PUIGFLOWS

Búsqueda Full-Text en PostgreSQL: tsvector, GIN, Ranking y pg_trgm sin Elasticsearch

Guía · Intermedio · Proyectos Reales · Edición ampliada

Full-Text Search in PostgreSQL: tsvector, GIN, Ranking and pg_trgm without Elasticsearch

Premium

puigflows.com · Agosto 2026

Versión en Español

La versión en inglés comienza en la segunda mitad del documento.

Por qué LIKE no es un buscador

El primer impulso de casi todo equipo es resolver la búsqueda con `ILIKE '%término%'`. Funciona en la demo y muere en producción: un patrón con comodín inicial no puede aprovechar un índice B-tree, así que cada búsqueda recorre la tabla entera. Y aunque el rendimiento no doliera, la calidad sí: ILIKE no sabe que «corriendo» y «correr» son la misma palabra, no ordena por relevancia y trata una tilde de menos como un fracaso total.

PostgreSQL trae de serie un motor de búsqueda full-text que resuelve las tres cosas: normaliza palabras (stemming), indexa con una estructura invertida (GIN) y ordena por relevancia. Para la mayoría de aplicaciones —hasta decenas de millones de filas— es todo lo que necesitas, sin sincronizar datos con un segundo sistema.

Cómo funciona: tsvector, tsquery y diccionarios

El pipeline tiene dos piezas. Un `tsvector` es el documento ya procesado: texto tokenizado, filtrado de stopwords y con cada palabra reducida a su lexema. Un `tsquery` es la consulta procesada de la misma forma. El operador `@@` comprueba si el vector satisface la consulta.

```
SELECT to_tsvector('spanish', 'Los gatos corrieron por el jardín');
-- 'corr':3 'gat':2 'jardin':6

SELECT to_tsvector('spanish', 'el gato corre') @@ to_tsquery('spanish', 'gatos');
-- true: «gato» y «gatos» comparten lexema
```

La elección del diccionario importa: el mismo texto procesado con `'spanish'` y con `'english'` produce lexemas distintos. La regla de oro es usar siempre la misma configuración al indexar y al consultar, y pasarla de forma explícita en lugar de depender de `default_text_search_config`.

El esquema correcto: columna generada con pesos

La práctica recomendada desde PostgreSQL 12 es una columna generada STORED: se mantiene sola en cada INSERT y UPDATE, nunca se desincroniza y con `setweight` puedes dar más importancia al título que al cuerpo.

```
CREATE TABLE articles (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  title text NOT NULL,
  body text NOT NULL,
  tags text[] NOT NULL DEFAULT '{}',
  search tsvector GENERATED ALWAYS AS (
    setweight(to_tsvector('spanish', coalesce(title, '')), 'A') ||
    setweight(to_tsvector('spanish', coalesce(body, '')), 'B') ||
    setweight(to_tsvector('spanish',
      coalesce(array_to_string(tags, ' '), '')), 'C')
  ) STORED
);
```

Los `coalesce` no son decorativos: si cualquiera de las columnas fuera NULL, el tsvector completo sería NULL y esa fila desaparecería del buscador sin ningún error visible.

Indexado con GIN

```
CREATE INDEX idx_articles_search ON articles USING GIN (search);
```

GIN es un índice invertido: mapea cada lexema a las filas que lo contienen, exactamente como el índice alfabético de un libro. Es el tipo que la documentación oficial recomienda para búsqueda de texto; GiST solo tiene sentido en casos muy concretos con actualizaciones extremadamente frecuentes y consultas que toleren más lentitud.

Consultas de usuario con websearch_to_tsquery

Nunca pases input del usuario directamente a `to_tsquery`: su sintaxis de operadores (&, |, !) lanza errores con cualquier entrada malformada. `websearch_to_tsquery` (PostgreSQL 11+) acepta sintaxis de buscador web —comillas para frases, OR, guion para excluir— y nunca lanza excepción, lo que la convierte en la opción correcta para cualquier caja de búsqueda pública.

```
SELECT id, title
FROM articles,
     websearch_to_tsquery('spanish',
       'postgres "búsqueda full text" -elastic') AS q
WHERE search @@ q;
```

Ranking y snippets: ts_rank y ts_headline

`ts_rank` puntúa cada coincidencia según la frecuencia de los lexemas y los pesos que definiste (A > B > C > D). `ts_headline` genera el fragmento resaltado que se muestra bajo cada resultado. Un detalle que separa un buscador rápido de uno lento: `ts_headline` re-procesa el documento original completo, así que debe aplicarse solo a la página final de resultados, nunca antes del LIMIT.

```
WITH hits AS (
  SELECT id, title, body, ts_rank(search, q) AS rank
  FROM articles,
       websearch_to_tsquery('spanish', 'índices gin') AS q
  WHERE search @@ q
  ORDER BY rank DESC
  LIMIT 20
)
SELECT id, title, rank,
       ts_headline('spanish', body,
         websearch_to_tsquery('spanish', 'índices gin'),
         'StartSel=<mark>, StopSel=</mark>, MaxWords=35, MinWords=15') AS snippet
FROM hits;
```

Acentos y errores tipográficos: unaccent y pg_trgm

Dos problemas muy reales en contenido en español: la gente escribe «jardin» sin tilde y comete typos. Para lo primero está `unaccent`; como la función no está marcada como inmutable, usarla en una columna generada exige un wrapper IMMUTABLE:

```
CREATE EXTENSION IF NOT EXISTS unaccent;
```

```
CREATE OR REPLACE FUNCTION immutable_unaccent(text)
RETURNS text LANGUAGE sql IMMUTABLE PARALLEL SAFE STRICT
AS $$ SELECT public.unaccent('public.unaccent', $1) $$;

-- En la columna generada (y también al construir la query):
-- setweight(to_tsvector('spanish',
--   immutable_unaccent(coalesce(title, ''))), 'A') || ...
```

Para los typos, el full-text no ayuda: «postgress» no produce el lexema de «postgres». Ahí entra `pg_trgm`, que compara trigramas de caracteres y devuelve una similitud difusa. Un patrón que funciona bien es usarlo como fallback cuando la búsqueda exacta devuelve pocos resultados:

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;

CREATE INDEX idx_articles_title_trgm
  ON articles USING GIN (title gin_trgm_ops);

SELECT id, title, similarity(title, 'postgress') AS sim
FROM articles
WHERE title % 'postgress'  -- operador de similitud
ORDER BY sim DESC
LIMIT 5;
```

Un endpoint de búsqueda completo

Así se ve todo junto en un servicio real con FastAPI y asyncpg: consulta segura, ranking, snippet calculado solo sobre la página devuelta y un límite defensivo.

```
from contextlib import asynccontextmanager

import asyncpg
from fastapi import FastAPI, Query

SEARCH_SQL = """
WITH hits AS (
  SELECT id, title, body, ts_rank(search, q) AS rank
  FROM articles,
       websearch_to_tsquery('spanish', $1) AS q
  WHERE search @@ q
  ORDER BY rank DESC
  LIMIT $2
)
SELECT id, title, rank,
       ts_headline('spanish', body,
         websearch_to_tsquery('spanish', $1),
         'StartSel=<mark>, StopSel=</mark>, MaxWords=35') AS snippet
FROM hits
"""

@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.pool = await asyncpg.create_pool(dsn="postgresql://app@db/app")
    yield
    await app.state.pool.close()

app = FastAPI(lifespan=lifespan)
```

```
@app.get("/search")
async def search(
    q: str = Query(min_length=2, max_length=100),
    limit: int = Query(default=20, le=50),
):
    async with app.state.pool.acquire() as conn:
        rows = await conn.fetch(SEARCH_SQL, q, limit)
    return [dict(r) for r in rows]
```

Errores comunes que arruinan tu buscador

Mezclar configuraciones. Indexar con `'spanish'` y consultar con la configuración por defecto genera lexemas distintos: cero resultados y ningún error. Es el bug número uno de los buscadores en Postgres.

ts_headline antes del LIMIT. Re-procesa el documento entero por cada fila; aplicado a miles de coincidencias convierte una consulta de milisegundos en una de segundos.

Olvidar coalesce en la columna generada. Un solo campo NULL anula el tsvector completo y la fila se vuelve invisible para el buscador.

Ordenar millones de coincidencias por rank. El índice GIN encuentra las filas, pero ts_rank se calcula fila a fila y el orden no puede salir del índice. Con términos muy comunes, combina el rank con una señal precalculada de popularidad o exige términos más específicos.

Ignorar la pending list de GIN. Con fastupdate (activo por defecto) las escrituras van a una lista pendiente que se fusiona después; si crece demasiado aparecen picos de latencia. Vigila autovacuum y ajusta gin_pending_list_limit si hace falta.

Documentos gigantes. Un tsvector no puede superar 1 MB y las posiciones se recortan a 16383; para textos enormes, indexa un resumen o trocea el documento.

¿Cuándo sí necesitas un motor dedicado?

La línea práctica es esta: Postgres gana mientras la búsqueda es una funcionalidad; un motor dedicado gana cuando la búsqueda es el producto. Elasticsearch u OpenSearch aportan relevancia BM25, sinónimos, corrección ortográfica integrada, facetas a gran escala y escalado horizontal del índice. El precio: operar otro sistema y resolver la sincronización de datos (CDC, reindexados, consistencia eventual). Un punto intermedio reciente es ParadeDB con pg_search, que lleva ranking BM25 al propio Postgres. La recomendación honesta: empieza en Postgres, mide la calidad de tus resultados con consultas reales, y migra solo cuando tengas evidencia de que te quedaste corto.

Checklist de producción

- Configuración de idioma explícita e idéntica en índice y consultas.
- Columna generada STORED con coalesce y pesos por campo.
- Índice GIN sobre la columna tsvector.
- websearch_to_tsquery para todo input de usuario, con longitud máxima validada.
- ts_headline solo sobre la página final de resultados.
- unaccent (con wrapper IMMUTABLE) si tu contenido lleva tildes; pg_trgm como fallback para typos.

- EXPLAIN ANALYZE de las consultas reales y métricas de latencia p95.
- Un dataset de consultas reales para medir relevancia antes de ajustar pesos.

Edición ampliada: de buscador funcional a buscador de producción

Todo lo anterior te deja un buscador correcto. Esta edición ampliada cubre lo que aparece después: autocompletar, sinónimos, ranking afinado con señales de negocio, contenido multiidioma, migraciones sobre tablas grandes, el comportamiento interno de GIN bajo carga de escritura, búsqueda híbrida con embeddings y cómo medir si tus resultados son buenos de verdad. Cada sección es independiente: puedes aplicarlas en el orden que tu producto las pida.

Las cuatro funciones de tsquery y cuándo usar cada una

PostgreSQL trae cuatro formas de convertir texto de usuario en un `tsquery`, y elegir mal es la segunda causa más común de buscadores decepcionantes (la primera sigue siendo mezclar configuraciones de idioma).

to_tsquery acepta la sintaxis completa de operadores: `&` (AND), `|` (OR), `!` (NOT), `<->` (seguido de) y agrupación con paréntesis. Es la más potente y la más peligrosa: cualquier entrada malformada lanza una excepción. Resérvala para consultas que construye tu código, nunca para input directo del usuario.

plainto_tsquery ignora toda sintaxis y une cada palabra con AND. Es segura pero rígida: el usuario no puede excluir términos ni buscar frases.

phraseto_tsquery también ignora operadores, pero conecta los términos con `<->`, exigiendo que aparezcan consecutivos y en orden. Es la base de la búsqueda de frases exactas.

websearch_to_tsquery combina lo mejor: comillas para frases, `OR`, guion para excluir, y jamás lanza una excepción. Para una caja de búsqueda pública es la elección correcta el 95 % de las veces.

```
SELECT to_tsquery('spanish', 'gato & !perro');        -- sintaxis completa, puede fallar
SELECT plainto_tsquery('spanish', 'gato perro');       -- 'gat' & 'perr'
SELECT phraseto_tsquery('spanish', 'gato negro');      -- 'gat' <-> 'negr'
SELECT websearch_to_tsquery('spanish', '"gato negro" OR siames -persa');
-- 'gat' <-> 'negr' | 'siames' & !'pers'
```

Autocompletar: prefijos con :* y búsqueda incremental

Un buscador moderno sugiere resultados mientras el usuario teclea. El full-text de Postgres lo soporta con el modificador de prefijo `:*`, que coincide con cualquier lexema que empiece por la cadena dada. El detalle clave: `websearch_to_tsquery` no entiende `:*` (lo trata como texto literal), así que la consulta de autocompletar hay que construirla con `to_tsquery` a partir de input previamente saneado.

```
-- Patrón: todas las palabras completas en AND, la última como prefijo
-- Input del usuario: "indices gi"
SELECT id, title
FROM articles,
     to_tsquery('spanish', 'indices & gi:*') AS q
WHERE search @@ q
ORDER BY ts_rank(search, q) DESC
LIMIT 8;
```

En el backend, sanea antes de construir: elimina todo lo que no sea letra, número o espacio, trocea por espacios y une con `&` añadiendo `:*` al último término.

