más valiosa de las dos: documenta la curva completa entre `ef_search` y recall para tu configuración, así que cuando alguien proponga bajar el parámetro para ganar latencia, la conversación deja de ser una opinión y pasa a ser un número.

Ejecútalo en cada pull request que toque la capa de recuperación. Con 20.000 filas y 384 dimensiones el conjunto entero corre en menos de un minuto en un runner normal, que es exactamente el presupuesto que hace que la gente no lo desactive.

```
name: recall
on: [pull_request]
jobs:
  recall:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: pgvector/pgvector:pg17
        env:
          POSTGRES_PASSWORD: postgres
        options: >-
          --health-cmd pg_isready --health-interval 5s --health-retries 10
        ports: ['5432:5432']
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.13'
```

```
      - run: pip install -r requirements-dev.txt
      - run: pytest tests/test_recall.py -q
```

18. Cuándo pgvector deja de ser la respuesta

Una guía que sólo defiende su herramienta no sirve de mucho. pgvector es excelente hasta cierto punto, y conviene saber dónde está ese punto antes de chocar con él.

El límite práctico no es de dimensiones ni de filas: es de memoria. Un despliegue de pgvector bien ajustado en un solo nodo se mueve con comodidad hasta unos 50 millones de vectores, y por debajo de unos 5 millones consigue latencias de entre 5 y 50 ms sin esfuerzo. A partir de 5 o 10 millones, el rendimiento empieza a depender por completo de que el grafo HNSW quepa en RAM; con 50 millones de vectores de 768 dimensiones estás hablando de más de 150 GB sólo de índice, y ahí el coste de la máquina se convierte en el argumento principal.

Antes de irte, agota las palancas que ya conoces, en este orden por relación coste/beneficio:

1. `halfvec`: divide el índice por dos con una pérdida de recall casi imperceptible.
2. Reducción de dimensionalidad en el propio modelo, si lo soporta: 3.072 a 1.024 dimensiones es un tercio del índice.
3. Cuantización binaria con reranking: divide el índice por 32 a cambio de una segunda fase de lectura.
4. Particionado por inquilino o por fecha: nunca buscas sobre el conjunto entero, sólo sobre la partición relevante.

Si después de eso sigues sin caber, hay dos caminos que no obligan a salir de PostgreSQL. **pgvectorscale** añade StreamingDiskANN, un índice escrito en Rust y diseñado para funcionar bien cuando el conjunto de vectores es mayor que la memoria disponible, trabajando desde disco en lugar de exigir que todo quepa; sus benchmarks sobre 50 millones de vectores de 768 dimensiones reportan latencias p95 muy inferiores a las de servicios gestionados equivalentes con recall del 99 %. **VectorChord** y **Lantern** ocupan un espacio parecido con enfoques distintos. Todos mantienen tus datos, tus joins y tus transacciones donde están, que suele ser la razón por la que elegiste PostgreSQL en primer lugar.

Salir a una base de datos vectorial dedicada tiene sentido cuando se cumplen varias de estas condiciones a la vez, no una sola:

- Cientos de millones de vectores o más, con crecimiento sostenido.
- La búsqueda es el producto, no una función del producto.
- Necesitas filtrado por metadatos de alta cardinalidad con garantías de exactitud que el ANN filtrado no te da.
- Tienes un equipo que puede operar un sistema distribuido más, con su replicación, sus copias de seguridad y sus incidentes.

Ese último punto es el que más se subestima. Añadir un almacén separado significa que tus documentos y tus vectores pueden desincronizarse, y ahora tienes un problema de consistencia distribuida que antes resolvía una transacción. Muchos equipos que migraron por rendimiento acabaron manteniendo dos sistemas para un volumen que un nodo con 128 GB de RAM habría servido perfectamente.

La versión corta: quédate en PostgreSQL mientras la aritmética de la sección 11 dé un número que puedas pagar. Cuando deje de darlo, prueba primero las extensiones que se quedan dentro.

19. Caso práctico: de recall 0,71 y p95 de 240 ms a recall 0,96 y p95 de 26 ms

El sistema: un buscador interno de documentación técnica para una empresa con 40 clientes empresariales. 12 millones de fragmentos, 1.536 dimensiones, filtro obligatorio por `tenant_id` y filtro opcional por producto. Todo en una instancia con 64 GB de RAM. La queja era doble y contradictoria: la búsqueda va lenta y además no encuentra cosas.

Punto de partida. Columna `vector(1536)`, índice HNSW con parámetros por defecto, `ef_search` en 40, un `WHERE tenant_id = $1 AND producto = $2` delante del `ORDER BY`. Nadie había medido recall nunca.

Primer día: medir. El script de la sección 4 sobre 200 consultas reales dio recall@10 de 0,71. El p95 estaba en 240 ms. Y una tercera métrica que nadie miraba: el 18 % de las búsquedas devolvían menos de 10 resultados. Esa cifra sola ya explicaba la mitad de las quejas: no es que los resultados fueran malos, es que faltaban.

Diagnóstico del tamaño. La cuenta de la sección 11: 12.000.000 x 1.536 x 4 x 2 son unos 147 GB de índice sobre una máquina de 64 GB. El `EXPLAIN (ANALYZE, BUFFERS)` lo confirmó con miles de `shared read` por consulta. El índice llevaba meses sin caber en memoria y nadie lo sabía porque la latencia había subido despacio.

Cambio 1: halfvec. Migración de la columna a `halfvec(1536)` con el procedimiento de la sección 13. El índice bajó a unos 74 GB. Seguía sin caber, pero el recall no se movió: 0,71 antes, 0,71 después. Coste: dos horas de backfill sin llamadas a la API, porque `halfvec` es una conversión de tipo, no un recálculo.

```
ALTER TABLE fragmentos ADD COLUMN embedding_h halfvec(1536);
UPDATE fragmentos SET embedding_h = embedding::halfvec WHERE embedding_h IS NULL;
-- por lotes de 50.000 con la misma estructura de la seccion 13
```

Cambio 2: particionar por inquilino. Aquí estaba la ganancia grande, y no era de rendimiento bruto sino estructural. Con 40 inquilinos, particionar por hash en 16 particiones significa que cada búsqueda recorre un grafo de unos 750.000 vectores en lugar de uno de 12 millones. La partición activa cabe en memoria de sobra, y el filtro por `tenant_id` deja de ser un filtro que descarta resultados del ANN para convertirse en poda de particiones que el planificador aplica antes de tocar el índice.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id FROM fragmentos
WHERE tenant_id = '...'::uuid
ORDER BY embedding_h <=> $1 LIMIT 10;
-- Antes:  Index Scan sobre 12M, shared read = 4.812
-- Despues: Append -> Index Scan sobre 1 particion, shared read = 0
```

Recall subió a 0,89 y el p95 cayó a 34 ms. La subida de recall no vino de tocar parámetros: vino de que el filtro dejó de descartar candidatos que el índice ya había elegido.

Cambio 3: el filtro por producto. Seguía siendo un filtro real dentro de cada partición, y era el responsable del 18 % de búsquedas cortas. Con tres valores posibles y consultas que casi siempre filtran, tres índices parciales resolvieron el problema de raíz:

