Facsímil 04

La caja de herramientas

Capítulo 08

Bases vectoriales, filtros y búsqueda híbrida

686f6c61 · 2026

Capítulo 08: Bases vectoriales, filtros y búsqueda híbrida

Cuando el vector deja de ser el problema

En el capítulo 07 construimos la pieza básica: convertir textos en vectores, compararlos y ordenar resultados. Eso ya permite hacer una búsqueda semántica pequeña. Pero en cuanto el sistema deja de ser una demo, aparece otra pregunta: dónde viven esos vectores, cómo se filtran, cómo se actualizan, cómo se borran y cómo sabemos que el índice no está devolviendo resultados bonitos pero equivocados.

Imagina una universidad con miles de documentos internos. Hay normativa de 2024, 2025 y 2026; manuales para estudiantes y profesorado; documentos públicos y documentos solo visibles para equipos concretos. La consulta "no puedo entrar a Moodle con doble factor" no puede devolver cualquier texto parecido. Debe devolver documentos vigentes, del curso correcto y visibles para la persona que pregunta.

Una base vectorial no es "una carpeta de embeddings". Es el lugar donde se cruzan similitud, filtros, permisos, versiones, índices, latencia, borrado y evaluación. Si esta pieza está mal diseñada, el RAG del capítulo 09 heredará el problema aunque el modelo generativo sea excelente.

Estado del arte con fecha de corte

Fecha de corte: 25 de mayo de 2026.
Fuentes consultadas ese día: documentación oficial de Qdrant, pgvector, Weaviate, Milvus y Pinecone; y trabajos académicos sobre FAISS, HNSW, cuantización de producto, BM25 y fusión de rankings.

Lo estable es la arquitectura: guardar vectores con identificadores, texto y metadata; construir índices; filtrar candidatos; combinar señales densas y léxicas; medir recall y latencia. Lo cambiante son APIs concretas, parámetros, límites por plan, soporte de filtros, algoritmos de índice, modelos integrados y costes de almacenamiento.

FUENTE	QUÉ APORTA	CÓMO USARLA
FAISS.[1]	Muestra técnicas de búsqueda eficiente de vectores a gran escala.	Para entender por qué no basta hacer producto punto contra todo cuando crece el corpus.
HNSW.[2]	Formaliza un índice por grafo usado por muchas bases vectoriales.	Para entender el intercambio entre memoria, construcción, rapidez y recall.
Qdrant.[3]	Explica que índice vectorial e índice de payload resuelven partes distintas del problema.	Para no confundir "tengo HNSW" con "mis filtros ya van bien".
pgvector.[4]	Integra búsqueda vectorial dentro de PostgreSQL con operadores de distancia e índices HNSW e IVFFlat.	Para proyectos donde transacciones, SQL, joins y vectores conviven en la misma base.
Weaviate.[5]	Documenta búsqueda híbrida que combina vector y BM25F mediante fusión configurable.	Para ver el patrón denso + léxico sin construir todo a mano.
Milvus.[6]	Distingue filtrado estándar e iterativo en búsqueda vectorial con metadata.	Para razonar sobre filtros complejos y latencia.
Pinecone.[7]	Compara usar un índice híbrido único frente a índices densos y dispersos separados.	Para entender que "híbrido" también es una decisión de arquitectura.

Qué no es una base vectorial

Una base vectorial no arregla embeddings malos. Si el modelo coloca cerca documentos que no deberían estarlo, el índice solo acelerará ese error. Tampoco arregla un mal troceado: si guardas párrafos sin título, sin sección y sin fecha, recuperarás fragmentos pobres con mucha rapidez.

Tampoco es una base de conocimiento completa. El vector no sustituye al texto original, a las citas, a las reglas de acceso, al historial de versiones ni al sistema que decide si un documento está vigente. La base vectorial guarda una representación para buscar; la verdad documental sigue viviendo en el contenido y en la metadata.