```
import re

def autocomplete_query(raw: str) -> str:
    """Convierte input libre en un tsquery de prefijo seguro."""
    words = re.sub(r"[^\w\s]", " ", raw, flags=re.UNICODE).split()
    if not words:
        return ""
    *completas, ultima = words
    parts = [w for w in completas] + [ultima + ":*"]
    return " & ".join(parts)

# autocomplete_query("indices gi") -> "indices & gi:*"
```

Dos consejos de producción: aplica un debounce de 150–250 ms en el cliente para no disparar una consulta por tecla, y limita el prefijo a un mínimo de 2–3 caracteres — un `:*` de una sola letra coincide con media tabla y castiga al índice. Si además quieres tolerancia a typos en el autocompletar, combina este patrón con el fallback de `pg_trgm` que amplío más abajo.

Configuraciones a medida: unaccent integrado en el pipeline

El wrapper `immutable_unaccent` de la primera parte funciona, pero obliga a acordarte de aplicarlo en la columna y en cada consulta. La solución más limpia es crear una configuración de búsqueda propia que incorpore `unaccent` como filtro del pipeline: se define una vez y a partir de ahí indexas y consultas con ella como con cualquier otra.

```
CREATE EXTENSION IF NOT EXISTS unaccent;

CREATE TEXT SEARCH CONFIGURATION es_sin_acentos (COPY = spanish);

ALTER TEXT SEARCH CONFIGURATION es_sin_acentos
  ALTER MAPPING FOR hword, hword_part, word
  WITH unaccent, spanish_stem;

-- Ahora el pipeline quita acentos antes del stemming:
SELECT to_tsvector('es_sin_acentos', 'El jardín de María');
-- 'jardin':3 'mari':5

SELECT to_tsvector('es_sin_acentos', 'jardín')
       @@ websearch_to_tsquery('es_sin_acentos', 'jardin');
-- true: con y sin tilde convergen al mismo lexema
```

Con esto, la columna generada usa `'es_sin_acentos'` directamente y no necesitas ningún wrapper. La regla de siempre se mantiene: la misma configuración en el índice y en todas las consultas.

Sinónimos: cuando «factura» y «recibo» deben encontrarse

Ningún stemmer sabe que en tu dominio «pgsql», «postgres» y «postgresql» son la misma cosa. Para eso existen los diccionarios de sinónimos: un archivo de pares en el directorio `tsearch_data` del servidor que se inserta en el pipeline antes del stemmer.

```
-- Archivo $SHAREDIR/tsearch_data/mis_sinonimos.syn:
-- pgsql postgres
-- postgresql postgres
-- factura recibo

CREATE TEXT SEARCH DICTIONARY sinonimos_es (
  TEMPLATE = synonym,
  SYNONYMS = mis_sinonimos
);

ALTER TEXT SEARCH CONFIGURATION es_sin_acentos
  ALTER MAPPING FOR word, hword, hword_part
  WITH unaccent, sinonimos_es, spanish_stem;
```

La advertencia honesta: esto requiere escribir un archivo en el sistema de archivos del servidor, algo que la mayoría de servicios gestionados (RDS, Cloud SQL, Supabase) no permite. En esos entornos la alternativa práctica es expandir sinónimos en la capa de aplicación: mantén un diccionario en una tabla y reescribe la consulta añadiendo alternativas con `OR` antes de pasarla a `websearch_to_tsquery`. Es menos elegante pero funciona en cualquier hosting y puedes editarlo sin tocar el servidor.

Ranking a fondo: ts_rank, ts_rank_cd y normalización

Hasta ahora usamos `ts_rank` con sus valores por defecto, que puntúa solo por frecuencia de lexemas y pesos. Hay dos palancas más que cambian resultados de forma visible.

La primera es `ts_rank_cd` (cover density): además de la frecuencia, premia que los términos de la consulta aparezcan *cerca unos de otros*. Para consultas de varias palabras suele ordenar mejor: un documento donde «índice» y «GIN» aparecen en la misma frase queda por encima de otro donde están a veinte párrafos de distancia. Requiere que el tsvector conserve las posiciones (no uses `strip()` si quieres usarlo).

La segunda es el parámetro de normalización, un bitmask que corrige el sesgo hacia documentos largos. Los valores útiles en la práctica:

- `0` (por defecto): sin normalización; los documentos largos acumulan más coincidencias y suben artificialmente.
- `1`: divide entre 1 + log(longitud del documento) — la corrección suave que funciona bien casi siempre.
- `2`: divide entre la longitud completa; castiga fuerte a los largos.
- `32`: comprime el resultado a rank/(rank+1), útil si necesitas una escala 0–1 para combinar con otras señales.

```
SELECT id, title,
       ts_rank_cd(search, q, 1) AS rank  -- cover density + normalización log
FROM articles,
     websearch_to_tsquery('es_sin_acentos', 'indices gin') AS q
WHERE search @@ q
ORDER BY rank DESC
LIMIT 20;
```

Combinar relevancia con señales de negocio

El ranking textual puro rara vez es el ranking que tu producto necesita. Un artículo viejo y popular puede merecer más visibilidad que uno nuevo sin lecturas, o al revés. El patrón que escala bien: normaliza el rank textual con `32`, y

combínalo multiplicativa o aditivamente con señales precalculadas — popularidad logarítmica y un decaimiento temporal.

```
SELECT id, title,
       ts_rank_cd(search, q, 32) AS text_rank,
       (0.6 * ts_rank_cd(search, q, 32)
      + 0.3 * (ln(1 + view_count) / ln(1 + max_views.mx))
      + 0.1 * exp(-extract(epoch FROM now() - published_at)
                  / 86400.0 / 90)) AS score
FROM articles,
     websearch_to_tsquery('es_sin_acentos', 'postgres') AS q,
     (SELECT max(view_count) AS mx FROM articles) AS max_views
WHERE search @@ q
ORDER BY score DESC
LIMIT 20;
```

Los pesos (0.6 / 0.3 / 0.1) no son sagrados: son el punto de partida que ajustarás midiendo con consultas reales, como veremos en la sección de evaluación. Lo importante es la estructura — relevancia textual como señal dominante, popularidad amortiguada con logaritmo para que un solo artículo viral no monopolice todos los resultados, y recencia con semivida explícita (aquí 90 días).

Contenido multiidioma en una sola tabla

Si tu aplicación sirve contenido en español y en inglés, la tentación es indexar todo con una sola configuración. No lo hagas: el stemmer español convierte «running» en «running» y el inglés convierte «corriendo» en «corriendo» — cada idioma mal procesado pierde el stemming por completo. El patrón correcto es una columna de idioma y un tsvector generado que elige la configuración por fila.

```
CREATE TABLE posts (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  lang text NOT NULL CHECK (lang IN ('es', 'en')),
  title text NOT NULL,
  body text NOT NULL,
  search tsvector GENERATED ALWAYS AS (
    CASE lang
      WHEN 'es' THEN
        setweight(to_tsvector('es_sin_acentos', coalesce(title, '')), 'A') ||
        setweight(to_tsvector('es_sin_acentos', coalesce(body,  '')), 'B')
      ELSE
        setweight(to_tsvector('english', coalesce(title, '')), 'A') ||
        setweight(to_tsvector('english', coalesce(body,  '')), 'B')
    END
  ) STORED
);
```

Para las consultas hay dos opciones de indexado. Un único índice GIN sobre `search` funciona; si el reparto de idiomas está muy sesgado o quieres índices más pequeños, crea índices parciales por idioma:

```
CREATE INDEX idx_posts_search_es ON posts USING GIN (search) WHERE lang = 'es';
CREATE INDEX idx_posts_search_en ON posts USING GIN (search) WHERE lang = 'en';

-- La consulta SIEMPRE filtra por idioma y usa la configuración de ese idioma:
SELECT id, title
FROM posts,
```

```
     websearch_to_tsquery('es_sin_acentos', 'indices') AS q
WHERE lang = 'es' AND search @@ q;
```

La disciplina que esto exige: el idioma de la interfaz decide la configuración de la consulta, y ambas cosas viajan juntas. Si un usuario busca en inglés dentro de contenido en español tendrá stemming pobre — es una limitación honesta de este enfoque, y la razón de que la búsqueda híbrida con embeddings (más abajo) sea tan buen complemento: los embeddings multilingües cruzan idiomas sin esfuerzo.

pg_trgm a fondo: similitud, umbrales y aceleración de ILIKE

La primera parte usó `pg_trgm` como fallback para typos. La extensión da bastante más de sí, y entender sus tres funciones de similitud evita falsos positivos frustrantes.

similarity(a, b) compara los conjuntos completos de trigramas de ambas cadenas. Funciona bien entre cadenas de longitud parecida (títulos, nombres), pero castiga comparar una palabra contra un texto largo.

word_similarity(a, b) busca la mejor coincidencia de `a` contra *cualquier parte* de `b`. Es la correcta cuando el usuario teclea una palabra y tú la buscas dentro de títulos completos. Su operador es `<%`.

strict_word_similarity(a, b) igual, pero respetando límites de palabra — menos falsos positivos con palabras cortas.

```
SET pg_trgm.similarity_threshold = 0.3;       -- umbral del operador %
SET pg_trgm.word_similarity_threshold = 0.4;  -- umbral del operador <%

-- Typo contra títulos: word_similarity es la herramienta correcta
SELECT id, title, word_similarity('postgress', title) AS sim
FROM articles
WHERE 'postgress' <% title
ORDER BY sim DESC
LIMIT 5;
```

Sobre índices: `gin_trgm_ops` acelera los operadores de similitud y — un regalo que mucha gente desconoce — también las búsquedas `LIKE` e `ILIKE` con comodín inicial, las mismas que al principio dijimos que no podían usar índices. Con un índice trigram, `ILIKE '%factura%'` pasa de scan secuencial a búsqueda indexada:

```
CREATE INDEX idx_articles_title_trgm
  ON articles USING GIN (title gin_trgm_ops);

EXPLAIN (ANALYZE, BUFFERS)
SELECT id FROM articles WHERE title ILIKE '%factura%';
-- Bitmap Index Scan on idx_articles_title_trgm  ✓
```

La variante GiST (`gist_trgm_ops`) es más lenta en lecturas pero soporta el operador de distancia `<->` con orden por índice (KNN), útil para «los 5 más parecidos» sin calcular la similitud de toda la tabla. Para el caso típico de fallback de typos, GIN es la elección por defecto. Dos precauciones: los índices trigram crecen rápido (pueden superar el tamaño de la columna indexada) y las cadenas de menos de 3 caracteres no generan trigramas útiles — valida longitud mínima antes de delegar en ellos.

Paginación de resultados sin OFFSET

La página 1 de un buscador es rápida; la página 50 con `OFFSET 980` obliga a calcular el rank de 1.000 filas para descartar 980. Con términos comunes, cada página es más lenta que la anterior. La solución es la misma keyset pagination que usarías en cualquier listado, con un matiz: el rank tiene empates, así que el cursor necesita un desempate estable.

```
-- Página 1
SELECT id, title, ts_rank_cd(search, q, 32) AS rank
FROM articles, websearch_to_tsquery('es_sin_acentos', 'postgres') AS q
WHERE search @@ q
ORDER BY rank DESC, id DESC
LIMIT 20;

-- Página siguiente: el cliente devuelve (last_rank, last_id)
SELECT id, title, ts_rank_cd(search, q, 32) AS rank
FROM articles, websearch_to_tsquery('es_sin_acentos', 'postgres') AS q
WHERE search @@ q
  AND (ts_rank_cd(search, q, 32), id) < ($1, $2)
ORDER BY rank DESC, id DESC
LIMIT 20;
```

La comparación de tuplas `(rank, id) < ($1, $2)` maneja los empates de rank correctamente. Codifica el cursor como un token opaco en base64 para tu API, igual que harías con cualquier paginación keyset.

Queda el contador de resultados. Un `count(*)` exacto sobre un término común recorre todas las coincidencias y puede costar más que la búsqueda entera. Dos alternativas honestas: un contador acotado («más de 1.000 resultados») que corta en un límite fijo, o la estimación del planificador para consultas donde la precisión no importa.