```
CREATE INDEX CONCURRENTLY ON fragmentos_p0 USING hnsw (embedding_h halfvec_cosine_ops)
  WHERE producto = 'plataforma';
-- idem para los otros dos productos y las otras 15 particiones,
-- generado con un DO block en lugar de a mano
```

Las búsquedas sin filtro de producto siguen usando el índice general. Las que filtran usan el parcial y obtienen los 10 vecinos reales dentro de su subconjunto. Las búsquedas cortas bajaron del 18 % al 0,4 %.

Cambio 4: ajustar ef_search con datos. Con el índice ya cabiendo en memoria, subir `ef_search` de 40 a 100 costó 8 ms y compró recall. La curva medida en preproducción fue: 40 da 0,89 y 26 ms; 100 da 0,96 y 34 ms; 200 da 0,975 y 58 ms. Se eligió 100 y se dejó documentada la curva en el test de la sección 17 para que nadie

la cambie por intuición.

Resultado. Recall@10 de 0,96, p95 de 26 ms en el percentil que importa (las consultas ya calientes), búsquedas cortas prácticamente eliminadas y, como efecto lateral, la instancia bajó de un uso de disco sostenido a prácticamente cero lecturas. Ninguno de los cuatro cambios fue un parámetro mágico: tres fueron decisiones de estructura de datos y el cuarto sólo se pudo tomar bien porque los tres anteriores habían quitado el ruido.

Preguntas frecuentes

¿Necesito una base de datos vectorial dedicada?

Depende del volumen y de tus requisitos. Hasta decenas de millones de vectores, pgvector bien configurado compite de tú a tú con los motores especializados y te ahorra un sistema entero que sincronizar, respaldar y monitorizar. Además puedes hacer JOIN con tus datos de negocio y filtrar transaccionalmente, que es exactamente lo que se vuelve incómodo cuando los vectores viven fuera. A escalas de cientos de millones, o si necesitas filtrado nativo sobre metadatos a muy alta velocidad, un motor dedicado empieza a justificar su coste operativo.

¿HNSW o IVFFlat?

HNSW salvo que tengas una razón concreta. IVFFlat construye más rápido y ocupa menos, lo cual importa si reconstruyes a menudo o vas muy justo de memoria, pero necesita reentrenarse cuando la distribución de los datos cambia y su recall es peor a igualdad de latencia.

¿Puedo combinar búsqueda vectorial con búsqueda por palabras clave?

Sí, y en la mayoría de los RAG deberías. La búsqueda léxica de PostgreSQL con `tsvector` encuentra lo que el vector se pierde (nombres propios, códigos de error, referencias exactas) y el vector encuentra lo que la palabra clave se pierde (sinónimos, paráfrasis). El patrón habitual es ejecutar ambas y fusionar los rankings con Reciprocal Rank Fusion, sin necesidad de calibrar puntuaciones entre sistemas distintos.

¿Cambio de modelo de embeddings; puedo hacerlo sin downtime?

Sí, con el patrón expand-contract. Añade una segunda columna, rellénala por lotes en segundo plano, crea el índice nuevo, cambia la consulta detrás de un feature flag, verifica el recall en producción y sólo entonces elimina la columna y el índice antiguos. No intentes reutilizar la misma columna: las distancias entre modelos distintos no son comparables y durante el backfill tendrías resultados mezclando dos espacios vectoriales.

¿Qué valor de m debería usar?

Empieza en 16. Súbelo a 32 sólo si has medido que el recall se queda corto incluso con un `ef_search` alto, y asume que el índice crecerá y el build tardará más. Cambiar `m` exige reconstruir; cambiar `ef_search` no. Agota siempre el mando gratuito antes de tocar el caro.

¿Cuánta RAM necesito exactamente?

Usa `N x D x 4 x 2` bytes para `vector`, la mitad para `halfvec`, y comprueba con `pg_relation_size` en cuanto tengas datos reales. La regla operativa es que el índice quepa en `shared_buffers` más la caché del sistema operativo. La sección 11 tiene el detalle.

¿Puedo usar pgvector con una réplica de lectura?

Sí, y es una buena idea: la búsqueda vectorial es intensiva en CPU y memoria, y aislarla de las escrituras evita que compitan. Dos avisos. El primero: la réplica necesita la misma RAM que la primaria, porque tiene que cachear el mismo índice. El segundo: `hnsw.ef_search` se configura por sesión, así que asegúrate de que tu pool apunta a la réplica con la misma configuración; una diferencia de `ef_search` entre nodos produce resultados distintos para la misma consulta.

¿Cómo hago copias de seguridad de una tabla con índices HNSW?

`pg_dump` no exporta el contenido del índice, sólo su definición, así que la restauración lo reconstruye desde cero. Para 12 millones de vectores eso puede ser una hora larga incluso con trabajadores en paralelo, y conviene tenerlo en el cálculo del RTO. Las copias físicas (`pg_basebackup`, snapshots de volumen) sí incluyen el índice y restauran mucho más rápido, a cambio de ocupar bastante más.

¿Merece la pena un reranker con cross-encoder después del ANN?

Casi siempre sí, y es la mejora de calidad más barata que queda cuando ya has ajustado el índice. Recupera 50 candidatos con pgvector, reordénalos con un cross-encoder y devuelve 10. El cross-encoder ve la consulta y el documento juntos, así que juzga mucho mejor que una distancia entre embeddings calculados por separado. El coste son unas decenas de milisegundos, que casi siempre son insignificantes al lado de la llamada al modelo generativo que viene después.

¿Por qué mi consulta empeoró después de un despliegue sin cambios en la base de datos?

El sospechoso habitual es un reinicio: la caché se vació y las primeras miles de consultas leen de disco. Si el efecto persiste más de unos minutos, mira si cambió el tamaño de la instancia, si alguien tocó `ef_search` en el código, o si el índice cruzó el umbral de caber en memoria mientras la tabla crecía. Las tres se distinguen en un minuto con `EXPLAIN (ANALYZE, BUFFERS)`.

Glosario

- **ANN** (approximate nearest neighbour): búsqueda de vecinos más cercanos que acepta perder algunos resultados a cambio de ser órdenes de magnitud más rápida que la exacta.
- **Recall@K**: proporción de los K vecinos verdaderos que tu búsqueda aproximada encontró de verdad. Es la métrica de calidad, y hay que medirla, no suponerla.
- **HNSW**: grafo jerárquico navegable de mundo pequeño. El índice por defecto recomendado en pgvector: construcción lenta, consulta rápida, sin fase de entrenamiento.
- **IVFFlat**: índice basado en listas invertidas sobre centroides. Construcción rápida y menor consumo de memoria, pero requiere datos representativos al crearlo.
- **m**: número máximo de conexiones por nodo del grafo HNSW. Sube el recall y el tamaño del índice.
- **ef_construction**: amplitud de la lista de candidatos al construir el grafo. Más calidad, más tiempo de construcción.
- **ef_search**: amplitud de la lista de candidatos al consultar. La palanca de recall frente a latencia en tiempo de ejecución.
- **halfvec**: vector en coma flotante de 16 bits. La mitad de espacio con una pérdida de recall casi siempre despreciable.
- **Cuantización binaria**: reducir cada dimensión a un bit con `binary_quantize()`. Índice 32 veces menor, exige reranking con los vectores originales.
- **Sobrefiltrado**: el índice ANN elige candidatos ignorando tu `WHERE`, y el filtro los descarta después. Se manifiesta como consultas que devuelven menos de K resultados.
- **Iterative scans**: mecanismo de pgvector 0.8 que amplía la búsqueda en el índice hasta reunir suficientes

resultados que pasen el filtro.

- **RRF** (reciprocal rank fusion): forma de combinar varios rankings sumando `1/(k + posición)`. Simple, robusta y difícil de mejorar.

pgvector in Production: HNSW, Filtered Search, Quantization, and Recall You Actually Measure

From a 5,000-row prototype to millions of vectors without surprises: column types, HNSW internals, real recall measurement, filters and iterative scans, quantization, memory, hybrid search, multi-tenancy and long-term operation.

Type: Guide | Difficulty: Intermediate | Category: AI
Reading time: ~42 min | Expanded edition: 21 August 2026
English version

When vector search stops scaling

The prototype always works. You load 5,000 text chunks into a PostgreSQL table with a `vector(1536)` column, run `ORDER BY embedding <=> $1 LIMIT 5`, and the answer comes back in twelve milliseconds. You do not even need an index: scanning 5,000 rows is trivial for any machine.

Six months later there are four million chunks, the same query takes six seconds, someone creates an HNSW index with default parameters, and latency drops back to 20 ms. Problem solved. Until product adds a language filter and the search starts returning two results where it used to return ten. No exception, no timeout, no alert fires. The RAG system simply answers worse, and it takes you weeks to figure out why.

That gap between *works* and *works well* is what this guide is about: choosing the column type before your disk reminds you, understanding what pgvector's two indexes actually do, measuring recall instead of assuming it, and above all learning why mixing filters with approximate search is the number one cause of silently wrong results.

Everything below assumes pgvector 0.8 or later on PostgreSQL 16+. Several key pieces (iterative scans, the improved cost estimation) do not exist in earlier versions, so step one is checking what you have.

```
SELECT extname, extversion FROM pg_extension WHERE extname = 'vector';
-- If you see 0.7.x or older, upgrade before going further:
ALTER EXTENSION vector UPDATE;
```

1. The column type decides your bill before the index does

pgvector does not offer a single data type, and the difference between them is not cosmetic. A `vector` stores every dimension as a 4-byte float plus 8 bytes of header. A `halfvec` uses 2 bytes per dimension. A `bit` uses one bit. At 1536 dimensions the arithmetic looks like this:

```
SELECT pg_column_size('[1,2,3]'::vector)   AS vector_3d,    -- 20 bytes  (4*3 + 8)
       pg_column_size('[1,2,3]'::halfvec)  AS halfvec_3d,   -- 14 bytes  (2*3 + 8)
       pg_column_size('101'::bit(3))       AS bit_3d;       -- 9 bytes

-- At 1536 dimensions and 4 million rows, for the column alone:
--   vector(1536)   -> 6,152 B/row  ->  ~23 GB
--   halfvec(1536)  -> 3,080 B/row  ->  ~11 GB
--   bit(1536)      ->   200 B/row  ->  ~0.7 GB
```

There is one detail almost nobody accounts for, and you pay it on every query: PostgreSQL moves any value out of line once the tuple grows past roughly 2 KB. A `vector(1536)` always ends up in TOAST, so every time you read the real embedding (when reranking candidates, for example) you pay an extra read against the TOAST table. A `halfvec(768)` takes 1,544 bytes and stays inline. This is not academic: when you rerank hundreds of candidates, that extra fetch shows up in your p99.

The practical rule: store the highest-precision vector your model produces, but **index** a reduced version. The index is what has to fit in memory, not the column.

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE documents (
    id          bigserial   PRIMARY KEY,
    tenant_id   bigint      NOT NULL,
    title       text        NOT NULL,
    body        text        NOT NULL,
    lang        text        NOT NULL DEFAULT 'en',
    published   boolean     NOT NULL DEFAULT true,
    deleted_at  timestamptz,
    embedding   vector(1536) NOT NULL,
    created_at  timestamptz NOT NULL DEFAULT now()
);
```

A limit worth knowing before you pick a model: the `vector` type can be indexed up to 2,000 dimensions, `halfvec` up to 4,000, and `bit` up to 64,000. If your model returns 3,072 dimensions you cannot build an HNSW index directly on the `vector` column; you will have to index the expression `embedding::halfvec(3072)`. Many modern models also support Matryoshka representations, so you can truncate to 512 or 1024 dimensions and lose very little. Verify that on your own data before assuming it.