```
-- Contador acotado: nunca cuenta más de 1.001 filas
SELECT count(*) FROM (
  SELECT 1
  FROM articles, websearch_to_tsquery('es_sin_acentos', 'postgres') AS q
  WHERE search @@ q
  LIMIT 1001
) t;
-- Si devuelve 1001, muestra "más de 1.000 resultados"
```

Migrar una tabla grande sin downtime

Todo lo anterior asume que puedes crear la tabla desde cero. La realidad habitual es una tabla de millones de filas en producción a la que quieres añadir búsqueda. Aquí hay una trampa seria: `ALTER TABLE ... ADD COLUMN ... GENERATED ALWAYS AS ... STORED` reescribe la tabla completa bajo un lock `ACCESS EXCLUSIVE` — en una tabla grande, eso es una caída del servicio.

La ruta sin downtime usa una columna normal mantenida por trigger, backfill por lotes e índice concurrente:

```
-- 1. Columna normal, instantánea (solo cambia el catálogo)
ALTER TABLE articles ADD COLUMN search tsvector;

-- 2. Trigger para filas nuevas y actualizadas
CREATE OR REPLACE FUNCTION articles_search_update() RETURNS trigger AS $$
BEGIN
  NEW.search :=
```

```
    setweight(to_tsvector('es_sin_acentos', coalesce(NEW.title, '')), 'A') ||
    setweight(to_tsvector('es_sin_acentos', coalesce(NEW.body,  '')), 'B');
  RETURN NEW;
END $$ LANGUAGE plpgsql;

CREATE TRIGGER trg_articles_search
  BEFORE INSERT OR UPDATE OF title, body ON articles
  FOR EACH ROW EXECUTE FUNCTION articles_search_update();
```

```
# 3. Backfill por lotes desde la aplicación (nunca un UPDATE masivo)
import asyncpg, asyncio

BATCH = """
UPDATE articles SET search =
  setweight(to_tsvector('es_sin_acentos', coalesce(title, '')), 'A') ||
  setweight(to_tsvector('es_sin_acentos', coalesce(body,  '')), 'B')
WHERE id IN (
  SELECT id FROM articles
  WHERE search IS NULL
  ORDER BY id
  LIMIT 5000
  FOR UPDATE SKIP LOCKED
)
"""

async def backfill(dsn: str):
    conn = await asyncpg.connect(dsn)
    while True:
        result = await conn.execute(BATCH)
        if result == "UPDATE 0":
            break
        await asyncio.sleep(0.5)  # respiro para autovacuum y réplicas
    await conn.close()
```

```
-- 4. Índice sin bloquear escrituras
CREATE INDEX CONCURRENTLY idx_articles_search ON articles USING GIN (search);
```

Los lotes de 5.000 con pausa evitan inflar el WAL de golpe y dan aire a autovacuum; `FOR UPDATE SKIP LOCKED` permite lanzar varios workers de backfill sin que se pisen. Cuando el backfill termina y el índice está creado, tu endpoint de búsqueda puede activarse con un feature flag. La columna por trigger queda como solución permanente perfectamente válida — la columna generada es más limpia, pero no justifica una reescritura con lock en una tabla que ya está en producción.

GIN por dentro: la pending list y el coste real de escribir

Para operar un buscador con carga de escritura constante conviene entender qué hace GIN con cada INSERT. Un documento medio produce decenas o cientos de lexemas, y actualizar la entrada de cada uno en el índice invertido por cada fila insertada sería carísimo. Por eso GIN tiene `fastupdate` (activo por defecto): las filas nuevas van a una *pending list* sin estructura, que se fusiona con el índice principal más tarde — durante autovacuum, o cuando la lista supera `gin_pending_list_limit` (4 MB por defecto).

El efecto visible: las búsquedas también tienen que escanear la pending list linealmente, así que si crece demasiado aparecen picos de latencia aparentemente aleatorios. Y el proceso (backend o autovacuum) al que le toca fusionar la lista paga un pico de escritura. Las palancas:

```
-- Ver el tamaño actual de la pending list de un índice
CREATE EXTENSION IF NOT EXISTS pgstattuple;
SELECT * FROM gin_metapage_info(get_raw_page('idx_articles_search', 0));

-- Fusionar manualmente (por ejemplo, tras una carga masiva)
SELECT gin_clean_pending_list('idx_articles_search');

-- Ajustar el umbral por índice
ALTER INDEX idx_articles_search SET (gin_pending_list_limit = 1024); -- KB

-- O desactivar fastupdate: escrituras más lentas pero uniformes,
-- lecturas siempre predecibles. Razonable con poca escritura.
ALTER INDEX idx_articles_search SET (fastupdate = off);
```

La heurística: si tu carga es de lecturas dominantes con escrituras esporádicas, `fastupdate = off` elimina la variabilidad. Si ingieres contenido en lotes grandes, deja fastupdate activo y llama a `gin_clean_pending_list` al terminar cada lote. Y en ambos casos, vigila que autovacuum visite la tabla con frecuencia — un autovacuum hambriento es la causa número uno de pending lists gigantes.

Para leer un plan de búsqueda, la forma sana es `EXPLAIN (ANALYZE, BUFFERS)`: verás un `Bitmap Index Scan` sobre el índice GIN seguido de un `Bitmap Heap Scan` con una línea `Recheck Cond`. El recheck es normal (GIN puede dar falsos positivos internos que se verifican contra la fila real); lo que debe alarmarte es un `Seq Scan` donde esperabas el índice — casi siempre significa que la expresión de la consulta no coincide exactamente con la expresión indexada, o que mezclaste configuraciones.

Búsqueda híbrida: full-text + pgvector con Reciprocal Rank Fusion

El full-text encuentra lo que contiene las palabras exactas; los embeddings encuentran lo que *significa* lo mismo aunque no comparta ni una palabra («cómo acelerar consultas» ↔ «optimización de rendimiento en SQL»). En 2026 la práctica estándar es combinar ambos — búsqueda híbrida — y Postgres lo hace en una sola consulta con `pgvector`. Los benchmarks públicos de sistemas RAG muestran mejoras de precisión de recuperación del orden de 20 puntos al añadir la señal léxica a la vectorial, con las consultas de coincidencia exacta (nombres, códigos de error, identificadores) como las grandes beneficiadas.

```
CREATE EXTENSION IF NOT EXISTS vector;

ALTER TABLE articles ADD COLUMN embedding vector(1536);

-- HNSW: el índice vectorial recomendado para consultas rápidas
CREATE INDEX idx_articles_embedding ON articles
  USING hnsw (embedding vector_cosine_ops);
```

La fusión se hace con Reciprocal Rank Fusion (RRF), que no necesita normalizar las puntuaciones de dos sistemas incomparables: cada resultado aporta `1 / (k + posición)` en cada lista y se suman. Un documento bien posicionado en ambas listas domina; uno presente en una sola sigue teniendo voz. La constante `k = 60` es el estándar de la literatura.

```
WITH fts AS (
  SELECT id, row_number() OVER (ORDER BY ts_rank_cd(search, q, 32) DESC) AS r
  FROM articles, websearch_to_tsquery('es_sin_acentos', $1) AS q
  WHERE search @@ q
  LIMIT 40
),
sem AS (
  SELECT id, row_number() OVER (ORDER BY embedding <=> $2) AS r
  FROM articles
  ORDER BY embedding <=> $2
  LIMIT 40
)
SELECT a.id, a.title,
       coalesce(1.0 / (60 + fts.r), 0) +
       coalesce(1.0 / (60 + sem.r), 0) AS rrf_score
FROM fts FULL OUTER JOIN sem USING (id)
JOIN articles a USING (id)
ORDER BY rrf_score DESC
LIMIT 20;
```

En el backend, el flujo es: calcula el embedding de la consulta (una llamada a tu proveedor o a un modelo local), y pasa el texto y el vector como parámetros. El coste añadido es esa llamada de embedding — cachéala para consultas repetidas — y el índice HNSW en disco. Dos matices de producción: el `FULL OUTER JOIN` es esencial (un resultado solo-léxico o solo-semántico debe sobrevivir), y los `LIMIT 40` de cada rama controlan el equilibrio entre calidad y latencia; súbelos si tu página final es grande. Si más adelante necesitas ranking BM25 real dentro de Postgres, extensiones como `pg_search` de ParadeDB lo aportan sin salir de la base de datos — el mismo criterio de siempre: migra cuando midas que te hace falta, no antes.

Observabilidad: registra cada búsqueda, sobre todo las que fallan

Un buscador sin telemetría es un buscador que se degrada en silencio. Las dos métricas que más información dan por euro invertido: la **tasa de cero resultados** (qué buscó la gente y no encontró nada) y el **CTR por posición** (si nadie hace clic en los tres primeros, tu ranking está mal). Ambas salen de una tabla de log barata que escribes desde el endpoint:

```
CREATE TABLE search_log (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  query text NOT NULL,
  results_count int NOT NULL,
  latency_ms int NOT NULL,
  clicked_id bigint,          -- se rellena vía evento de clic
  clicked_position int,
  created_at timestamptz NOT NULL DEFAULT now()
);

-- Las 20 búsquedas sin resultados más frecuentes de la semana:
SELECT query, count(*) AS veces
FROM search_log
WHERE results_count = 0
  AND created_at > now() - interval '7 days'
GROUP BY query
ORDER BY veces DESC
LIMIT 20;
```

Esa consulta de cero resultados es oro puro: ahí aparecen los sinónimos que faltan en tu diccionario, los typos que tu fallback de pg_trgm no cubre y el vocabulario real de tus usuarios que tu contenido no usa. Revisarla una vez por semana y convertir los patrones en sinónimos o contenido nuevo es la mejora de buscador con mejor relación esfuerzo/resultado que existe.

En el lado de infraestructura, `pg_stat_statements` te da la latencia media y p95 de la consulta de búsqueda real, y el tamaño del índice se vigila con `pg_relation_size`. Si el índice crece muy por encima del contenido (los GIN acumulan bloat con updates frecuentes), `REINDEX INDEX CONCURRENTLY` lo compacta sin cortar el servicio.

Medir la relevancia: un eval mínimo que cabe en CI

Ajustar pesos de ranking «a ojo» es cambiar una cosa que no mides por otra que tampoco. El antídoto es pequeño: un dataset dorado de consultas reales con los documentos que *deberían* aparecer, y una métrica simple — recall@10: ¿está el documento esperado entre los diez primeros?

```
GOLDEN = [
    ("indices gin", {12, 47}),          # ids que deben aparecer
    ("busqueda sin acentos", {103}),
    ("postgress typo", {12}),           # valida el fallback trigram
    ("como acelerar consultas", {55, 12}),  # valida la rama semantica
]

async def test_recall_at_10(search_client):
    fallos = []
    for query, esperados in GOLDEN:
        ids = {r["id"] for r in await search_client.search(query, limit=10)}
        if not esperados & ids:
            fallos.append(query)
    # Exige 90% de aciertos para pasar el CI
    assert len(fallos) <= len(GOLDEN) * 0.1, f"Consultas sin acierto: {fallos}"
```

Treinta consultas bastan para empezar; aliméntalo con las búsquedas reales del search_log. A partir de ahí, cada cambio de pesos, de configuración o de diccionario corre contra el eval antes de desplegarse — exactamente igual que harías con cualquier otra regresión.

Caso práctico: el buscador de una base de conocimiento

Para aterrizar todo, el recorrido completo de un caso realista: una base de conocimiento interna con 800.000 artículos en español e inglés, 2.000 escrituras al día y un pico de 50 búsquedas por segundo.

Semana 1 — lo básico bien hecho. Configuración `es_sin_acentos` + `english` con columna por trigger (la tabla ya existía: nada de reescrituras), backfill por lotes en una noche, `CREATE INDEX CONCURRENTLY`, endpoint con `websearch_to_tsquery`, ranking `ts_rank_cd(search, q, 1)` y `ts_headline` solo sobre la página devuelta. Latencia p95: 38 ms. La búsqueda pasó de «inutilizable» a «correcta» sin tocar la infraestructura.

Semana 3 — los datos mandan. El search_log mostró un 14 % de búsquedas con cero resultados. La mitad eran typos (se activó el fallback pg_trgm con `word_similarity` cuando el full-text devuelve menos de 3 resultados) y un tercio eran vocabulario: la gente buscaba «vpn caida» y los artículos decían «conectividad de red privada». Veinte sinónimos aplicados en la capa de aplicación bajaron la tasa de cero resultados al 5 %.

Mes 2 — híbrida donde aporta. Se añadió `pgvector` con embeddings multilingües y RRF. El mayor salto no fue en las búsquedas típicas — fue en las consultas largas tipo pregunta («por qué no me llega el correo de recuperación») y

en las búsquedas en inglés sobre contenido en español, que los embeddings cruzan sin esfuerzo. El eval de 60 consultas subió de 78 % a 91 % de recall@10. El coste operativo real: 6 GB más de índice HNSW y una llamada de embedding por búsqueda, cacheada con un TTL de un día para las consultas repetidas.

La lección transversal: cada mejora fue *medida* antes de ser adoptada, y ninguna exigió un segundo sistema. El día que este equipo necesite facetas complejas a gran escala o corrección ortográfica integrada, migrará a un motor dedicado con un dataset de evaluación ya construido — la mejor póliza de seguro posible para una migración.

Preguntas frecuentes

¿Necesito reindexar si cambio la configuración de búsqueda? Sí. La configuración se aplica al construir el tsvector, así que cambiar el pipeline (añadir unaccent, sinónimos) exige regenerar la columna y, con ella, el índice. Con la columna por trigger: UPDATE por lotes igual que el backfill inicial.

¿Puedo buscar en JSONB? Sí: extrae los campos de texto y concaténalos en el tsvector generado, o usa `jsonb_to_tsvector`, que indexa los valores de un documento JSON completo con un filtro por tipo de dato.

¿Full-text o trigram para nombres de persona? Trigram. Los nombres no se benefician del stemming y sí sufren typos y variantes; `word_similarity` con un umbral de 0.4 suele dar el mejor equilibrio.

¿Cómo manejo documentos PDF o HTML? Extrae el texto plano en la aplicación antes de guardar (con strip de etiquetas); indexar HTML crudo mete tags como lexemas y arruina el ranking.

¿El orden de las palabras importa en websearch_to_tsquery? No, salvo dentro de comillas (frase exacta). Los términos sueltos se combinan con AND sin importar el orden.

¿Qué pasa con las palabras muy cortas o stopwords? Las stopwords del idioma («el», «de», «y») se eliminan en el pipeline y no son buscables. Si tu dominio necesita buscar «IT» o «IA», considera un diccionario simple sin stopwords para el campo título.

¿Cuánta RAM necesita esto? La regla práctica: el índice GIN caliente debe caber en shared_buffers + page cache. Un GIN sobre un millón de documentos de tamaño medio ronda 1–3 GB; medirlo con `pg_relation_size` es trivial y vale más que cualquier estimación.

¿Sirve esto para logs o para métricas? No es su punto fuerte. Para logs, las herramientas de la familia Loki/Elasticsearch con retención por tiempo encajan mejor; el full-text de Postgres brilla con contenido editorial: artículos, productos, tickets, documentación.

Multi-tenant: aislar la búsqueda por cliente

En un SaaS, cada búsqueda lleva un `WHERE tenant_id = $n` obligatorio, y eso interactúa con GIN de una forma poco intuitiva: GIN indexa el tsvector, no el tenant, así que Postgres tiene que encontrar todas las coincidencias textuales y filtrar el tenant después. Con miles de tenants y términos comunes, eso es trabajo desperdiciado a escala. Hay tres estrategias, por orden de esfuerzo:

1. Confiar en el bitmap combinado. Con un B-tree en `tenant_id` y el GIN en `search`, el planificador puede hacer un BitmapAnd de ambos índices. Para la mayoría de cargas es suficiente — verifica con `EXPLAIN (ANALYZE, BUFFERS)` que el BitmapAnd aparece de verdad.

2. Índice GIN compuesto con btree_gin. La extensión `btree_gin` permite meter columnas escalares dentro de un índice GIN, de modo que tenant y texto se resuelven en una sola pasada de índice:

```
CREATE EXTENSION IF NOT EXISTS btree_gin;

CREATE INDEX idx_articles_tenant_search
  ON articles USING GIN (tenant_id, search);

-- Ahora este WHERE se resuelve entero dentro del indice:
SELECT id, title
FROM articles, websearch_to_tsquery('es_sin_acentos', $1) AS q
WHERE tenant_id = $2 AND search @@ q;
```

3. Particionado por tenant (o por hash de tenant). Si además del filtrado necesitas retención o límites de tamaño por cliente, particionar la tabla convierte cada búsqueda en un escaneo de una sola partición con su propio GIN pequeño. Es la opción más cara de operar; resérvala para cuando los números la pidan.

Si usas Row-Level Security, las políticas se aplican con normalidad sobre las filas candidatas que devuelve el índice — la búsqueda no abre ningún agujero nuevo. El coste es el mismo del punto 1: el filtrado de RLS ocurre después del índice, así que el consejo del BitmapAnd o btree_gin aplica igual.

Defender el endpoint: límites, timeouts y consultas patológicas

Un endpoint de búsqueda público es una superficie de ataque de denegación de servicio barata: consultas larguísimas, decenas de términos, frases enormes entre comillas o prefijos de una letra pueden costar órdenes de magnitud más que una búsqueda normal. Defensas en capas, todas baratas:

```
from fastapi import FastAPI, HTTPException, Query

MAX_TERMS = 8

@app.get("/search")
async def search(
    q: str = Query(min_length=2, max_length=100),
    limit: int = Query(default=20, le=50),
):
    if len(q.split()) > MAX_TERMS:
        raise HTTPException(422, "Demasiados términos de búsqueda")
    async with app.state.pool.acquire() as conn:
        # Techo duro por consulta: nada puede colgarse mas de 300 ms
        await conn.execute("SET LOCAL statement_timeout = '300ms'")
        rows = await conn.fetch(SEARCH_SQL, q, limit)
    return [dict(r) for r in rows]
```

El `SET LOCAL statement_timeout` es la red de seguridad definitiva: ninguna combinación patológica de términos puede retener una conexión del pool más de 300 ms — devuelve un error controlado que tu API traduce en «afina tu búsqueda». Complementa esto con el rate limiting por usuario que ya deberías tener en el gateway, y con una regla de producto: los prefijos `:*` solo a partir de 3 caracteres, como vimos en autocompletar.

Una defensa adicional para términos extremadamente comunes es `gin_fuzzy_search_limit`: un límite blando (desactivado por defecto) que corta la cantidad de resultados que GIN devuelve por búsqueda. Es un compromiso — introduce aleatoriedad en qué resultados aparecen — pero para cajas de búsqueda donde «postgres» coincide con el 40 % de la tabla, un valor de 5.000–10.000 mantiene la latencia plana sin que el usuario note diferencia en las primeras páginas.

Facetas: contar resultados por categoría sin segunda consulta

Las facetas («Guías (12) · Tutoriales (5) · Casos (2)») se resuelven con la misma consulta de búsqueda y agregación condicional, evitando una ida y vuelta extra:

```
WITH hits AS (
  SELECT id, category, type
  FROM articles, websearch_to_tsquery('es_sin_acentos', $1) AS q
  WHERE search @@ q
)
SELECT
  (SELECT count(*) FROM hits) AS total,
  (SELECT jsonb_object_agg(category, n) FROM (
     SELECT category, count(*) AS n FROM hits GROUP BY category
  ) c) AS por_categoria,
  (SELECT jsonb_object_agg(type, n) FROM (
     SELECT type, count(*) AS n FROM hits GROUP BY type
  ) t) AS por_tipo;
```

La advertencia de escala: esta agregación cuenta *todas* las coincidencias, así que hereda el problema del count exacto. Con términos comunes en tablas grandes, aplica el mismo tope defensivo (un `LIMIT 5000` dentro del CTE) y muestra «5.000+» — nadie toma decisiones distintas entre 5.000 y 80.000 resultados en una faceta.

Snippets afinados: las opciones de ts_headline que importan

Además de StartSel/StopSel, tres opciones de `ts_headline` cambian visiblemente la experiencia. `MaxFragments` genera varios fragmentos separados cuando las coincidencias están repartidas por el documento (útil en textos largos); `FragmentDelimiter` define el separador entre ellos; y `ShortWord` controla el recorte en los bordes. Un ajuste que funciona bien para resultados tipo documentación:

```
ts_headline('es_sin_acentos', body, q,
  'StartSel=<mark>, StopSel=</mark>,
   MaxFragments=2, FragmentDelimiter=" … ",
   MaxWords=25, MinWords=10')
```

Y el recordatorio de coste, porque es el error que más veces he visto en revisiones: `ts_headline` procesa el documento original completo — no el tsvector — por cada fila donde se aplica. Su sitio es exclusivamente el SELECT final sobre la página ya limitada. Si tu página son 20 resultados, pagas 20 headlines; si lo pones antes del LIMIT, pagas uno por cada coincidencia de la tabla. En páginas muy calientes, cachear los snippets por (documento, consulta normalizada) con un TTL corto en Redis elimina también ese coste residual.

Operar el buscador: mantenimiento programado y señales de alarma

Un buscador en Postgres no exige un equipo de plataforma, pero sí tres hábitos operativos. Primero, una revisión semanal del search_log: cero resultados, latencia p95 y las consultas nuevas más frecuentes — quince minutos que dirigen todas las demás mejoras. Segundo, una mirada mensual al tamaño de los índices con `pg_relation_size`: un GIN que crece mucho más rápido que su tabla acumula bloat por updates y agradece un `REINDEX INDEX CONCURRENTLY` en horario tranquilo. Tercero, alertas sobre dos señales que anticipan problemas: la latencia p95 del endpoint (umbral típico: 3× tu línea base) y la tasa de errores por `statement_timeout`, que si sube de forma

sostenida indica que alguna clase nueva de consulta se volvió cara — casi siempre tras un cambio de contenido o de pesos.

Como resumen operativo de toda la edición ampliada, la checklist extendida:

- Configuración propia (`es_sin_acentos`) con unaccent en el pipeline, idéntica en índice y consultas.
- Autocompletar con `to_tsquery` + `:*` sobre input saneado, mínimo 3 caracteres, debounce en cliente.
- Sinónimos en tabla de aplicación (o diccionario synonym si controlas el servidor), alimentados por el log de cero resultados.
- `ts_rank_cd` con normalización 1 o 32, combinado con popularidad logarítmica y decaimiento temporal explícito.
- Multiidioma: configuración por fila con CASE, índices parciales por idioma, y el idioma siempre presente en el WHERE.
- Typos: fallback con `word_similarity` y umbral 0.4 cuando el full-text devuelve menos de 3 resultados.
- Paginación keyset con desempate por id; contador acotado en lugar de count exacto.
- Tablas existentes: columna por trigger + backfill por lotes + `CREATE INDEX CONCURRENTLY` — nunca una columna generada con reescritura.
- GIN: vigilar pending list y autovacuum; `fastupdate = off` si la carga es mayormente de lectura; `gin_clean_pending_list` tras cargas masivas.
- Híbrida: pgvector + RRF con k=60 y FULL OUTER JOIN; embedding de consulta cacheado.
- Multi-tenant: BitmapAnd verificado o `btree_gin`; particionado solo cuando los números lo pidan.
- Defensa: límites de longitud y términos, `SET LOCAL statement_timeout`, rate limiting en el gateway.
- Telemetría: search_log con cero-resultados y CTR; eval de recall@10 en CI antes de cada cambio de ranking.

Con esto, la frase inicial de la guía se sostiene con todas sus consecuencias: para la gran mayoría de aplicaciones, Postgres no es el buscador «de mientras tanto» — es el buscador, punto. Y el día que deje de serlo, lo sabrás por tus métricas, no por una corazonada.

Frases y proximidad: los operadores <-> y <N>

La búsqueda de frase que genera `websearch_to_tsquery` con comillas usa por debajo el operador de adyacencia `<->`: exige que el segundo lexema aparezca exactamente en la posición siguiente al primero. Su generalización `<N>` exige distancia exacta N, y combinándolos puedes construir consultas de proximidad que el full-text clásico no ofrece de serie:

```
-- "indice" seguido inmediatamente de "gin"
SELECT to_tsquery('spanish', 'indice <-> gin');

-- "indice" con exactamente dos posiciones de separacion: cubre
-- "indice parcial gin", "indice de gin", etc.
SELECT to_tsquery('spanish', 'indice <2> gin');

-- tsquery_phrase construye proximidad programaticamente:
SELECT tsquery_phrase(
  to_tsquery('spanish', 'indice'),
  to_tsquery('spanish', 'gin'),
```

```
  3  -- distancia EXACTA de 3
);
```

El matiz que confunde a todo el mundo: `<N>` es distancia *exacta*, no máxima. No existe un operador nativo «a menos de N palabras»; si lo necesitas, genera la disyunción `a <-> b | a <2> b | a <3> b` desde la aplicación. Y recuerda que las stopwords eliminadas siguen ocupando posición: en «índice de GIN», el «de» desaparece del tsvector pero deja hueco, así que la frase real es `indice <2> gin`, no `<->`. Este es el motivo por el que las búsquedas de frase con comillas a veces «no encuentran» textos que claramente contienen la frase — las posiciones no perdonan.

Depurar el pipeline con ts_debug

Cuando una búsqueda no encuentra lo que debería, adivinar es lento. `ts_debug` muestra exactamente qué hace el pipeline con cada token — qué diccionario lo capturó y qué lexema produjo:

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('es_sin_acentos', 'El índice GIN de PostgreSQL 18');
--  alias    | token      | dictionary   | lexemes
-- ----------+------------+--------------+----------
--  asciiword| El         | spanish_stem | {}        <- stopword: fuera
--  word     | índice     | spanish_stem | {indic}
--  asciiword| GIN        | spanish_stem | {gin}
--  asciiword| PostgreSQL | spanish_stem | {postgresql}
--  uint     | 18         | simple       | {18}
```

Tres diagnósticos que salen de esta tabla en segundos: una stopword inesperada (lexemes vacío), un stemming agresivo que fusiona palabras que tu dominio distingue, y tokens numéricos o con guiones que se parten de forma distinta a como el usuario los escribe. Ante cualquier informe de «no me sale tal resultado», `ts_debug` sobre el texto del documento y sobre la consulta, lado a lado, resuelve el misterio el 90 % de las veces.

El stemmer español en la práctica: dónde ayuda y dónde estorba

El diccionario `spanish_stem` es un stemmer Snowball: aplica reglas de sufijos sin entender significado. Con vocabulario general funciona muy bien — «corriendo», «corrió» y «correrá» convergen — pero tiene esquinas que conviene conocer en un buscador serio.

Los nombres propios y marcas sufren recortes indeseados: un stemmer feliz convierte «Cursor» y «cursores» en el mismo lexema, mezclando la herramienta con el concepto. El vocabulario técnico mixto español-inglés («deployar», «commitear») produce lexemas imprevisibles. Y las siglas cortas pueden colisionar con stopwords o palabras comunes.

La mitigación pragmática tiene dos piezas. Primera: el campo título — donde los nombres propios importan más — puede indexarse dos veces, una con stemming (peso A) y otra con el diccionario `simple` que solo pasa a minúsculas (peso B), de modo que la forma exacta siempre es encontrable:

```
search tsvector GENERATED ALWAYS AS (
  setweight(to_tsvector('es_sin_acentos', coalesce(title, '')), 'A') ||
  setweight(to_tsvector('simple',         coalesce(title, '')), 'B') ||
  setweight(to_tsvector('es_sin_acentos', coalesce(body,  '')), 'C')
) STORED
```

Segunda: mantén una lista corta de términos de dominio verificados con `ts_debug` en tu suite de tests, para que una futura actualización de diccionarios (por ejemplo, al migrar de versión mayor de Postgres) no cambie lexemas en silencio. Es un test de cinco líneas que ha salvado más de un buscador tras un upgrade.