2. HNSW and IVFFlat: what each one actually does

Both indexes solve the same problem (avoid comparing your query against every row) with opposite philosophies.

IVFFlat partitions the space into `lists` cells using k-means, assigns each vector to its nearest centroid, and at query time only inspects the `ivfflat.probes` most promising cells. It builds fast and stays small, but it has a serious operational flaw: the centroids are computed once, from whatever data existed at that moment. If your embedding distribution shifts (new languages, a new domain, a different model), the index degrades and has to be rebuilt. It also needs data to train on: creating an IVFFlat index on an empty table produces a useless index.

HNSW builds a multi-layer navigable graph. Upper layers are long-distance shortcuts; lower layers are dense neighbourhoods. A search enters at the top, descends, and explores. It is slower to build and noticeably larger, but it delivers better recall at the same latency, accepts incremental inserts without falling off a cliff, and needs no training phase. In 2026, for most production workloads, HNSW is the default and IVFFlat is the justified exception (very large datasets, scarce RAM, tolerance for periodic rebuilds).

The three HNSW parameters that matter:

- `m` (default 16): connections per node. More connections mean better recall and more memory. Raising it to

32 or 48 makes sense in high-dimensional spaces where recall resists; going below 16 almost never pays off.

- `ef_construction` (default 64): how many candidates are examined when inserting each node. It affects graph quality and build time, not index size. For 1536-dimensional embeddings, 128–200 is a far more sensible starting point than the default.
- `hnsw.ef_search` (default 40): how many candidates are explored *per query*. It is the only one of the three you can change without rebuilding anything, and it is your main dial for trading recall against latency.

That default of 40 is the source of half the posts claiming "pgvector is slow" or "pgvector has bad recall". It is not pgvector: nobody touched the dial.

3. Building the index without losing your afternoon

Building an HNSW index over millions of rows can take three hours or twenty minutes, and the difference comes down to two settings almost nobody configures.

The first is `maintenance_work_mem`. pgvector builds the graph in memory if it fits; if it does not, it falls back to an on-disk mode that is an order of magnitude slower. Overflowing memory does not raise an error, it just logs a notice while the build crawls. The second is parallelism: pgvector has supported parallel builds since 0.6, controlled by `max_parallel_maintenance_workers`.

```
-- Session-scoped only; do not edit postgresql.conf for a one-off build.
SET maintenance_work_mem = '8GB';
SET max_parallel_maintenance_workers = 7;   -- 7 workers + the leader = 8

CREATE INDEX documents_embedding_hnsw
    ON documents
    USING hnsw (embedding vector_cosine_ops)
    WITH (m = 16, ef_construction = 128);
```

There is an important trap here: `CREATE INDEX CONCURRENTLY` **does not use parallel workers**. That is a PostgreSQL limitation, not a pgvector one. You have to choose between building fast while blocking writes to the table, or building online and paying for it in wall-clock time. On a four-million-vector table the difference can be twenty minutes with a lock versus three hours without one. If you have a maintenance window, use it; if you do not, fire the `CONCURRENTLY` build and be patient.

```
-- Watch progress from another session
SELECT phase,
       round(100.0 * blocks_done  / NULLIF(blocks_total, 0), 1)  AS pct_blocks,
       round(100.0 * tuples_done  / NULLIF(tuples_total, 0), 1)  AS pct_tuples
FROM pg_stat_progress_create_index;
```

Two operational warnings. If you run PostgreSQL in Docker or Kubernetes, parallel builds use shared memory: `--shm-size` (or an `emptyDir` volume with `medium: Memory`) must be at least as large as your `maintenance_work_mem`, or the build fails halfway through. And if you are loading millions of rows from scratch, load first and index afterwards: inserting four million vectors into an existing HNSW index is dramatically slower than building the graph in one pass.

4. Recall: measure it, do not believe it

An ANN index is an approximation. The relevant question is not "is it fast?" but "how many of the true ten neighbours am I actually getting back?". That question has a numeric answer you can compute in twenty minutes, so there is no excuse for guessing.

The method is direct: for a set of sample queries, compute the exact top-K with the index disabled, compute the top-K the index returns at various `ef_search` values, and compare the sets.

```
import statistics

import psycopg
from pgvector.psycopg import register_vector

DSN = "postgresql://app@localhost/rag"
K = 10        # the top-K your API actually serves
SAMPLE = 200  # probe queries

conn = psycopg.connect(DSN)
register_vector(conn)

# Use real embeddings as queries: synthetic vectors lie to you.
with conn.cursor() as cur:
    cur.execute(
        "SELECT embedding FROM documents TABLESAMPLE SYSTEM (1) LIMIT %s",
        (SAMPLE,),
    )
    queries = [row[0] for row in cur.fetchall()]


def top_k(cur, q):
    cur.execute(
        "SELECT id FROM documents ORDER BY embedding <=> %s LIMIT %s", (q, K)
    )
    return {row[0] for row in cur.fetchall()}


# Ground truth: computed once, because this is the expensive part.
ground_truth = []
for q in queries:
    with conn.transaction(), conn.cursor() as cur:
        cur.execute("SET LOCAL enable_indexscan = off")
        ground_truth.append(top_k(cur, q))

for ef in (20, 40, 80, 160, 320):
    scores = []
    for q, exact in zip(queries, ground_truth):
        with conn.transaction(), conn.cursor() as cur:
            # SET takes no bind parameters: interpolate a validated int.
            cur.execute(f"SET LOCAL hnsw.ef_search = {int(ef)}")
            approx = top_k(cur, q)
        scores.append(len(exact & approx) / K)
    print(f"ef_search={ef:<4} recall@{K} = {statistics.mean(scores):.3f}")
```

A typical run over 1536-dimensional embeddings and a few million rows looks like this:

```
ef_search=20   recall@10 = 0.842
ef_search=40   recall@10 = 0.913
ef_search=80   recall@10 = 0.961
ef_search=160  recall@10 = 0.984
ef_search=320  recall@10 = 0.993
```

Your exact numbers will differ, but the shape of the curve is almost always the same: it climbs fast, then flattens. That is where the product decision lives. For a support chatbot, 0.96 at 80 candidates is probably more than enough. For legal or medical search you may want 0.99 and be willing to pay the latency. What matters is that it is a deliberate decision rather than a library default.

Put this measurement in CI. Swapping embedding models, migrating data, or tuning `m` can move recall without moving latency or any other metric you are already watching.

```
-- Per-query tuning, not global: each endpoint can pick its own trade-off.
BEGIN;
SET LOCAL hnsw.ef_search = 100;
SELECT id, title FROM documents ORDER BY embedding <=> :q LIMIT 10;
COMMIT;
```

5. The silent failure: filters plus approximate search

This is the problem that does the most damage and gets caught the least. Picture the most natural query in the world:

```
SET hnsw.ef_search = 40;   -- the default

SELECT id, title
FROM documents
WHERE lang = 'en' AND published
ORDER BY embedding <=> :q
LIMIT 10;
```

PostgreSQL walks the HNSW index asking for candidates and applies the `WHERE` clause *afterwards*. With `ef_search = 40` the index hands back roughly 40 candidates; if only 10% of your documents are English and published, four will survive the filter. You asked for ten results and got four. The query is correct, fast, and quietly under-delivers.

pgvector 0.8 introduced iterative scans for exactly this: instead of fetching one fixed batch and filtering, the index keeps pulling candidates until it has enough rows that pass the filter.

```
-- 'strict_order': results come back strictly ordered by distance.
SET hnsw.iterative_scan = 'strict_order';

-- Work ceiling so a very selective filter cannot sweep the whole table.
SET hnsw.max_scan_tuples = 20000;      -- default 20000
SET hnsw.scan_mem_multiplier = 2;      -- default 1; raise it if the scan runs short

SELECT id, title
FROM documents
WHERE lang = 'en' AND published
ORDER BY embedding <=> :q
LIMIT 10;
```

There is a second mode, `relaxed_order`, which is faster because it does not guarantee that rows come out perfectly ordered by distance. If exact ordering matters (and in a RAG pipeline it does, because you usually truncate by position), wrap it and re-sort outside:

```
SET hnsw.iterative_scan = 'relaxed_order';

WITH candidates AS MATERIALIZED (
    SELECT id, title, embedding <=> :q AS distance
    FROM documents
    WHERE lang = 'en' AND published
    ORDER BY embedding <=> :q
    LIMIT 10
)
SELECT id, title, distance
FROM candidates
ORDER BY distance;
```

The `MATERIALIZED` keyword is not decorative: it forces PostgreSQL to execute the subquery as written instead of flattening it and losing the re-sort.