Cuánto ocupa todo esto: almacenamiento y alternativas ligeras

La columna tsvector no es gratis: para texto editorial típico suele ocupar entre un 30 % y un 60 % del tamaño del texto original, y se almacena TOASTeada si supera el umbral de la fila. A eso se suma el índice GIN, que ronda entre la mitad y el tamaño completo del tsvector según la diversidad de vocabulario. En una tabla de contenido con 10 GB de texto, presupuestar 4–8 GB extra entre columna e índice es una estimación razonable — y medible desde el primer día:

```
SELECT
  pg_size_pretty(pg_total_relation_size('articles'))          AS tabla_total,
  pg_size_pretty(pg_relation_size('idx_articles_search'))     AS indice_gin,
  pg_size_pretty(sum(pg_column_size(search))) AS columna_tsvector
FROM articles;
```

Dos palancas si el espacio aprieta. La función `strip()` elimina posiciones y pesos del tsvector, reduciéndolo de forma notable — a cambio pierdes `ts_rank_cd`, los operadores de frase y los pesos por campo, así que solo tiene sentido para casos de filtrado puro sin ranking. Y para tablas pequeñas (menos de unas decenas de miles de filas) existe la alternativa de no materializar columna alguna: un índice de expresión directamente sobre `to_tsvector(...)`.

```
CREATE INDEX idx_articles_search_expr ON articles
  USING GIN (to_tsvector('es_sin_acentos', coalesce(title, '') || ' ' || coalesce(body,
'')));

-- La consulta debe repetir EXACTAMENTE la misma expresion
-- para que el planificador use el indice:
SELECT id FROM articles
WHERE to_tsvector('es_sin_acentos', coalesce(title, '') || ' ' || coalesce(body, ''))
      @@ websearch_to_tsquery('es_sin_acentos', 'postgres');
```

El índice de expresión ahorra la columna pero tiene dos peajes: la expresión de la consulta debe coincidir carácter a carácter con la indexada (una fuente clásica de «el índice no se usa y no sé por qué»), y no puedes aplicar pesos por campo. Para un proyecto que empieza, la columna generada sigue siendo la recomendación por defecto; el índice de expresión es el atajo válido para añadir búsqueda a una tabla auxiliar sin ceremonia.

Escalar lecturas: réplicas, estadísticas y el último kilómetro

Cuando el buscador se vuelve popular, la primera palanca de escala no es un motor nuevo: es servir las búsquedas desde réplicas de lectura. El full-text funciona en réplicas sin ninguna consideración especial — el índice GIN se replica como cualquier otro — con las precauciones estándar de lectura eventual: un documento recién publicado tarda el lag de replicación en aparecer en resultados. Para contenido editorial, un lag de uno o dos segundos es invisible; si tu producto promete «publicado y buscable al instante», enruta las búsquedas del autor del documento al primario durante unos segundos y las del resto del mundo a las réplicas.

Segunda palanca silenciosa: las estadísticas del planificador. El costeo de las consultas `@@` depende de las estadísticas de la columna tsvector, y una tabla que crece rápido con un `ANALYZE` atrasado lleva al planificador a decisiones malas

— típicamente elegir el índice equivocado en consultas con varios filtros. Si observas planes inestables tras cargas grandes, un `ANALYZE articles;` explícito al final del lote estabiliza el comportamiento, y subir el objetivo de estadísticas para la columna ayuda en tablas con vocabulario muy diverso.

```
-- Estadisticas mas ricas solo para la columna de busqueda
ALTER TABLE articles ALTER COLUMN search SET STATISTICS 500;
ANALYZE articles;
```

Y un último detalle fino: las configuraciones de búsqueda no dependen de la collation de la base, así que no hay sorpresas al migrar entre locales o a ICU — pero los índices B-tree auxiliares (el de `tenant_id`, los de ordenación) sí. Si en una migración de versión mayor cambia la versión de ICU o de glibc, esos B-tree pueden requerir `REINDEX`; el GIN del tsvector queda al margen. Saber qué índice depende de qué es la diferencia entre una migración tranquila y una noche larga.

Próximos pasos

Si vienes de cero, el orden de implementación con mejor retorno es: configuración con unaccent, columna con pesos e índice GIN (una tarde), endpoint con websearch_to_tsquery, ranking y snippets (otra tarde), search_log y revisión semanal (una hora que se repite), fallback de typos con pg_trgm (una mañana), y a partir de ahí, sinónimos, híbrida y evals según lo que el log te pida. Cada paso funciona solo y ninguno te compromete con el siguiente. La documentación oficial de PostgreSQL sobre text search es excelente y cabe en una sesión de lectura; las páginas de pg_trgm y pgvector completan el mapa. Y si este recurso te ahorró la semana de prueba y error que a casi todos nos costó este tema, ya cumplió su trabajo.

English Version

Full-Text Search in PostgreSQL: tsvector, GIN, Ranking and pg_trgm without Elasticsearch

Why LIKE Is Not a Search Engine

The first instinct of almost every team is to solve search with `ILIKE '%term%'`. It works in the demo and dies in production: a pattern with a leading wildcard cannot use a B-tree index, so every search scans the entire table. And even if performance didn't hurt, quality would: ILIKE doesn't know that "running" and "run" are the same word, doesn't rank by relevance, and treats a single missing accent as total failure.

PostgreSQL ships with a full-text search engine that solves all three problems: it normalizes words (stemming), indexes them with an inverted structure (GIN), and ranks by relevance. For most applications — up to tens of millions of rows — it's all you need, with no data to sync into a second system.

How It Works: tsvector, tsquery, and Dictionaries

The pipeline has two pieces. A `tsvector` is the processed document: tokenized text, stopwords removed, and each word reduced to its lexeme. A `tsquery` is the query processed the same way. The `@@` operator checks whether the vector satisfies the query.

```
SELECT to_tsvector('english', 'The cats were running through the garden');
-- 'cat':2 'garden':7 'run':4

SELECT to_tsvector('english', 'the cat runs') @@ to_tsquery('english', 'cats');
-- true: "cat" and "cats" share a lexeme
```

Dictionary choice matters: the same text processed with `'english'` and with `'spanish'` produces different lexemes. The golden rule is to always use the same configuration when indexing and when querying, passing it explicitly instead of relying on `default_text_search_config`.

The Right Schema: a Generated Column with Weights

The recommended practice since PostgreSQL 12 is a STORED generated column: it maintains itself on every INSERT and UPDATE, never drifts out of sync, and with `setweight` you can make the title matter more than the body.

```
CREATE TABLE articles (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  title text NOT NULL,
  body text NOT NULL,
  tags text[] NOT NULL DEFAULT '{}',
  search tsvector GENERATED ALWAYS AS (
    setweight(to_tsvector('english', coalesce(title, '')), 'A') ||
    setweight(to_tsvector('english', coalesce(body, '')), 'B') ||
    setweight(to_tsvector('english',
      coalesce(array_to_string(tags, ' '), '')), 'C')
  ) STORED
);
```

Those `coalesce` calls are not decorative: if any column were NULL, the entire tsvector would be NULL and that row would silently vanish from search with no visible error.

Indexing with GIN

```
CREATE INDEX idx_articles_search ON articles USING GIN (search);
```

GIN is an inverted index: it maps each lexeme to the rows containing it, exactly like the alphabetical index at the back of a book. It's the type the official documentation recommends for text search; GiST only makes sense in very specific cases with extremely frequent updates and queries that can tolerate being slower.

User Queries with websearch_to_tsquery

Never pass user input directly to `to_tsquery`: its operator syntax (&, |, !) throws errors on any malformed input. `websearch_to_tsquery` (PostgreSQL 11+) accepts web-search syntax — quotes for phrases, OR, a dash to exclude — and never raises an exception, which makes it the right choice for any public search box.

```
SELECT id, title
FROM articles,
     websearch_to_tsquery('english',
       'postgres "full text search" -elastic') AS q
WHERE search @@ q;
```

Ranking and Snippets: ts_rank and ts_headline

`ts_rank` scores each match based on lexeme frequency and the weights you defined (A > B > C > D). `ts_headline` generates the highlighted fragment shown under each result. One detail that separates a fast search from a slow one: `ts_headline` re-processes the entire original document, so it must only be applied to the final page of results, never before the LIMIT.

```
WITH hits AS (
  SELECT id, title, body, ts_rank(search, q) AS rank
  FROM articles,
       websearch_to_tsquery('english', 'gin indexes') AS q
  WHERE search @@ q
  ORDER BY rank DESC
  LIMIT 20
)
SELECT id, title, rank,
       ts_headline('english', body,
         websearch_to_tsquery('english', 'gin indexes'),
         'StartSel=<mark>, StopSel=</mark>, MaxWords=35, MinWords=15') AS snippet
FROM hits;
```

Accents and Typos: unaccent and pg_trgm

Two very real problems with user-facing search: people type accented words without the accent ("jardin" instead of "jardín") and they make typos. For the first there's `unaccent`; since the function isn't marked immutable, using it in a generated column requires an IMMUTABLE wrapper:

```
CREATE EXTENSION IF NOT EXISTS unaccent;

CREATE OR REPLACE FUNCTION immutable_unaccent(text)
```

```
RETURNS text LANGUAGE sql IMMUTABLE PARALLEL SAFE STRICT
AS $$ SELECT public.unaccent('public.unaccent', $1) $$;

-- In the generated column (and when building the query too):
-- setweight(to_tsvector('english',
--   immutable_unaccent(coalesce(title, ''))), 'A') || ...
```

For typos, full-text search doesn't help: "postgress" doesn't produce the lexeme for "postgres". That's where `pg_trgm` comes in — it compares character trigrams and returns fuzzy similarity. A pattern that works well is using it as a fallback when the exact search returns few results:

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;

CREATE INDEX idx_articles_title_trgm
  ON articles USING GIN (title gin_trgm_ops);

SELECT id, title, similarity(title, 'postgress') AS sim
FROM articles
WHERE title % 'postgress'  -- similarity operator
ORDER BY sim DESC
LIMIT 5;
```

A Complete Search Endpoint

Here's everything together in a real service with FastAPI and asyncpg: a safe query, ranking, snippets computed only over the returned page, and a defensive limit.

```
from contextlib import asynccontextmanager

import asyncpg
from fastapi import FastAPI, Query

SEARCH_SQL = """
WITH hits AS (
  SELECT id, title, body, ts_rank(search, q) AS rank
  FROM articles,
       websearch_to_tsquery('english', $1) AS q
  WHERE search @@ q
  ORDER BY rank DESC
  LIMIT $2
)
SELECT id, title, rank,
       ts_headline('english', body,
         websearch_to_tsquery('english', $1),
         'StartSel=<mark>, StopSel=</mark>, MaxWords=35') AS snippet
FROM hits
"""

@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.pool = await asyncpg.create_pool(dsn="postgresql://app@db/app")
    yield
    await app.state.pool.close()

app = FastAPI(lifespan=lifespan)
```

```
@app.get("/search")
async def search(
    q: str = Query(min_length=2, max_length=100),
    limit: int = Query(default=20, le=50),
):
    async with app.state.pool.acquire() as conn:
        rows = await conn.fetch(SEARCH_SQL, q, limit)
    return [dict(r) for r in rows]
```

Common Mistakes That Ruin Your Search

Mixing configurations. Indexing with `'english'` and querying with the default configuration produces different lexemes: zero results and no error. It's the number-one bug in Postgres-based search.

ts_headline before the LIMIT. It re-processes the whole document for every row; applied to thousands of matches it turns a millisecond query into a multi-second one.

Forgetting coalesce in the generated column. A single NULL field nullifies the entire tsvector and the row becomes invisible to search.

Ranking millions of matches. The GIN index finds the rows, but ts_rank is computed per matching row and the ordering can't come from the index. For very common terms, blend the rank with a precomputed popularity signal or require more specific terms.

Ignoring GIN's pending list. With fastupdate (on by default) writes go to a pending list that gets merged later; if it grows too large you'll see latency spikes. Watch autovacuum and tune gin_pending_list_limit if needed.

Huge documents. A tsvector can't exceed 1 MB and positions are capped at 16383; for very large texts, index a summary or split the document.

When Do You Actually Need a Dedicated Engine?

The practical line is this: Postgres wins while search is a feature; a dedicated engine wins when search is the product. Elasticsearch or OpenSearch bring BM25 relevance, synonyms, built-in spell correction, large-scale faceting, and horizontal scaling of the index. The price: operating another system and solving data synchronization (CDC, reindexing, eventual consistency). A recent middle ground is ParadeDB with pg_search, which brings BM25 ranking into Postgres itself. The honest recommendation: start in Postgres, measure result quality against real queries, and migrate only when you have evidence you've outgrown it.

Production Checklist

- Explicit language configuration, identical in index and queries.
- STORED generated column with coalesce and per-field weights.
- GIN index on the tsvector column.
- websearch_to_tsquery for all user input, with a validated maximum length.
- ts_headline only on the final page of results.
- unaccent (with an IMMUTABLE wrapper) if your content has accented characters; pg_trgm as a typo fallback.
- EXPLAIN ANALYZE on real queries and p95 latency metrics.
- A dataset of real user queries to measure relevance before tuning weights.

Expanded Edition: From a Working Search to a Production Search

Everything above gives you a correct search feature. This expanded edition covers what comes next: autocomplete, synonyms, ranking tuned with business signals, multilingual content, migrations on large tables, how GIN behaves under constant write load, hybrid search with embeddings, and how to measure whether your results are actually good. Each section stands alone: apply them in whatever order your product demands.

The Four tsquery Functions and When to Use Each

PostgreSQL ships four ways to turn user text into a `tsquery`, and choosing the wrong one is the second most common cause of disappointing search (the first is still mixing language configurations).

to_tsquery accepts the full operator syntax: `&` (AND), `|` (OR), `!` (NOT), `<->` (followed-by), and parentheses for grouping. It is the most powerful and the most dangerous: any malformed input throws an exception. Reserve it for queries your code builds — never for raw user input.

plainto_tsquery ignores all syntax and joins every word with AND. Safe but rigid: users cannot exclude terms or search for phrases.

phraseto_tsquery also ignores operators, but connects terms with `<->`, requiring them to appear consecutively and in order. It is the foundation of exact-phrase search.

websearch_to_tsquery combines the best of all: quotes for phrases, `OR`, a dash to exclude, and it never raises an exception. For a public search box it is the right choice 95% of the time.