The IVFFlat equivalent is `ivfflat.iterative_scan = 'relaxed_order'` together with `ivfflat.max_probes`; IVFFlat offers no `strict_order` mode.

When the filter is fixed, do not pay for it at query time

Iterative scans fix correctness, but they still cost work. If a filter appears in 100% of your queries, bake it into the index:

```
CREATE INDEX CONCURRENTLY documents_embedding_live_hnsw
    ON documents
    USING hnsw (embedding vector_cosine_ops)
    WITH (m = 16, ef_construction = 128)
    WHERE published AND deleted_at IS NULL;
```

A partial index only contains live rows, so the graph is smaller, fits in RAM more comfortably, and does not waste candidates on documents you are going to discard anyway. For the planner to use it, your query's `WHERE` clause must imply the index predicate.

And if your dominant filter is the tenant in a multi-tenant application, the right answer is not one partial index per customer (that does not scale to a thousand tenants) but partitioning the table:

```
CREATE TABLE documents (
    id        bigserial,
    tenant_id bigint       NOT NULL,
    embedding vector(1536) NOT NULL,
    -- ...
    PRIMARY KEY (id, tenant_id)
) PARTITION BY HASH (tenant_id);

CREATE TABLE documents_p0 PARTITION OF documents FOR VALUES WITH (MODULUS 8, REMAINDER 0);
-- ... p1 through p7

-- One HNSW index per partition: smaller graphs, parallelizable builds.
CREATE INDEX ON documents_p0 USING hnsw (embedding vector_cosine_ops);
```

The condition for this to pay off is that **every** query includes `tenant_id` so the planner can prune partitions. Verify it with `EXPLAIN`: if you see all eight partitions in the plan, you are not pruning and you just multiplied your work by eight.

6. Quantization: cutting the index in half, or by 32

An HNSW index performs when it fits in memory. If the graph takes 40 GB and your instance has 32 GB of RAM, every query ends up reading from disk and latency becomes erratic: fast when the cache cooperates, terrible when it does not. Quantization attacks that at the root by shrinking what gets indexed.

halfvec: half the size, practically the same recall

Moving from float32 to float16 halves the index, and with normalized embeddings from current models the recall loss is usually a fraction of a point. It is the best benefit-to-risk change in this entire guide. Do not convert the column: index the expression.

```
CREATE INDEX CONCURRENTLY documents_embedding_half_hnsw
    ON documents
    USING hnsw ((embedding::halfvec(1536)) halfvec_cosine_ops)
    WITH (m = 16, ef_construction = 128);

-- The query MUST repeat the exact expression, or the index is ignored.
SELECT id, title
```

```
FROM documents
ORDER BY embedding::halfvec(1536) <=> (:q)::halfvec(1536)
LIMIT 10;
```

This is one of those things that breaks in silence: if your query says `embedding <=> :q` instead of the cast expression, PostgreSQL cannot use the expression index and falls back to a sequential scan. Fast in development, catastrophic in production. Always confirm with `EXPLAIN`.

Binary quantization plus reranking: the two-phase pattern

Binary quantization turns each dimension into a single bit based on its sign: positive becomes 1, everything else 0. It sounds brutal, and it is: a 1536-dimensional vector goes from 6 KB to 192 bytes, a 32x reduction. On its own it loses too much precision to serve final results, but it is excellent as a *first-stage filter*. The recipe is to retrieve many candidates with the binary index and rerank them with the full vectors.

```
CREATE INDEX CONCURRENTLY documents_embedding_bit_hnsw
    ON documents
    USING hnsw ((binary_quantize(embedding)::bit(1536)) bit_hamming_ops);

-- Phase 1: 200 candidates by Hamming distance over the binary index.
-- Phase 2: rerank those 200 with the real cosine distance.
SELECT id, title, distance
FROM (
    SELECT id, title, embedding <=> (:q)::vector(1536) AS distance
    FROM documents
    ORDER BY binary_quantize(embedding)::bit(1536)
             <~> binary_quantize((:q)::vector(1536))::bit(1536)
    LIMIT 200
) AS candidates
ORDER BY distance
LIMIT 10;
```

The knob to calibrate is the phase-1 `LIMIT`, the oversampling factor. For 10 final results, starting at 200 candidates (20x) is reasonable; measure recall with the same script from section 4 and adjust. If 100 already hits your target, save the work.

One honest caveat: binary quantization works well with models trained or evaluated to survive it (Cohere Embed v3, mxbai-embed-large, Jina v3, among others) and can degrade badly with models that are not. Never adopt it without measuring recall on your own data and your own queries.

Trade-off summary at 1536 dimensions:

- **vector** — the baseline. Maximum precision, maximum size, large index.
- **halfvec** — half the index, near-zero recall loss. Make it your default unless you measure otherwise.
- **bit + reranking** — a 32x smaller index in exchange for a second phase and a dependency on your model. Reach for this when the index no longer fits in RAM.

7. Queries that actually use the index

A vector index only kicks in under fairly strict conditions, and stepping outside them produces no warning at all.

- **The operator must match the index operator class.** An index built with `vector_cosine_ops` only serves `<=>`. Query with `<->` (L2) and you get a sequential scan. This is the most common mistake and the easiest one to make when copying code from a tutorial.
- **You need `ORDER BY` with `LIMIT`.** Without a limit, PostgreSQL has to sort the whole table and the index buys you nothing.
- **No thresholds in the `WHERE` clause.** A `WHERE embedding <=> :q < 0.4` is not a nearest-neighbour

query. Retrieve first, filter second.

```
-- BAD: not a k-NN search, does not use the ANN index.
SELECT id FROM documents WHERE embedding <=> :q < 0.4;

-- GOOD: retrieve k, then apply the threshold.
SELECT id, distance
FROM (
    SELECT id, embedding <=> :q AS distance
    FROM documents
    ORDER BY embedding <=> :q
    LIMIT 100
) t
WHERE distance < 0.4;
```

And always check the plan. An `Index Scan using documents_embedding_hnsw` is what you want; a `Seq Scan` followed by `Sort` means something does not line up.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id FROM documents ORDER BY embedding <=> :q LIMIT 10;
```

A note on distance functions: if your embeddings are normalized to unit length (OpenAI, Cohere and most sentence-transformers models produce them that way), cosine similarity and inner product rank identically, and `vector_ip_ops` with the `<#>` operator is slightly cheaper to compute. Note that `<#>` returns the *negative* inner product so that "smaller is better" still holds. If you are not sure your vectors are normalized, stay with cosine: it is the safe choice.

8. Operating the index over time

Day one is fine. Things get interesting around month six.

The graph degrades with churn. HNSW accepts inserts and deletes without a rebuild, but under MVCC every embedding update is really a delete plus an insert, and dead nodes leave holes in the graph. On high-churn tables recall drifts down slowly while the index grows. The fix is periodic and needs no downtime:

```
REINDEX INDEX CONCURRENTLY documents_embedding_hnsw;
```

Watch the size against your RAM. There is no reliable formula to predict HNSW index size: it depends on `m`, on dimensionality, and on how your data is distributed. Build it over a 5% sample, measure, extrapolate.

```
SELECT indexrelname,
       pg_size_pretty(pg_relation_size(indexrelid)) AS index_size,
       idx_scan
FROM pg_stat_user_indexes
WHERE relname = 'documents'
ORDER BY pg_relation_size(indexrelid) DESC;
```

Warm the cache after a restart. The first minute after a failover or a database deploy is the worst one: the index is cold and every query hits disk. `pg_prewarm` turns that problem into a startup step.

```
CREATE EXTENSION IF NOT EXISTS pg_prewarm;
SELECT pg_prewarm('documents_embedding_hnsw');
```

Put a ceiling on each query. With iterative scans enabled, a query with a pathologically selective filter can do far more work than you expect. A local `statement_timeout` is your safety net.

Here is a search function that pulls all of the above together:

```
from dataclasses import dataclass

SEARCH_SQL = """
WITH candidates AS MATERIALIZED (
    SELECT id, title, body, embedding <=> %(q)s AS distance
    FROM documents
    WHERE tenant_id = %(tenant)s
      AND lang = %(lang)s
      AND published
      AND deleted_at IS NULL
    ORDER BY embedding <=> %(q)s
    LIMIT %(k)s
)
SELECT id, title, body, distance
FROM candidates
WHERE distance < %(max_distance)s
ORDER BY distance
"""


@dataclass(frozen=True)
class Hit:
    id: int
    title: str
    body: str
    distance: float


async def search(conn, q, *, tenant, lang, k=10, ef_search=100, max_distance=0.45):
    """Filtered, ordered and time-bounded vector search."""
    async with conn.transaction():
        # SET LOCAL: these settings die with the transaction and never leak
        # into the next request that reuses this pooled connection.
        await conn.execute(f"SET LOCAL hnsw.ef_search = {int(ef_search)}")
        await conn.execute("SET LOCAL hnsw.iterative_scan = 'relaxed_order'")
        await conn.execute("SET LOCAL statement_timeout = '2s'")

        cur = await conn.execute(
            SEARCH_SQL,
            {
                "q": q,
                "tenant": tenant,
                "lang": lang,
                "k": k,
                "max_distance": max_distance,
            },
        )
        return [Hit(*row) for row in await cur.fetchall()]
```

The `SET LOCAL` detail deserves its own paragraph. If you use a plain `SET` on a pooled connection, the setting outlives your request and affects the next one, which may come from a different endpoint with different needs. With PgBouncer in transaction mode it gets even more entertaining to debug, because connection assignment changes between runs. `SET LOCAL` inside an explicit transaction eliminates the entire class of problem.

9. Eight recurring mistakes

1. **Leaving `hnsw.ef_search` at 40 and concluding recall is bad.** Measure the curve before forming an opinion.
2. **Filtering with `WHERE` without enabling iterative scans.** You return fewer results than requested and nobody notices.
3. **Using an operator that does not match the index operator class.** `<->` against a `vector_cosine_ops` index is a sequential scan in disguise.

4. **Creating an expression index and not repeating the expression in the query.** The index exists, occupies disk, and is never used.
5. **Building the index before loading the data.** Much slower, and with IVFFlat outright useless because there is nothing to train the centroids on.
6. **Forgetting** `maintenance_work_mem`**.** The build drops into on-disk mode and takes hours without telling you why.
7. **Expecting parallelism from** `CREATE INDEX CONCURRENTLY`**.** There is none; plan your timeline accordingly.
8. **Adopting binary quantization without reranking or measurement.** It is a first-stage filter, not a replacement for the full vector.

10. Production checklist

- pgvector 0.8+ installed and verified with `SELECT extversion FROM pg_extension WHERE extname = 'vector'`.
- Recall@K measured on real queries and documented next to the chosen `ef_search` value.
- The recall measurement runs in CI and fails the build if it drops below the agreed threshold.
- `hnsw.iterative_scan` configured on every query that combines `WHERE` with distance ordering.
- Permanent filters moved into partial indexes; tenant filtering moved into partitions.
- Per-query tuning done with `SET LOCAL` inside a transaction, never with a global `SET`.
- `statement_timeout` applied on the search path.
- Index size monitored against available RAM, with an alert above 70%.
- `pg_prewarm` wired into startup or the post-failover procedure.
- `REINDEX CONCURRENTLY` scheduled if the table has high churn.
- `EXPLAIN` reviewed for every new vector query before it ships.

11. The memory budget: size your RAM before you pick the machine

The question that most often arrives too late is a simple one: how much RAM does this need? It arrives late because the prototype fit anywhere and nobody did the arithmetic. The arithmetic exists, and it is napkin-sized.

An HNSW index stores, for every row, the full vector plus the graph links. A conservative estimate that holds up well in practice is `N x D x 4 bytes x 2`: the first factor is the 32-bit float vector, the second covers graph and page overhead. For five million rows of 1,536 dimensions that is roughly 60 GB. That is not a typo, and it is not the size of the table: it is the index alone.

That number changes your architecture before any parameter does. With `halfvec` you drop to 2 bytes per dimension and the index falls to around 30 GB. With binary quantization and two-phase reranking, the bit index takes about 2 GB and the original vectors live in the table, read only for the handful of finalists.

Measure instead of estimating as soon as you have real data, even a sample:

```
SELECT
  pg_size_pretty(pg_relation_size('documents'))              AS table_size,
  pg_size_pretty(pg_relation_size('documents_embedding_idx')) AS hnsw_index,
  pg_size_pretty(pg_total_relation_size('documents'))        AS total,
  (SELECT count(*) FROM documents)                           AS rows;
```

Extrapolate linearly: if 500,000 rows give you 6 GB of index, ten million will give you roughly 120 GB. Linearity holds well because the graph has degree bounded by `m`.

With the size in hand, the operating rule is short: the index must fit in memory. Not necessarily in

`shared_buffers`, but in the sum of `shared_buffers` and the operating system page cache. When it does not fit, every graph hop that misses cache becomes a random disk read, and an HNSW search makes dozens or hundreds of hops. That is where an 8 ms query becomes a 400 ms query without a single parameter having changed.

```
# postgresql.conf on a 128 GB machine dedicated to vector search
shared_buffers = 32GB          # 25% of RAM; the OS manages the rest
effective_cache_size = 96GB    # planner hint, does not reserve memory
work_mem = 64MB                # per plan node; watch out with concurrency
maintenance_work_mem = 24GB    # index builds only, see section 3
max_parallel_maintenance_workers = 7
```

After a restart or a failover the cache is empty and the first queries pay full price. `pg_prewarm` turns that cold start into a deployment step instead of an alert:

```
CREATE EXTENSION IF NOT EXISTS pg_prewarm;
SELECT pg_prewarm('documents_embedding_idx');
```

Checking whether you are reading from disk is straightforward. Run the query with `EXPLAIN (ANALYZE, BUFFERS)` and look at `shared read`: if it is high and stays high across several runs, the index does not fit, and no amount of `hnsw.ef_search` will fix it. What fixes it is more RAM, `halfvec`, binary quantization, or partitioning.

12. Hybrid search: vectors alone will never find "ORD-2026-88131"

Embeddings are good at understanding intent and bad at remembering literal strings. If a user searches for an order ID, an internal function name, or an error code, semantic search returns documents that are topically similar and none that contain the exact string. It is the most common failure in internal search, and raising `hnsw.ef_search` does not fix it.

The answer is not choosing between semantic and lexical, but fusing the two lists. PostgreSQL already ships the lexical half with `tsvector`, so there is no new infrastructure to operate.

```
ALTER TABLE documents
  ADD COLUMN fts tsvector
  GENERATED ALWAYS AS (to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''))) STORED;

CREATE INDEX documents_fts_idx ON documents USING gin (fts);
```

To combine the two rankings, the method that almost always wins is also the most boring: Reciprocal Rank Fusion. It does not compare scores (they live on incomparable scales: a cosine distance and a `ts_rank` cannot be added), it compares positions. Each document gets `1 / (k + rank)` from each list and the contributions are summed. The constant `k`, traditionally 60, damps the gap between the top positions and stops a single overconfident retriever from dominating the result.

```
WITH semantic AS (
  SELECT id,
         ROW_NUMBER() OVER (ORDER BY embedding <=> $1) AS rank
  FROM documents
  ORDER BY embedding <=> $1
  LIMIT 50
),
lexical AS (
  SELECT id,
         ROW_NUMBER() OVER (ORDER BY ts_rank_cd(fts, q) DESC) AS rank
  FROM documents, websearch_to_tsquery('english', $2) AS q
```

```
  WHERE fts @@ q
  ORDER BY ts_rank_cd(fts, q) DESC
  LIMIT 50
)
SELECT d.id, d.title,
       COALESCE(1.0 / (60 + s.rank), 0) + COALESCE(1.0 / (60 + l.rank), 0) AS rrf
FROM documents d
LEFT JOIN semantic s ON s.id = d.id
LEFT JOIN lexical  l ON l.id = d.id
WHERE s.id IS NOT NULL OR l.id IS NOT NULL
ORDER BY rrf DESC
LIMIT 10;
```

Two details matter more than they look. First: each branch asks for 50 candidates even though you return 10, because fusion needs depth to have anything to fuse; with lists of 10 the result looks too much like an intersection. Second: the `LIMIT` inside the semantic CTE is what allows the ANN index to be used, and if you remove it so as not to "lose results" you end up with a Seq Scan computing distances over the whole table.

If you want to weight one signal more heavily, multiply that branch's contribution rather than touching `k`. A factor of 1.5 on the lexical branch is enough to push exact codes to the top without wrecking the rest of the ranking:

```
RRF_K = 60
SEMANTIC_WEIGHT = 1.0
LEXICAL_WEIGHT = 1.5

def fuse(semantic: list[str], lexical: list[str], limit: int = 10):
    """Fuse two already-ordered id lists. Useful when one branch does not
    live in PostgreSQL (an external reranker, a search engine, and so on)."""
    scores: dict[str, float] = {}
    for weight, ranked in ((SEMANTIC_WEIGHT, semantic), (LEXICAL_WEIGHT, lexical)):
        for position, doc_id in enumerate(ranked, start=1):
            scores[doc_id] = scores.get(doc_id, 0.0) + weight / (RRF_K + position)
    return sorted(scores, key=scores.get, reverse=True)[:limit]
```

The improvement is measurable and usually large. On internal evaluation sets it is common to go from around 60% retrieval precision with vectors alone to over 80% with fusion, and the gain lands exactly where it hurts most: queries containing proper nouns, references, and codes.

13. Changing embedding models without downtime

Sooner or later you will want to change models: a better one ships, a cheaper one ships, or your provider deprecates the one you use. The problem is that vectors from two different models are not comparable. There is no conversion and no incremental value migration: everything has to be recomputed. And while you recompute, search has to keep working.

The pattern is the same expand-contract you would use to change a column type, with the difference that here the backfill costs API money.