```
SELECT to_tsquery('english', 'cat & !dog');           -- full syntax, can fail
SELECT plainto_tsquery('english', 'cat dog');          -- 'cat' & 'dog'
SELECT phraseto_tsquery('english', 'black cat');       -- 'black' <-> 'cat'
SELECT websearch_to_tsquery('english', '"black cat" OR siamese -persian');
-- 'black' <-> 'cat' | 'siames' & !'persian'
```

Autocomplete: Prefixes with :* and Incremental Search

A modern search box suggests results as the user types. Postgres full-text supports this with the prefix modifier `:*`, which matches any lexeme starting with the given string. The key detail: `websearch_to_tsquery` does not understand `:*` (it treats it as literal text), so the autocomplete query must be built with `to_tsquery` from previously sanitized input.

```
-- Pattern: all complete words ANDed, the last one as a prefix
-- User input: "gin ind"
SELECT id, title
FROM articles,
     to_tsquery('english', 'gin & ind:*') AS q
WHERE search @@ q
ORDER BY ts_rank(search, q) DESC
LIMIT 8;
```

On the backend, sanitize before building: strip everything that is not a letter, digit, or space, split on whitespace, and join with `&`, appending `:*` to the last term.

```
import re
```

```
def autocomplete_query(raw: str) -> str:
    """Turn free-form input into a safe prefix tsquery."""
    words = re.sub(r"[^\w\s]", " ", raw, flags=re.UNICODE).split()
    if not words:
        return ""
    *complete, last = words
    parts = [w for w in complete] + [last + ":*"]
    return " & ".join(parts)

# autocomplete_query("gin ind") -> "gin & ind:*"
```

Two production tips: debounce on the client for 150–250 ms so you do not fire a query per keystroke, and require a minimum of 2–3 characters before applying the prefix — a single-letter `:*` matches half the table and punishes the index. If you also want typo tolerance in autocomplete, combine this pattern with the `pg_trgm` fallback expanded below.

Custom Configurations: unaccent Built Into the Pipeline

The `immutable_unaccent` wrapper from part one works, but it forces you to remember to apply it in the column and in every query. The cleaner solution is a custom text search configuration that incorporates `unaccent` as a pipeline filter: define it once and from then on you index and query with it like any other configuration.

```
CREATE EXTENSION IF NOT EXISTS unaccent;

CREATE TEXT SEARCH CONFIGURATION en_unaccent (COPY = english);

ALTER TEXT SEARCH CONFIGURATION en_unaccent
  ALTER MAPPING FOR hword, hword_part, word
  WITH unaccent, english_stem;

-- The pipeline now strips accents before stemming:
SELECT to_tsvector('en_unaccent', 'The café in the garden');
-- 'cafe':2 'garden':5

SELECT to_tsvector('en_unaccent', 'café')
       @@ websearch_to_tsquery('en_unaccent', 'cafe');
-- true: with and without the accent converge to the same lexeme
```

With this in place, the generated column uses `'en_unaccent'` directly and no wrapper is needed. The usual rule still holds: the same configuration in the index and in every query.

Synonyms: When "invoice" and "receipt" Must Find Each Other

No stemmer knows that in your domain "pgsql", "postgres", and "postgresql" are the same thing. That is what synonym dictionaries are for: a file of pairs in the server's `tsearch_data` directory, inserted into the pipeline before the stemmer.

```
-- File $SHAREDIR/tsearch_data/my_synonyms.syn:
-- pgsql postgres
-- postgresql postgres
-- invoice receipt

CREATE TEXT SEARCH DICTIONARY synonyms_en (
```

```
  TEMPLATE = synonym,
  SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION en_unaccent
  ALTER MAPPING FOR word, hword, hword_part
  WITH unaccent, synonyms_en, english_stem;
```

The honest warning: this requires writing a file on the database server's filesystem, which most managed services (RDS, Cloud SQL, Supabase) do not allow. In those environments the practical alternative is expanding synonyms in the application layer: keep a dictionary in a table and rewrite the query, adding alternatives with `OR` before passing it to `websearch_to_tsquery`. Less elegant, but it works on any hosting and you can edit it without touching the server.

Ranking in Depth: ts_rank, ts_rank_cd, and Normalization

So far we used `ts_rank` with its defaults, which scores only by lexeme frequency and weights. There are two more levers that change results visibly.

The first is `ts_rank_cd` (cover density): besides frequency, it rewards query terms appearing *close to each other*. For multi-word queries it usually ranks better: a document where "index" and "GIN" appear in the same sentence beats one where they sit twenty paragraphs apart. It requires the tsvector to keep its positions (do not use `strip()` if you want it).

The second is the normalization parameter, a bitmask that corrects the bias toward long documents. The values that matter in practice:

- `0` (default): no normalization; long documents accumulate more matches and rise artificially.
- `1`: divides by 1 + log(document length) — the gentle correction that works well almost always.
- `2`: divides by the full length; punishes long documents hard.
- `32`: compresses the result to rank/(rank+1), useful when you need a 0–1 scale to combine with other signals.

```
SELECT id, title,
       ts_rank_cd(search, q, 1) AS rank  -- cover density + log normalization
FROM articles,
     websearch_to_tsquery('en_unaccent', 'gin indexes') AS q
WHERE search @@ q
ORDER BY rank DESC
LIMIT 20;
```

Blending Relevance with Business Signals

Pure text ranking is rarely the ranking your product needs. An old, popular article may deserve more visibility than a fresh one with no reads — or the other way around. The pattern that scales well: normalize the text rank with `32`, then combine it additively with precomputed signals — log-dampened popularity and an explicit time decay.

```
SELECT id, title,
       ts_rank_cd(search, q, 32) AS text_rank,
       (0.6 * ts_rank_cd(search, q, 32)
     + 0.3 * (ln(1 + view_count) / ln(1 + max_views.mx))
     + 0.1 * exp(-extract(epoch FROM now() - published_at)
```

```
                  / 86400.0 / 90)) AS score
FROM articles,
     websearch_to_tsquery('en_unaccent', 'postgres') AS q,
     (SELECT max(view_count) AS mx FROM articles) AS max_views
WHERE search @@ q
ORDER BY score DESC
LIMIT 20;
```

The weights (0.6 / 0.3 / 0.1) are not sacred: they are the starting point you will tune by measuring against real queries, as we will see in the evaluation section. What matters is the structure — text relevance as the dominant signal, popularity dampened with a logarithm so one viral article cannot monopolize every result, and recency with an explicit half-life (90 days here).

Multilingual Content in a Single Table

If your application serves content in Spanish and English, the temptation is to index everything with one configuration. Don't: the Spanish stemmer leaves "running" as "running" and the English one leaves "corriendo" as "corriendo" — each mis-processed language loses stemming entirely. The right pattern is a language column and a generated tsvector that picks the configuration per row.

```
CREATE TABLE posts (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  lang text NOT NULL CHECK (lang IN ('es', 'en')),
  title text NOT NULL,
  body text NOT NULL,
  search tsvector GENERATED ALWAYS AS (
    CASE lang
      WHEN 'es' THEN
        setweight(to_tsvector('es_unaccent', coalesce(title, '')), 'A') ||
        setweight(to_tsvector('es_unaccent', coalesce(body,  '')), 'B')
      ELSE
        setweight(to_tsvector('english', coalesce(title, '')), 'A') ||
        setweight(to_tsvector('english', coalesce(body,  '')), 'B')
    END
  ) STORED
);
```

For querying there are two indexing options. A single GIN index on `search` works; if the language split is heavily skewed or you want smaller indexes, create partial indexes per language:

```
CREATE INDEX idx_posts_search_es ON posts USING GIN (search) WHERE lang = 'es';
CREATE INDEX idx_posts_search_en ON posts USING GIN (search) WHERE lang = 'en';

-- The query ALWAYS filters by language and uses that language's config:
SELECT id, title
FROM posts,
     websearch_to_tsquery('english', 'indexes') AS q
WHERE lang = 'en' AND search @@ q;
```

The discipline this demands: the interface language decides the query configuration, and the two always travel together. A user searching in English against Spanish content will get poor stemming — an honest limitation of this approach, and the reason hybrid search with embeddings (below) is such a good complement: multilingual embeddings cross languages effortlessly.

pg_trgm in Depth: Similarity, Thresholds, and Accelerating ILIKE

Part one used `pg_trgm` as a typo fallback. The extension offers quite a bit more, and understanding its three similarity functions avoids frustrating false positives.

similarity(a, b) compares the complete trigram sets of both strings. It works well between strings of similar length (titles, names), but punishes comparing one word against a long text.

word_similarity(a, b) finds the best match of `a` against *any part* of `b`. It is the right one when the user types a word and you search it inside full titles. Its operator is `<%`.

strict_word_similarity(a, b) does the same but respects word boundaries — fewer false positives with short words.

```
SET pg_trgm.similarity_threshold = 0.3;       -- threshold for the % operator
SET pg_trgm.word_similarity_threshold = 0.4;  -- threshold for the <% operator

-- Typo against titles: word_similarity is the right tool
SELECT id, title, word_similarity('postgress', title) AS sim
FROM articles
WHERE 'postgress' <% title
ORDER BY sim DESC
LIMIT 5;
```

On indexes: `gin_trgm_ops` accelerates the similarity operators and — a gift many people miss — also `LIKE` and `ILIKE` searches with a leading wildcard, the very ones we said at the start could not use indexes. With a trigram index, `ILIKE '%invoice%'` goes from sequential scan to indexed lookup:

```
CREATE INDEX idx_articles_title_trgm
  ON articles USING GIN (title gin_trgm_ops);

EXPLAIN (ANALYZE, BUFFERS)
SELECT id FROM articles WHERE title ILIKE '%invoice%';
-- Bitmap Index Scan on idx_articles_title_trgm  ✓
```

The GiST variant (`gist_trgm_ops`) is slower to read but supports the distance operator `<->` with index-ordered results (KNN), useful for "the 5 most similar" without computing similarity over the whole table. For the typical typo-fallback case, GIN is the default choice. Two cautions: trigram indexes grow fast (they can exceed the size of the indexed column), and strings under 3 characters produce no useful trigrams — validate a minimum length before delegating to them.

Paginating Results Without OFFSET

Page 1 of a search is fast; page 50 with `OFFSET 980` forces ranking 1,000 rows to discard 980. With common terms, each page is slower than the last. The fix is the same keyset pagination you would use in any listing, with one nuance: rank has ties, so the cursor needs a stable tiebreaker.

```
-- Page 1
SELECT id, title, ts_rank_cd(search, q, 32) AS rank
FROM articles, websearch_to_tsquery('en_unaccent', 'postgres') AS q
WHERE search @@ q
ORDER BY rank DESC, id DESC
LIMIT 20;
```

```
-- Next page: the client sends back (last_rank, last_id)
SELECT id, title, ts_rank_cd(search, q, 32) AS rank
FROM articles, websearch_to_tsquery('en_unaccent', 'postgres') AS q
WHERE search @@ q
  AND (ts_rank_cd(search, q, 32), id) < ($1, $2)
ORDER BY rank DESC, id DESC
LIMIT 20;
```

The tuple comparison `(rank, id) < ($1, $2)` handles rank ties correctly. Encode the cursor as an opaque base64 token in your API, exactly as you would for any keyset pagination.

That leaves the result counter. An exact `count(*)` on a common term walks every match and can cost more than the search itself. Two honest alternatives: a capped counter ("1,000+ results") that stops at a fixed limit, or the planner's estimate for queries where precision does not matter.