```
-- Expand: the new column lives alongside the old one
ALTER TABLE documents ADD COLUMN embedding_v2 halfvec(3072);

-- No index yet: building it before the backfill means paying the
-- graph insertion cost row by row.
```

The backfill runs in batches, with retries, and without blocking writes. The key is to advance by primary key instead of using `OFFSET`, and to commit per batch so a failure halfway through does not force you to start over:

```
import time
import psycopg
from openai import OpenAI

BATCH_SIZE = 256
client = OpenAI()

def backfill(dsn: str) -> None:
    with psycopg.connect(dsn, autocommit=False) as conn:
        last_id = "00000000-0000-0000-0000-000000000000"
        while True:
            with conn.cursor() as cur:
                cur.execute(
                    """SELECT id, body FROM documents
                       WHERE id > %s AND embedding_v2 IS NULL
                       ORDER BY id LIMIT %s""",
                    (last_id, BATCH_SIZE),
                )
                rows = cur.fetchall()

            if not rows:
                break

            texts = [body for _, body in rows]
            response = client.embeddings.create(
                model="text-embedding-3-large", input=texts
            )
            vectors = [d.embedding for d in response.data]

            with conn.cursor() as cur:
                cur.executemany(
                    "UPDATE documents SET embedding_v2 = %s WHERE id = %s",
                    [(v, row[0]) for v, row in zip(vectors, rows)],
                )
            conn.commit()
            last_id = rows[-1][0]
            time.sleep(0.05)  # respect the provider rate limit
```

With the backfill done you build the index, now over complete data and therefore far faster than if it had been filled row by row:

```
SET maintenance_work_mem = '24GB';
SET max_parallel_maintenance_workers = 7;
CREATE INDEX CONCURRENTLY documents_embedding_v2_idx
  ON documents USING hnsw (embedding_v2 halfvec_cosine_ops)
  WITH (m = 16, ef_construction = 128);
```

Before you move traffic, compare. This is the step almost everyone skips and the one that stops you discovering in production that the new model is worse for your specific domain. Run your evaluation query set against both columns and look at recall and result overlap. A model with better numbers on a public benchmark can perform worse on your vocabulary.

The cutover itself is a feature flag, not a deployment. Keep both columns for a few days with whatever share of traffic you like on each, watch latency and quality metrics, and only then:

```
-- Contract: once the new model has been stable for days
DROP INDEX CONCURRENTLY documents_embedding_idx;
ALTER TABLE documents DROP COLUMN embedding;
ALTER TABLE documents RENAME COLUMN embedding_v2 TO embedding;
```

A warning about cost: recomputing embeddings for ten million chunks is neither free nor instant. Do the arithmetic with your provider's price per million tokens before approving the migration, and consider whether you really need 3,072 dimensions or whether the model supports dimensionality reduction in the API call itself,

which is the cheapest way to shrink both index and bill at once.

14. Writes: what updates do to your graph

Almost all the literature on HNSW assumes a static index. Your table is not: new documents arrive, edited documents get reindexed, expired ones get deleted. It is worth understanding what happens underneath, because the failure mode is slow and silent.

When you insert a row, pgvector walks the graph to find neighbours for it and links it in. That is a full search operation per insert, which is why loading a million rows with the index already in place is orders of magnitude slower than creating the index afterwards. Practical rule: for bulk loads, drop the index, load, and rebuild.

When you delete or update a row, PostgreSQL marks the old version dead and the index keeps its entry until `VACUUM` comes through. In the meantime those entries still occupy space in the graph and are still visited during search, only to be discarded when visibility is checked against the table. That is work paid for with no result: the search makes the same hops but returns fewer usable candidates, which shows up as recall drifting downward over weeks.

On a vector table with high churn it pays to be more aggressive than the defaults:

```
ALTER TABLE documents SET (
  autovacuum_vacuum_scale_factor = 0.02,   -- 2% instead of the default 20%
  autovacuum_vacuum_cost_limit   = 2000,   -- more I/O budget
  autovacuum_analyze_scale_factor = 0.05
);
```

Even so, `VACUUM` reclaims space but does not rebuild the graph. Links that pointed at removed nodes are repaired locally, and with enough churn graph quality degrades: paths become less direct and more visits are needed for the same recall. The only real cure is a rebuild:

```
REINDEX INDEX CONCURRENTLY documents_embedding_idx;
```

How often? There is no universal answer, but there is a reliable signal: measure recall periodically with the script from section 4 and rebuild when it falls below your threshold. As a rough guide, a table where 20% of rows are rewritten monthly tends to want a quarterly rebuild; a table that only grows almost never needs one.

Watch index growth against row growth as well. If the index grows faster than the rows, you have bloat:

```
SELECT
  pg_size_pretty(pg_relation_size('documents_embedding_idx')) AS index_size,
  n_live_tup, n_dead_tup,
  round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS pct_dead,
  last_autovacuum
FROM pg_stat_user_tables
WHERE relname = 'documents';
```

One last pattern that saves a lot of pain: do not rewrite the embedding if the text has not changed. Store a hash of the content and compare before calling the API and before touching the vector column. It is a one-line check that removes most of the writes in periodic reindexing pipelines.

```
ALTER TABLE documents ADD COLUMN body_hash text;

UPDATE documents SET embedding = $1, body_hash = $2
WHERE id = $3 AND body_hash IS DISTINCT FROM $2;
```

15. Observability: what to measure in a vector search system

Vector search fails in an awkward way: it keeps returning ten results, they look plausible, and they arrive in a few milliseconds. No exception, no 500, no alert. The results are simply worse. If your instrumentation only measures latency and error rate, you will not find out.

There are three layers worth measuring, and they are different from each other.

Latency per stage. A RAG query is not one query: it is generating the embedding, searching, optionally reranking, and then calling the model. Measuring the total hides which of the four degraded. In more than one investigation the culprit turned out to be the embeddings API call, not the database.

```
import time
from contextlib import contextmanager
from prometheus_client import Histogram, Gauge, Counter

LATENCY = Histogram(
    "search_stage_seconds", "Latency per stage", ["stage"],
    buckets=(0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5),
)
RECALL = Gauge("search_recall_at_10", "Recall@10 measured by the canary")
EMPTY = Counter("search_no_results_total", "Queries that returned nothing")

@contextmanager
def measure(stage: str):
    start = time.perf_counter()
    try:
        yield
    finally:
        LATENCY.labels(stage=stage).observe(time.perf_counter() - start)

def search(text: str, k: int = 10):
    with measure("embedding"):
        vector = build_embedding(text)
    with measure("ann"):
        rows = query_pgvector(vector, k)
    if not rows:
        EMPTY.inc()
    with measure("rerank"):
        return rerank(text, rows)
```

Recall in production. The recall you measured on your laptop describes the index as it was that day. Build a canary: a scheduled job that every hour takes a sample of real queries, runs both the approximate and the exact search (with `enable_indexscan = off`, or an unindexed `ORDER BY` over a sample) and publishes the overlap as a metric. It is cheap, it runs off the critical path, and it is the only way to catch gradual degradation before your users do.

```
def recall_canary(queries: list[str], k: int = 10) -> float:
    hits = 0
    for text in queries:
        vector = build_embedding(text)
        approximate = {r.id for r in query_pgvector(vector, k)}
        exact = {r.id for r in query_exact(vector, k)}
        hits += len(approximate & exact)
    value = hits / (len(queries) * k)
    RECALL.set(value)
    return value
```

What PostgreSQL tells you. `pg_stat_statements` gives you the real p95 per normalized query and, more usefully still, the ratio of blocks read to blocks found in cache. A jump in `shared_blks_read` on the search query is the unmistakable signature of an index that has stopped fitting in memory.

```
SELECT
  calls,
  round(mean_exec_time::numeric, 2)           AS mean_ms,
  round((total_exec_time / 1000)::numeric, 1) AS total_s,
  shared_blks_hit, shared_blks_read,
  round(100.0 * shared_blks_hit / NULLIF(shared_blks_hit + shared_blks_read, 0), 2) AS pct_cache
FROM pg_stat_statements
WHERE query ILIKE '%<=>%'
ORDER BY total_exec_time DESC
LIMIT 5;
```

With those three layers, three alerts cover almost everything: canary recall below threshold, cache hit percentage on the vector query trending down, and the share of searches returning fewer than `k` results trending up (the signature of the over-filtering from section 5).

16. Security and multi-tenancy: two things most teams leave for last

Let us start with the boring, urgent part: **upgrade the extension**. pgvector 0.8.2, released in February 2026, fixes a buffer overflow in parallel HNSW index builds (CVE-2026-3172) that can leak data from other relations or crash the server. If you build indexes with parallel workers, and section 3 recommends you do, you are on exactly the affected path.

```
SELECT extversion FROM pg_extension WHERE extname = 'vector';
-- after upgrading the system package or image:
ALTER EXTENSION vector UPDATE;
```

Tenant isolation has a twist specific to approximate search that surprises a lot of people. Row-Level Security is the correct way to guarantee that one tenant never sees another's rows, but from the ANN index's point of view an RLS policy *is a filter*, with exactly the over-filtering problem from section 5: the index returns the globally nearest candidates and the policy discards the ones that do not belong to the tenant. With many tenants, the query can come back empty even though the tenant has thousands of relevant documents.

```
ALTER TABLE documents ENABLE ROW LEVEL SECURITY;
ALTER TABLE documents FORCE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation ON documents
  USING (tenant_id = (SELECT current_setting('app.tenant_id', true)::uuid));
```

Note the `(SELECT ...)` wrapping `current_setting`: it turns the call into an InitPlan evaluated once per query instead of once per row. Without those parentheses the policy becomes the bottleneck before the index does.

To keep search correct under RLS you have two good options, and the right one depends on how tenant sizes are distributed. With a few dozen large tenants, a partial index per tenant gives exact results within that tenant's slice of the graph. With thousands of tenants, partition by hash and let pruning do the work:

```
-- Thousands of tenants: hash partitioning on tenant_id
CREATE TABLE documents (
  id uuid PRIMARY KEY,
  tenant_id uuid NOT NULL,
  body text,
  embedding halfvec(1536)
) PARTITION BY HASH (tenant_id);

CREATE TABLE documents_p0 PARTITION OF documents FOR VALUES WITH (MODULUS 8, REMAINDER 0);
-- ... p1 through p7

-- One HNSW index per partition; the planner prunes by tenant_id
CREATE INDEX ON documents_p0 USING hnsw (embedding halfvec_cosine_ops);
```

The third option, and the simplest if your scale allows it, is to enable the iterative scans from section 5 with `relaxed_order` and a generous `max_scan_tuples`. It works, but it costs latency in proportion to how small the tenant is.

One operational detail that breaks in production and never locally: `SET hnsw.ef_search` and session-level `set_config('app.tenant_id', ...)` leak between requests when there is a transaction-mode pool such as PgBouncer in front. Always use the transactional variant:

```
with conn.transaction():
    cur.execute("SELECT set_config('app.tenant_id', %s, true)", (tenant_id,))
    cur.execute("SET LOCAL hnsw.ef_search = 100")
    cur.execute(ANN_QUERY, (vector, k))
```

The third `true` argument to `set_config` and the `LOCAL` in `SET` are what binds the setting to the transaction. Without them, a request from tenant A can end up executing with tenant B's context.

17. A CI benchmark that blocks recall regressions

Everything above is lost if nobody checks it again. The changes that degrade vector search are not the dramatic ones: it is someone lowering `ef_search` to fix latency, someone adding a `WHERE` to the query, someone swapping the embedding model in a config file. None of them breaks a functional test.

What does catch them is a test that measures recall against a fixed set and fails when it drops. It needs three pieces: a small reproducible dataset, a set of golden queries, and a threshold.

```
import numpy as np
import psycopg
import pytest

DIMENSIONS = 384
N_ROWS = 20_000
RECALL_THRESHOLD = 0.95

@pytest.fixture(scope="session")
def db(postgresql_dsn):
    """Load a deterministic synthetic corpus and build the index."""
    rng = np.random.default_rng(seed=42)
    vectors = rng.normal(size=(N_ROWS, DIMENSIONS)).astype(np.float32)
    vectors /= np.linalg.norm(vectors, axis=1, keepdims=True)

    with psycopg.connect(postgresql_dsn, autocommit=True) as conn:
        conn.execute("CREATE EXTENSION IF NOT EXISTS vector")
        conn.execute("DROP TABLE IF EXISTS bench")
        conn.execute(f"CREATE TABLE bench (id int PRIMARY KEY, v vector({DIMENSIONS}))")
        with conn.cursor().copy("COPY bench (id, v) FROM STDIN") as copy:
            for i, v in enumerate(vectors):
                copy.write_row((i, "[" + ",".join(map(str, v)) + "]"))
        conn.execute("SET maintenance_work_mem = '2GB'")
        conn.execute(
            "CREATE INDEX ON bench USING hnsw (v vector_cosine_ops)"
            " WITH (m = 16, ef_construction = 64)"
        )
        conn.execute("ANALYZE bench")
        yield conn, vectors

def measured_recall(conn, vectors, ef_search: int, k: int = 10) -> float:
    rng = np.random.default_rng(seed=7)
    queries = vectors[rng.choice(len(vectors), size=100, replace=False)]
    total = 0
    for q in queries:
        literal = "[" + ",".join(map(str, q)) + "]"
        conn.execute(f"SET LOCAL hnsw.ef_search = {ef_search}")
        approximate = {r[0] for r in conn.execute(
```

```
            "SELECT id FROM bench ORDER BY v <=> %s::vector LIMIT %s", (literal, k))}
        conn.execute("SET LOCAL enable_indexscan = off")
        exact = {r[0] for r in conn.execute(
            "SELECT id FROM bench ORDER BY v <=> %s::vector LIMIT %s", (literal, k))}
        conn.execute("RESET enable_indexscan")
        total += len(approximate & exact)
    return total / (len(queries) * k)

def test_default_recall_does_not_regress(db):
    conn, vectors = db
    assert measured_recall(conn, vectors, ef_search=100) >= RECALL_THRESHOLD

@pytest.mark.parametrize("ef,minimum", [(40, 0.80), (100, 0.95), (200, 0.98)])
def test_recall_curve(db, ef, minimum):
    conn, vectors = db
    assert measured_recall(conn, vectors, ef_search=ef) >= minimum
```

The parametrized test is the more valuable of the two: it documents the full `ef_search` to recall curve for your configuration, so when someone proposes lowering the parameter to win latency, the conversation stops being an opinion and becomes a number.

Run it on every pull request that touches the retrieval layer. At 20,000 rows and 384 dimensions the whole suite finishes in under a minute on an ordinary runner, which is exactly the budget that keeps people from disabling it.

```
name: recall
on: [pull_request]
jobs:
  recall:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: pgvector/pgvector:pg17
        env:
          POSTGRES_PASSWORD: postgres
        options: >-
          --health-cmd pg_isready --health-interval 5s --health-retries 10
        ports: ['5432:5432']
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.13'
      - run: pip install -r requirements-dev.txt
      - run: pytest tests/test_recall.py -q
```

18. When pgvector stops being the answer

A guide that only defends its own tool is not much use. pgvector is excellent up to a point, and it is worth knowing where that point is before you hit it.

The practical limit is not dimensions or rows: it is memory. A well-tuned single-node pgvector deployment is comfortable up to roughly 50 million vectors, and below about 5 million it reaches 5 to 50 ms latencies without effort. Past 5 or 10 million, performance depends entirely on the HNSW graph fitting in RAM; at 50 million vectors of 768 dimensions you are looking at 150 GB or more of index alone, and at that point the machine bill becomes the main argument.

Before you leave, exhaust the levers you already know, in this order of cost-benefit:

1. `halfvec`: halves the index with a recall loss that is usually imperceptible.
2. Dimensionality reduction in the model itself, if it supports it: 3,072 down to 1,024 dimensions is a third of the index.

3. Binary quantization with reranking: divides the index by 32 in exchange for a second read phase.
4. Partitioning by tenant or by date: you never search the whole set, only the relevant partition.

If after all that you still do not fit, there are two roads that do not require leaving PostgreSQL. **pgvectorscale** adds StreamingDiskANN, an index written in Rust and designed to stay fast when the vector set is larger than available memory, working from disk instead of demanding that everything fit; its benchmarks on 50 million 768-dimension vectors report p95 latencies far below comparable managed services at 99% recall. **VectorChord** and **Lantern** occupy similar ground with different approaches. All of them keep your data, your joins, and your transactions where they already are, which is usually why you chose PostgreSQL in the first place.

Moving to a dedicated vector database makes sense when several of these hold at once, not just one:

- Hundreds of millions of vectors or more, with sustained growth.
- Search is the product, not a feature of the product.
- You need high-cardinality metadata filtering with exactness guarantees that filtered ANN does not give you.
- You have a team that can operate one more distributed system, with its replication, its backups, and its incidents.

That last point is the most underestimated. Adding a separate store means your documents and your vectors can drift apart, and now you have a distributed consistency problem where a transaction used to be the answer. Plenty of teams that migrated for performance ended up maintaining two systems for a volume that a single node with 128 GB of RAM would have served perfectly well.

The short version: stay in PostgreSQL while the arithmetic from section 11 produces a number you can pay. When it stops doing so, try the extensions that keep you inside first.

19. Case study: from recall 0.71 and 240 ms p95 to recall 0.96 and 26 ms

The system: an internal technical documentation search for a company with 40 enterprise customers. 12 million chunks, 1,536 dimensions, a mandatory `tenant_id` filter and an optional product filter. All on a single instance with 64 GB of RAM. The complaint was twofold and contradictory: search is slow, and it also does not find things.

Starting point. A `vector(1536)` column, an HNSW index with default parameters, `ef_search` at 40, and a `WHERE tenant_id = $1 AND product = $2` in front of the `ORDER BY`. Nobody had ever measured recall.

Day one: measure. The script from section 4 over 200 real queries gave recall@10 of 0.71. The p95 was 240 ms. And a third metric nobody was watching: 18% of searches returned fewer than 10 results. That figure alone explained half the complaints. It was not that the results were bad, it was that they were missing.

Sizing diagnosis. The arithmetic from section 11: 12,000,000 x 1,536 x 4 x 2 is roughly 147 GB of index on a 64 GB machine. `EXPLAIN (ANALYZE, BUFFERS)` confirmed it with thousands of `shared read` per query. The index had not fit in memory for months and nobody knew, because latency had crept up slowly.

Change 1: halfvec. Migrating the column to `halfvec(1536)` using the procedure from section 13. The index dropped to about 74 GB. Still did not fit, but recall did not move: 0.71 before, 0.71 after. Cost: two hours of backfill with no API calls, because `halfvec` is a type conversion, not a recomputation.