```
-- Capped counter: never counts more than 1,001 rows
SELECT count(*) FROM (
  SELECT 1
  FROM articles, websearch_to_tsquery('en_unaccent', 'postgres') AS q
  WHERE search @@ q
  LIMIT 1001
) t;
-- If it returns 1001, display "1,000+ results"
```

Migrating a Large Table Without Downtime

Everything so far assumes you can create the table from scratch. The usual reality is a production table with millions of rows that you want to add search to. There is a serious trap here: `ALTER TABLE ... ADD COLUMN ... GENERATED ALWAYS AS ... STORED` rewrites the entire table under an `ACCESS EXCLUSIVE` lock — on a large table, that is an outage.

The zero-downtime route uses a regular column maintained by a trigger, a batched backfill, and a concurrent index:

```
-- 1. Regular column, instantaneous (catalog-only change)
ALTER TABLE articles ADD COLUMN search tsvector;

-- 2. Trigger for new and updated rows
CREATE OR REPLACE FUNCTION articles_search_update() RETURNS trigger AS $$
BEGIN
  NEW.search :=
    setweight(to_tsvector('en_unaccent', coalesce(NEW.title, '')), 'A') ||
    setweight(to_tsvector('en_unaccent', coalesce(NEW.body,  '')), 'B');
  RETURN NEW;
END $$ LANGUAGE plpgsql;

CREATE TRIGGER trg_articles_search
  BEFORE INSERT OR UPDATE OF title, body ON articles
  FOR EACH ROW EXECUTE FUNCTION articles_search_update();
```

```
# 3. Batched backfill from the application (never one massive UPDATE)
import asyncpg, asyncio

BATCH = """
UPDATE articles SET search =
```

```
  setweight(to_tsvector('en_unaccent', coalesce(title, '')), 'A') ||
  setweight(to_tsvector('en_unaccent', coalesce(body,  '')), 'B')
WHERE id IN (
  SELECT id FROM articles
  WHERE search IS NULL
  ORDER BY id
  LIMIT 5000
  FOR UPDATE SKIP LOCKED
)
"""

async def backfill(dsn: str):
    conn = await asyncpg.connect(dsn)
    while True:
        result = await conn.execute(BATCH)
        if result == "UPDATE 0":
            break
        await asyncio.sleep(0.5)  # breathing room for autovacuum and replicas
    await conn.close()
```

```
-- 4. Index without blocking writes
CREATE INDEX CONCURRENTLY idx_articles_search ON articles USING GIN (search);
```

Batches of 5,000 with a pause avoid inflating the WAL all at once and give autovacuum air; `FOR UPDATE SKIP LOCKED` lets you run several backfill workers without them stepping on each other. When the backfill finishes and the index is built, your search endpoint can go live behind a feature flag. The trigger-maintained column is a perfectly valid permanent solution — the generated column is cleaner, but it does not justify a locked rewrite on a table already serving production traffic.

Inside GIN: the Pending List and the Real Cost of Writing

To operate a search feature under constant write load it helps to understand what GIN does with every INSERT. An average document produces dozens or hundreds of lexemes, and updating each one's entry in the inverted index for every inserted row would be brutally expensive. That is why GIN has `fastupdate` (on by default): new rows go into an unstructured *pending list* that gets merged into the main index later — during autovacuum, or when the list exceeds `gin_pending_list_limit` (4 MB by default).

The visible effect: searches must also scan the pending list linearly, so if it grows too large you get seemingly random latency spikes. And whichever process (backend or autovacuum) gets to merge the list pays a write spike. The levers:

```
-- Inspect the current pending list of an index
CREATE EXTENSION IF NOT EXISTS pgstattuple;
SELECT * FROM gin_metapage_info(get_raw_page('idx_articles_search', 0));

-- Merge manually (for example, after a bulk load)
SELECT gin_clean_pending_list('idx_articles_search');

-- Tune the threshold per index
ALTER INDEX idx_articles_search SET (gin_pending_list_limit = 1024); -- KB

-- Or disable fastupdate: slower but uniform writes,
-- always-predictable reads. Reasonable with light write load.
ALTER INDEX idx_articles_search SET (fastupdate = off);
```

The heuristic: if your workload is read-dominant with sporadic writes, `fastupdate = off` removes the variability. If you ingest content in large batches, keep fastupdate on and call `gin_clean_pending_list` after each batch. In both cases, make sure autovacuum visits the table often — a starved autovacuum is the number-one cause of giant pending lists.

To read a search plan, the sane way is `EXPLAIN (ANALYZE, BUFFERS)`: you will see a `Bitmap Index Scan` on the GIN index followed by a `Bitmap Heap Scan` with a `Recheck Cond` line. The recheck is normal (GIN can produce internal false positives that get verified against the actual row); what should alarm you is a `Seq Scan` where you expected the index — almost always it means the query expression does not exactly match the indexed expression, or you mixed configurations.

Hybrid Search: Full-Text + pgvector with Reciprocal Rank Fusion

Full-text finds what contains the exact words; embeddings find what *means* the same even when it shares no words ("how to speed up queries" ↔ "SQL performance optimization"). In 2026 the standard practice is combining both — hybrid search — and Postgres does it in a single query with `pgvector`. Public RAG benchmarks show retrieval-precision gains on the order of 20 points when the lexical signal is added to the vector one, with exact-match queries (names, error codes, identifiers) benefiting the most.

```
CREATE EXTENSION IF NOT EXISTS vector;

ALTER TABLE articles ADD COLUMN embedding vector(1536);

-- HNSW: the recommended vector index for fast queries
CREATE INDEX idx_articles_embedding ON articles
  USING hnsw (embedding vector_cosine_ops);
```

The fusion uses Reciprocal Rank Fusion (RRF), which never needs to normalize scores from two incomparable systems: each result contributes `1 / (k + position)` in each list and the contributions are summed. A document ranked well in both lists dominates; one present in only one list still gets a voice. The constant `k = 60` is the standard from the literature.

```
WITH fts AS (
  SELECT id, row_number() OVER (ORDER BY ts_rank_cd(search, q, 32) DESC) AS r
  FROM articles, websearch_to_tsquery('en_unaccent', $1) AS q
  WHERE search @@ q
  LIMIT 40
),
sem AS (
  SELECT id, row_number() OVER (ORDER BY embedding <=> $2) AS r
  FROM articles
  ORDER BY embedding <=> $2
  LIMIT 40
)
SELECT a.id, a.title,
       coalesce(1.0 / (60 + fts.r), 0) +
       coalesce(1.0 / (60 + sem.r), 0) AS rrf_score
FROM fts FULL OUTER JOIN sem USING (id)
JOIN articles a USING (id)
ORDER BY rrf_score DESC
LIMIT 20;
```

On the backend the flow is: compute the query embedding (one call to your provider or a local model), then pass both the text and the vector as parameters. The added cost is that embedding call — cache it for repeated queries — and the HNSW index on disk. Two production nuances: the `FULL OUTER JOIN` is essential (a lexical-only or semantic-only result must survive), and each branch's `LIMIT 40` controls the quality/latency balance; raise them if your final page is large. If you later need true BM25 ranking inside Postgres, extensions like ParadeDB's `pg_search` provide it without leaving the database — same criterion as always: migrate when you measure that you need it, not before.

Observability: Log Every Search, Especially the Failing Ones

A search feature without telemetry is a search feature degrading in silence. The two metrics that give the most information per euro invested: the **zero-results rate** (what people searched for and did not find) and **CTR by position** (if nobody clicks the top three, your ranking is wrong). Both come from a cheap log table you write from the endpoint:

```
CREATE TABLE search_log (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  query text NOT NULL,
  results_count int NOT NULL,
  latency_ms int NOT NULL,
  clicked_id bigint,          -- filled in via click event
  clicked_position int,
  created_at timestamptz NOT NULL DEFAULT now()
);

-- The 20 most frequent zero-result searches of the week:
SELECT query, count(*) AS times
FROM search_log
WHERE results_count = 0
  AND created_at > now() - interval '7 days'
GROUP BY query
ORDER BY times DESC
LIMIT 20;
```

That zero-results query is pure gold: it surfaces the synonyms missing from your dictionary, the typos your pg_trgm fallback does not cover, and your users' real vocabulary that your content does not use. Reviewing it weekly and turning the patterns into synonyms or new content is the highest ROI search improvement there is.

On the infrastructure side, `pg_stat_statements` gives you mean and p95 latency of the actual search query, and index size is watched with `pg_relation_size`. If the index grows far beyond the content (GIN accumulates bloat under frequent updates), `REINDEX INDEX CONCURRENTLY` compacts it without downtime.

Measuring Relevance: a Minimal Eval That Fits in CI

Tuning ranking weights "by eye" is swapping one thing you do not measure for another. The antidote is small: a golden dataset of real queries with the documents that *should* appear, and one simple metric — recall@10: is the expected document among the first ten?

```
GOLDEN = [
    ("gin indexes", {12, 47}),          # ids that must appear
    ("search without accents", {103}),
    ("postgress typo", {12}),           # validates the trigram fallback
    ("how to speed up queries", {55, 12}),  # validates the semantic branch
]
```

```
async def test_recall_at_10(search_client):
    failures = []
    for query, expected in GOLDEN:
        ids = {r["id"] for r in await search_client.search(query, limit=10)}
        if not expected & ids:
            failures.append(query)
    # Require 90% hits to pass CI
    assert len(failures) <= len(GOLDEN) * 0.1, f"Queries with no hit: {failures}"
```

Thirty queries are enough to start; feed it with real searches from search_log. From then on, every change to weights, configuration, or dictionaries runs against the eval before deploying — exactly as you would with any other regression.

Case Study: the Search Behind a Knowledge Base

To land everything, the complete journey of a realistic case: an internal knowledge base with 800,000 articles in Spanish and English, 2,000 writes per day, and a peak of 50 searches per second.

Week 1 — the basics done right. `es_unaccent` + `english` configurations with a trigger-maintained column (the table already existed: no rewrites), an overnight batched backfill, `CREATE INDEX CONCURRENTLY`, an endpoint with `websearch_to_tsquery`, ranking with `ts_rank_cd(search, q, 1)`, and `ts_headline` only on the returned page. P95 latency: 38 ms. Search went from "unusable" to "correct" without touching the infrastructure.

Week 3 — the data decides. The search_log showed 14% of searches returning zero results. Half were typos (the pg_trgm fallback with `word_similarity` was enabled whenever full-text returned fewer than 3 results) and a third were vocabulary: people searched "vpn down" while articles said "private network connectivity". Twenty synonyms applied at the application layer brought the zero-results rate down to 5%.

Month 2 — hybrid where it pays. `pgvector` was added with multilingual embeddings and RRF. The biggest jump was not in typical searches — it was in long question-style queries ("why am I not getting the recovery email") and in English searches over Spanish content, which the embeddings cross effortlessly. The 60-query eval went from 78% to 91% recall@10. The real operating cost: 6 GB more of HNSW index and one embedding call per search, cached with a one-day TTL for repeated queries.

The cross-cutting lesson: every improvement was *measured* before being adopted, and none required a second system. The day this team needs complex large-scale faceting or built-in spell correction, they will migrate to a dedicated engine with an evaluation dataset already built — the best possible insurance policy for a migration.

Frequently Asked Questions

Do I need to reindex if I change the search configuration? Yes. The configuration is applied when building the tsvector, so changing the pipeline (adding unaccent, synonyms) requires regenerating the column and, with it, the index. With the trigger-maintained column: a batched UPDATE, same as the initial backfill.

Can I search inside JSONB? Yes: extract the text fields and concatenate them into the generated tsvector, or use `jsonb_to_tsvector`, which indexes the values of a whole JSON document with a filter by data type.

Full-text or trigram for people's names? Trigram. Names gain nothing from stemming and suffer typos and variants; `word_similarity` with a 0.4 threshold is usually the best balance.

How do I handle PDF or HTML documents? Extract plain text in the application before saving (stripping tags); indexing raw HTML injects tags as lexemes and ruins ranking.

Does word order matter in websearch_to_tsquery? No, except inside quotes (exact phrase). Loose terms are combined with AND regardless of order.

What about very short words or stopwords? The language's stopwords ("the", "of", "and") are removed in the pipeline and are not searchable. If your domain needs to find "IT" or "AI", consider a simple dictionary without stopwords for the title field.

How much RAM does this need? The practical rule: the hot GIN index should fit in shared_buffers + page cache. A GIN over a million average-sized documents runs 1–3 GB; measuring it with `pg_relation_size` is trivial and beats any estimate.

Is this good for logs or metrics? Not its strong suit. For logs, the Loki/Elasticsearch family with time-based retention fits better; Postgres full-text shines with editorial content: articles, products, tickets, documentation.

Multi-Tenant: Isolating Search per Customer

In a SaaS, every search carries a mandatory `WHERE tenant_id = $n`, and that interacts with GIN in an unintuitive way: GIN indexes the tsvector, not the tenant, so Postgres has to find every textual match first and filter the tenant afterward. With thousands of tenants and common terms, that is wasted work at scale. Three strategies, in order of effort:

1. Trust the combined bitmap. With a B-tree on `tenant_id` and the GIN on `search`, the planner can BitmapAnd both indexes. For most workloads this is enough — verify with `EXPLAIN (ANALYZE, BUFFERS)` that the BitmapAnd actually shows up.

2. A composite GIN index with btree_gin. The `btree_gin` extension lets scalar columns live inside a GIN index, so tenant and text resolve in a single index pass:

```
CREATE EXTENSION IF NOT EXISTS btree_gin;

CREATE INDEX idx_articles_tenant_search
  ON articles USING GIN (tenant_id, search);