```
ALTER TABLE chunks ADD COLUMN embedding_h halfvec(1536);
UPDATE chunks SET embedding_h = embedding::halfvec WHERE embedding_h IS NULL;
-- in batches of 50,000 using the same structure as section 13
```

Change 2: partition by tenant. This was the big win, and it was structural rather than a raw performance trick. With 40 tenants, hash partitioning into 16 partitions means each search walks a graph of roughly 750,000

vectors instead of 12 million. The active partition fits in memory with room to spare, and the `tenant_id` filter stops being a filter that discards ANN results and becomes partition pruning the planner applies before touching the index at all.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id FROM chunks
WHERE tenant_id = '...'::uuid
ORDER BY embedding_h <=> $1 LIMIT 10;
-- Before: Index Scan over 12M, shared read = 4,812
-- After:  Append -> Index Scan over 1 partition, shared read = 0
```

Recall rose to 0.89 and p95 fell to 34 ms. The recall gain did not come from tuning parameters: it came from the filter no longer discarding candidates the index had already chosen.

Change 3: the product filter. It was still a real filter inside each partition, and it accounted for the 18% of short searches. With three possible values and queries that almost always filter, three partial indexes solved it at the root:

```
CREATE INDEX CONCURRENTLY ON chunks_p0 USING hnsw (embedding_h halfvec_cosine_ops)
  WHERE product = 'platform';
-- same for the other two products and the other 15 partitions,
-- generated with a DO block rather than by hand
```

Searches without a product filter still use the general index. Those that filter use the partial one and get the true 10 neighbours within their subset. Short searches fell from 18% to 0.4%.

Change 4: tune ef_search with data. With the index now fitting in memory, raising `ef_search` from 40 to 100 cost 8 ms and bought recall. The curve measured in staging was: 40 gives 0.89 and 26 ms; 100 gives 0.96 and 34 ms; 200 gives 0.975 and 58 ms. They chose 100 and left the curve documented in the test from section 17 so nobody changes it on intuition.

Result. Recall@10 of 0.96, a 26 ms p95 on the percentile that matters (already-warm queries), short searches practically eliminated, and as a side effect the instance went from sustained disk usage to essentially zero reads. None of the four changes was a magic parameter: three were data structure decisions, and the fourth could only be made well because the first three had removed the noise.

Frequently asked questions

Do I need a dedicated vector database?

It depends on volume and requirements. Up to tens of millions of vectors, a well-configured pgvector holds its own against specialized engines and saves you an entire system to sync, back up and monitor. You can also JOIN against your business data and filter transactionally, which is exactly what becomes awkward once vectors live elsewhere. At hundreds of millions of vectors, or if you need very high-throughput native metadata filtering, a dedicated engine starts to justify its operational cost.

HNSW or IVFFlat?

HNSW unless you have a specific reason. IVFFlat builds faster and takes less space, which matters if you rebuild often or are tight on memory, but it needs retraining when the data distribution shifts and its recall is worse at equal latency.

Can I combine vector search with keyword search?

Yes, and in most RAG systems you should. PostgreSQL's lexical search with `tsvector` finds what vectors miss

(proper nouns, error codes, exact references) and vectors find what keywords miss (synonyms, paraphrases). The usual pattern is to run both and merge the rankings with Reciprocal Rank Fusion, which avoids having to calibrate scores across two different systems.

I am switching embedding models. Can I do it without downtime?

Yes, with the expand-contract pattern. Add a second column, backfill it in batches in the background, build the new index, switch the query behind a feature flag, verify recall in production, and only then drop the old column and index. Do not try to reuse the same column: distances from different models are not comparable, and during the backfill you would be mixing two vector spaces in one result set.

What value of m should I use?

Start at 16. Raise it to 32 only if you have measured that recall falls short even at a high `ef_search`, and accept that the index will grow and the build will take longer. Changing `m` requires a rebuild; changing `ef_search` does not. Always exhaust the free dial before touching the expensive one.

How much RAM do I need, exactly?

Use `N x D x 4 x 2` bytes for `vector`, half that for `halfvec`, and verify with `pg_relation_size` as soon as you have real data. The operating rule is that the index should fit in `shared_buffers` plus the OS page cache. Section 11 has the detail.

Can I use pgvector on a read replica?

Yes, and it is a good idea: vector search is CPU and memory intensive, and isolating it from writes stops them competing. Two warnings. First: the replica needs the same RAM as the primary, because it has to cache the same index. Second: `hnsw.ef_search` is a session setting, so make sure your pool points at the replica with the same configuration; a difference in `ef_search` between nodes produces different results for the same query.

How do I back up a table with HNSW indexes?

`pg_dump` does not export index contents, only the definition, so restoring rebuilds it from scratch. For 12 million vectors that can be well over an hour even with parallel workers, and it belongs in your RTO calculation. Physical backups (`pg_basebackup`, volume snapshots) do include the index and restore far faster, at the cost of considerably more space.

Is a cross-encoder reranker after the ANN worth it?

Almost always yes, and it is the cheapest quality improvement left once you have tuned the index. Retrieve 50 candidates with pgvector, reorder them with a cross-encoder, return 10. The cross-encoder sees query and document together, so it judges far better than a distance between separately computed embeddings. The cost is tens of milliseconds, which is almost always negligible next to the generative model call that comes after.

Why did my query get worse after a deploy that changed nothing in the database?

The usual suspect is a restart: the cache emptied and the first few thousand queries read from disk. If the effect lasts more than a few minutes, check whether the instance size changed, whether someone touched `ef_search` in code, or whether the index crossed the fits-in-memory threshold as the table grew. All three are distinguishable in a minute with `EXPLAIN (ANALYZE, BUFFERS)`.

Glossary

- **ANN** (approximate nearest neighbour): nearest-neighbour search that accepts missing some results in exchange for being orders of magnitude faster than exact search.
- **Recall@K**: the share of the K true neighbours your approximate search actually found. It is the quality metric, and it has to be measured, not assumed.
- **HNSW**: hierarchical navigable small world graph. The recommended default index in pgvector: slow to build, fast to query, no training phase.
- **IVFFlat**: an inverted-list index over centroids. Fast to build and lighter on memory, but it needs representative data at creation time.
- **m**: maximum number of connections per node in the HNSW graph. Raises both recall and index size.
- **ef_construction**: candidate list width while building the graph. More quality, more build time.
- **ef_search**: candidate list width at query time. The runtime recall-versus-latency lever.
- **halfvec**: 16-bit floating point vector. Half the space with a recall loss that is almost always negligible.
- **Binary quantization**: reducing each dimension to one bit with `binary_quantize()`. A 32x smaller index that requires reranking against the original vectors.
- **Over-filtering**: the ANN index picks candidates while ignoring your `WHERE`, and the filter discards them afterwards. It shows up as queries returning fewer than K results.
- **Iterative scans**: the pgvector 0.8 mechanism that keeps expanding the index search until enough results pass the filter.
- **RRF** (reciprocal rank fusion): a way to combine several rankings by summing `1/(k + rank)`. Simple, robust, and hard to beat.