-- This WHERE now resolves entirely inside the index:
SELECT id, title
FROM articles, websearch_to_tsquery('en_unaccent', $1) AS q
WHERE tenant_id = $2 AND search @@ q;
```

3. Partitioning by tenant (or by tenant hash). If besides filtering you need per-customer retention or size limits, partitioning turns each search into a scan of a single partition with its own small GIN. It is the most expensive option to operate; save it for when the numbers demand it.

If you use Row-Level Security, policies apply normally to the candidate rows the index returns — search opens no new hole. The cost is the same as point 1: RLS filtering happens after the index, so the BitmapAnd or btree_gin advice applies equally.

Defending the Endpoint: Limits, Timeouts, and Pathological Queries

A public search endpoint is a cheap denial-of-service surface: extremely long queries, dozens of terms, huge quoted phrases, or single-letter prefixes can cost orders of magnitude more than a normal search. Layered defenses, all cheap:

```
from fastapi import FastAPI, HTTPException, Query

MAX_TERMS = 8

@app.get("/search")
async def search(
    q: str = Query(min_length=2, max_length=100),
    limit: int = Query(default=20, le=50),
):
    if len(q.split()) > MAX_TERMS:
        raise HTTPException(422, "Too many search terms")
    async with app.state.pool.acquire() as conn:
        # Hard ceiling per query: nothing can hang longer than 300 ms
        await conn.execute("SET LOCAL statement_timeout = '300ms'")
        rows = await conn.fetch(SEARCH_SQL, q, limit)
    return [dict(r) for r in rows]
```

The `SET LOCAL statement_timeout` is the ultimate safety net: no pathological combination of terms can hold a pool connection longer than 300 ms — it returns a controlled error your API translates into "refine your search". Complement this with the per-user rate limiting you should already have at the gateway, and with a product rule: `:*` prefixes only from 3 characters on, as we saw in autocomplete.

An extra defense for extremely common terms is `gin_fuzzy_search_limit`: a soft limit (disabled by default) that caps how many results GIN returns per search. It is a tradeoff — it introduces randomness in which results appear — but for search boxes where "postgres" matches 40% of the table, a value of 5,000–10,000 keeps latency flat with no visible difference in the first pages.

Facets: Counting Results per Category Without a Second Query

Facets ("Guides (12) · Tutorials (5) · Cases (2)") resolve with the same search query and conditional aggregation, avoiding an extra round trip:

```
WITH hits AS (
  SELECT id, category, type
  FROM articles, websearch_to_tsquery('en_unaccent', $1) AS q
  WHERE search @@ q
)
SELECT
  (SELECT count(*) FROM hits) AS total,
  (SELECT jsonb_object_agg(category, n) FROM (
     SELECT category, count(*) AS n FROM hits GROUP BY category
  ) c) AS by_category,
  (SELECT jsonb_object_agg(type, n) FROM (
     SELECT type, count(*) AS n FROM hits GROUP BY type
  ) t) AS by_type;
```

The scale warning: this aggregation counts *all* matches, so it inherits the exact-count problem. With common terms on large tables, apply the same defensive cap (a `LIMIT 5000` inside the CTE) and display "5,000+" — nobody makes different decisions between 5,000 and 80,000 results in a facet.

Tuned Snippets: the ts_headline Options That Matter

Beyond StartSel/StopSel, three `ts_headline` options visibly change the experience. `MaxFragments` produces several separate fragments when matches are scattered across the document (useful in long texts); `FragmentDelimiter` defines the separator between them; and `ShortWord` controls trimming at the edges. A setting that works well for documentation-style results:

```
ts_headline('en_unaccent', body, q,
  'StartSel=<mark>, StopSel=</mark>,
   MaxFragments=2, FragmentDelimiter=" … ",
   MaxWords=25, MinWords=10')
```

And the cost reminder, because it is the mistake I have seen most often in reviews: `ts_headline` processes the full original document — not the tsvector — for every row it touches. Its only correct place is the final SELECT over the already-limited page. If your page is 20 results, you pay 20 headlines; put it before the LIMIT and you pay one per match in the table. On very hot pages, caching snippets by (document, normalized query) with a short TTL in Redis removes even that residual cost.

Operating the Search: Scheduled Maintenance and Warning Signs

A Postgres-based search does not demand a platform team, but it does demand three operational habits. First, a weekly review of the search_log: zero results, p95 latency, and the most frequent new queries — fifteen minutes that steer every other improvement. Second, a monthly look at index sizes with `pg_relation_size`: a GIN growing much faster than its table is accumulating update bloat and will appreciate a `REINDEX INDEX CONCURRENTLY` during a quiet window. Third, alerts on two leading indicators: the endpoint's p95 latency (typical threshold: 3× your baseline) and the rate of `statement_timeout` errors, which — if it climbs steadily — means some new class of query got expensive, almost always after a content or weight change.

As an operational summary of the whole expanded edition, the extended checklist:

- A custom configuration (`en_unaccent`) with unaccent in the pipeline, identical in index and queries.
- Autocomplete with `to_tsquery` + `:*` over sanitized input, 3-character minimum, client-side debounce.
- Synonyms in an application table (or a synonym dictionary if you control the server), fed by the zero-results log.
- `ts_rank_cd` with normalization 1 or 32, blended with log-dampened popularity and explicit time decay.
- Multilingual: per-row configuration with CASE, partial indexes per language, and the language always present in the WHERE.
- Typos: fallback with `word_similarity` and a 0.4 threshold when full-text returns fewer than 3 results.
- Keyset pagination with id tiebreaker; capped counter instead of an exact count.
- Existing tables: trigger-maintained column + batched backfill + `CREATE INDEX CONCURRENTLY` — never a generated column with a rewrite.
- GIN: watch the pending list and autovacuum; `fastupdate = off` for read-mostly loads; `gin_clean_pending_list` after bulk loads.

- Hybrid: pgvector + RRF with k=60 and a FULL OUTER JOIN; query embedding cached.
- Multi-tenant: verified BitmapAnd or `btree_gin`; partitioning only when the numbers demand it.
- Defense: length and term limits, `SET LOCAL statement_timeout`, gateway rate limiting.
- Telemetry: search_log with zero-results and CTR; recall@10 eval in CI before every ranking change.

With this, the guide's opening claim holds with all its consequences: for the vast majority of applications, Postgres is not the "meanwhile" search engine — it is the search engine, full stop. And the day it stops being enough, you will know from your metrics, not from a hunch.

Phrases and Proximity: the <-> and <N> Operators

The phrase search that `websearch_to_tsquery` generates from quotes uses the adjacency operator `<->` underneath: it requires the second lexeme to appear in exactly the position after the first. Its generalization `<N>` requires an exact distance of N, and combining them builds proximity queries that classic full-text does not offer out of the box:

```
-- "gin" immediately followed by "index"
SELECT to_tsquery('english', 'gin <-> index');

-- "gin" exactly two positions away: covers
-- "gin partial index", "gin like index", etc.
SELECT to_tsquery('english', 'gin <2> index');

-- tsquery_phrase builds proximity programmatically:
SELECT tsquery_phrase(
  to_tsquery('english', 'gin'),
  to_tsquery('english', 'index'),
  3  -- EXACT distance of 3
);
```

The nuance that confuses everyone: `<N>` is *exact* distance, not maximum. There is no native "within N words" operator; if you need it, generate the disjunction `a <-> b | a <2> b | a <3> b` from the application. And remember that removed stopwords still occupy a position: in "index of GIN", the "of" disappears from the tsvector but leaves a gap, so the real phrase is `index <2> gin`, not `<->`. This is why quoted phrase searches sometimes "fail to find" texts that clearly contain the phrase — positions are unforgiving.

Debugging the Pipeline with ts_debug

When a search does not find what it should, guessing is slow. `ts_debug` shows exactly what the pipeline does with each token — which dictionary captured it and which lexeme it produced:

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('en_unaccent', 'The GIN index in PostgreSQL 18');
--  alias    | token      | dictionary   | lexemes
-- ----------+------------+--------------+----------
--  asciiword| The        | english_stem | {}        <- stopword: gone
--  asciiword| GIN        | english_stem | {gin}
--  asciiword| index      | english_stem | {index}
--  asciiword| PostgreSQL | english_stem | {postgresql}
--  uint     | 18         | simple       | {18}
```

Three diagnoses fall out of this table in seconds: an unexpected stopword (empty lexemes), overly aggressive stemming that merges words your domain distinguishes, and numeric or hyphenated tokens splitting differently from how users type them. For any "result X does not show up" report, running `ts_debug` on the document text and on the query, side by side, solves the mystery 90% of the time.

The Stemmer in Practice: Where It Helps and Where It Hurts

The `english_stem` dictionary is a Snowball stemmer: it applies suffix rules without understanding meaning. With general vocabulary it works very well — "running", "ran", and "runs" converge — but it has corners worth knowing in a serious search feature.

Proper nouns and brand names suffer unwanted trimming: a happy stemmer merges "Cursor" the tool with "cursors" the concept. Mixed-language technical vocabulary produces unpredictable lexemes. And short acronyms can collide with stopwords or common words.

The pragmatic mitigation has two pieces. First: the title field — where proper nouns matter most — can be indexed twice, once with stemming (weight A) and once with the `simple` dictionary that only lowercases (weight B), so the exact form is always findable:

```
search tsvector GENERATED ALWAYS AS (
  setweight(to_tsvector('en_unaccent', coalesce(title, '')), 'A') ||
  setweight(to_tsvector('simple',      coalesce(title, '')), 'B') ||
  setweight(to_tsvector('en_unaccent', coalesce(body,  '')), 'C')
) STORED
```

Second: keep a short list of domain terms verified with `ts_debug` in your test suite, so a future dictionary update (say, on a major Postgres upgrade) cannot silently change lexemes. It is a five-line test that has saved more than one search feature after an upgrade.

How Much All This Weighs: Storage and Lightweight Alternatives

The tsvector column is not free: for typical editorial text it runs between 30% and 60% of the original text's size, and it is TOASTed when it exceeds the row threshold. Add the GIN index, which lands between half and the full size of the tsvector depending on vocabulary diversity. For a content table with 10 GB of text, budgeting an extra 4–8 GB between column and index is a reasonable estimate — and measurable from day one:

```
SELECT
  pg_size_pretty(pg_total_relation_size('articles'))          AS table_total,
  pg_size_pretty(pg_relation_size('idx_articles_search'))     AS gin_index,
  pg_size_pretty(sum(pg_column_size(search))) AS tsvector_column
FROM articles;
```

Two levers if space gets tight. The `strip()` function removes positions and weights from the tsvector, shrinking it notably — in exchange you lose `ts_rank_cd`, phrase operators, and per-field weights, so it only makes sense for pure filtering without ranking. And for small tables (under a few tens of thousands of rows) there is the alternative of not materializing a column at all: an expression index directly over `to_tsvector(...)`.

```
CREATE INDEX idx_articles_search_expr ON articles
  USING GIN (to_tsvector('en_unaccent', coalesce(title, '') || ' ' || coalesce(body,
'')));
```

```
-- The query must repeat EXACTLY the same expression
-- for the planner to use the index:
SELECT id FROM articles
WHERE to_tsvector('en_unaccent', coalesce(title, '') || ' ' || coalesce(body, ''))
      @@ websearch_to_tsquery('en_unaccent', 'postgres');
```

The expression index saves the column but has two tolls: the query expression must match the indexed one character for character (a classic source of "the index is not used and I do not know why"), and you cannot apply per-field weights. For a new project, the generated column remains the default recommendation; the expression index is the valid shortcut for adding search to an auxiliary table without ceremony.

Scaling Reads: Replicas, Statistics, and the Last Mile

When search gets popular, the first scaling lever is not a new engine: it is serving searches from read replicas. Full-text works on replicas with no special handling — the GIN index replicates like any other — with the standard eventual-read caveats: a freshly published document takes the replication lag to appear in results. For editorial content, a one-or-two-second lag is invisible; if your product promises "published and searchable instantly", route the author's own searches to the primary for a few seconds and everyone else's to the replicas.

The second silent lever: planner statistics. Costing of `@@` queries depends on the tsvector column's statistics, and a fast-growing table with a stale `ANALYZE` pushes the planner into bad decisions — typically picking the wrong index in queries with several filters. If you see unstable plans after big loads, an explicit `ANALYZE articles;` at the end of the batch stabilizes behavior, and raising the statistics target for the column helps on tables with very diverse vocabulary.

```
-- Richer statistics only for the search column
ALTER TABLE articles ALTER COLUMN search SET STATISTICS 500;
ANALYZE articles;
```

One final fine-grained detail: text search configurations do not depend on the database collation, so there are no surprises when migrating between locales or to ICU — but the auxiliary B-tree indexes (the one on `tenant_id`, the ordering ones) do. If a major-version migration changes the ICU or glibc version, those B-trees may need a `REINDEX`; the tsvector's GIN stays out of it. Knowing which index depends on what is the difference between a calm migration and a long night.

Next Steps

Starting from zero, the implementation order with the best return is: configuration with unaccent, weighted column and GIN index (one afternoon), endpoint with websearch_to_tsquery, ranking and snippets (another afternoon), search_log plus a weekly review (one recurring hour), typo fallback with pg_trgm (one morning), and from there, synonyms, hybrid search, and evals as the log demands them. Each step works on its own and none commits you to the next. The official PostgreSQL text search documentation is excellent and fits in one reading session; the pg_trgm and pgvector pages complete the map. And if this resource saved you the week of trial and error this topic costs almost everyone, it has done its job.