Y no es siempre la herramienta adecuada. Para cien documentos, una búsqueda exacta en memoria puede bastar. Para datos relacionales con filtros complejos, PostgreSQL con `pgvector` puede ser suficiente. Para millones de fragmentos, alta concurrencia, filtros frecuentes o múltiples señales de ranking, conviene pensar en una base vectorial dedicada o en un buscador que combine índice invertido y vectores.

Qué sí es: un contrato de recuperación

Una base vectorial no guarda solo vectores: guarda **registros**. Cada registro r_i es una tupla con identificador, vector, texto, metadata y versión, y el contrato de recuperación se define sobre esa estructura, no sobre el vector aislado:

$$r_i = (id_i,\ v_i,\ texto_i,\ m_i,\ version_i)$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
r_i	Registro número i.	Fragmento de una normativa.
id_i	Identificador estable.	`normativa-2026#sec-04`.
v_i	Vector del fragmento.	768 números `float32`.
$texto_i$	Texto recuperable o puntero al texto.	Párrafo que se pasará al RAG.
m_i	Metadata o payload.	`curso=2026`, `rol=estudiante`, `vigente=true`.
$version_i$	Versión de embedding y documento.	`embed-v3-large@2026-05-25`.

La búsqueda con filtro se expresa así:

$$\mathrm{TopK}(q, C, F, k) = \{r_{(1)}, \ldots, r_{(k)}\}$$

$$r_{(j)} \in C, \quad F(m_{(j)}) = 1, \quad s(q, v_{(1)}) \geq \cdots \geq s(q, v_{(k)})$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
q	Vector de la consulta.	Embedding de "acceso a Moodle".
C	Colección donde buscamos.	Fragmentos de documentación interna.
F	Función de filtro sobre metadata.	`curso == 2026 and vigente == true`.
$m_{(j)}$	Metadata del resultado en posición j.	Curso, idioma, rol, fuente.
$s(q, v)$	Puntuación de similitud.	Coseno o producto punto.
k	Número de resultados devueltos.	8 fragmentos para un RAG.

Esta fórmula tiene una lección importante: el filtro no es decoración posterior. Forma parte de lo que significa "resultado válido". Si el sistema encuentra el fragmento más parecido del mundo pero no cumple `F`, ese resultado no debería existir para la consulta.

El coste real: vectores, índices y payload

En el capítulo anterior calculamos el coste bruto de guardar vectores: $N \cdot d \cdot b$. En una base vectorial real, esa es solo la primera línea de la factura de memoria; el total suma índice, payload y réplicas:

$$M_{\text{total}} \approx N \cdot d \cdot b + M_{\text{índice}} + M_{\text{payload}} + M_{\text{réplicas}}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
N	Número de registros vectoriales.	10 millones de chunks.
d	Dimensión del vector.	768 dimensiones.
b	Bytes por componente.	4 bytes en `float32`, 2 en `float16`.
$M_{\text{índice}}$	Memoria del índice ANN.	Grafo HNSW o listas IVF.
M_{payload}	Metadata, texto corto, ids y estructuras auxiliares.	Fechas, permisos, fuente, idioma.
$M_{\text{réplicas}}$	Copias por disponibilidad o rendimiento.	Dos réplicas duplican parte del coste.

Con 10 millones de vectores de 768 dimensiones en `float32`, solo el bloque vectorial ocupa alrededor de 30,72 GB. Eso no incluye índice, payload, logs, réplicas, snapshots ni espacio temporal para reconstruir índices. Por eso la dimensión del capítulo 07 y la operación del capítulo 06 vuelven aquí con fuerza.

La selectividad del filtro también importa:

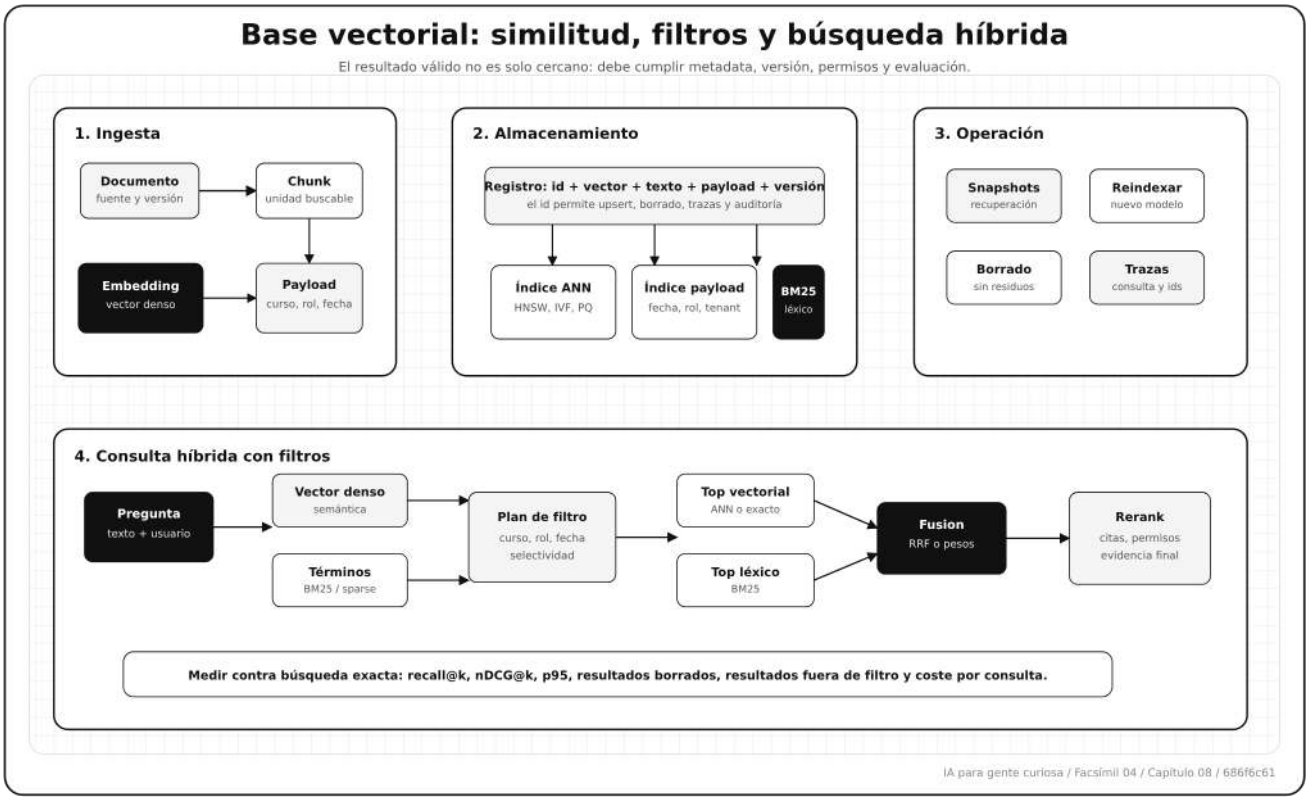
$$\sigma(F) = \frac{|\{r_i \in C : F(m_i) = 1\}|}{|C|}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$\sigma(F)$	Fracción de la colección que pasa el filtro.	0,02 si quedan 2 de cada 100.
$	C	$
$	{…}	$

Un filtro con $\sigma = 0{,}9$ apenas reduce el problema. Uno con $\sigma = 0{,}001$ puede romper supuestos del índice aproximado si no está bien planificado. Las bases vectoriales serias dedican mucha ingeniería a combinar índice vectorial e índice de metadata porque las dos piezas tiran en direcciones distintas.

Cómo funciona por dentro

Una base vectorial tiene dos rutas principales: ingesta y consulta. En ingesta recibe texto, genera o recibe embeddings, valida el esquema, guarda payload y actualiza índices. En consulta recibe una pregunta, genera el vector de consulta, aplica filtros, busca candidatos, fusiona señales si hay búsqueda híbrida y devuelve resultados con puntuaciones y metadata.



La imagen resume el punto central: una consulta real atraviesa dos mundos. Por un lado está la cercania semántica; por otro, el contrato operativo que decide si ese resultado se puede usar. El producto final no debería aceptar un resultado solo porque su vector está cerca.

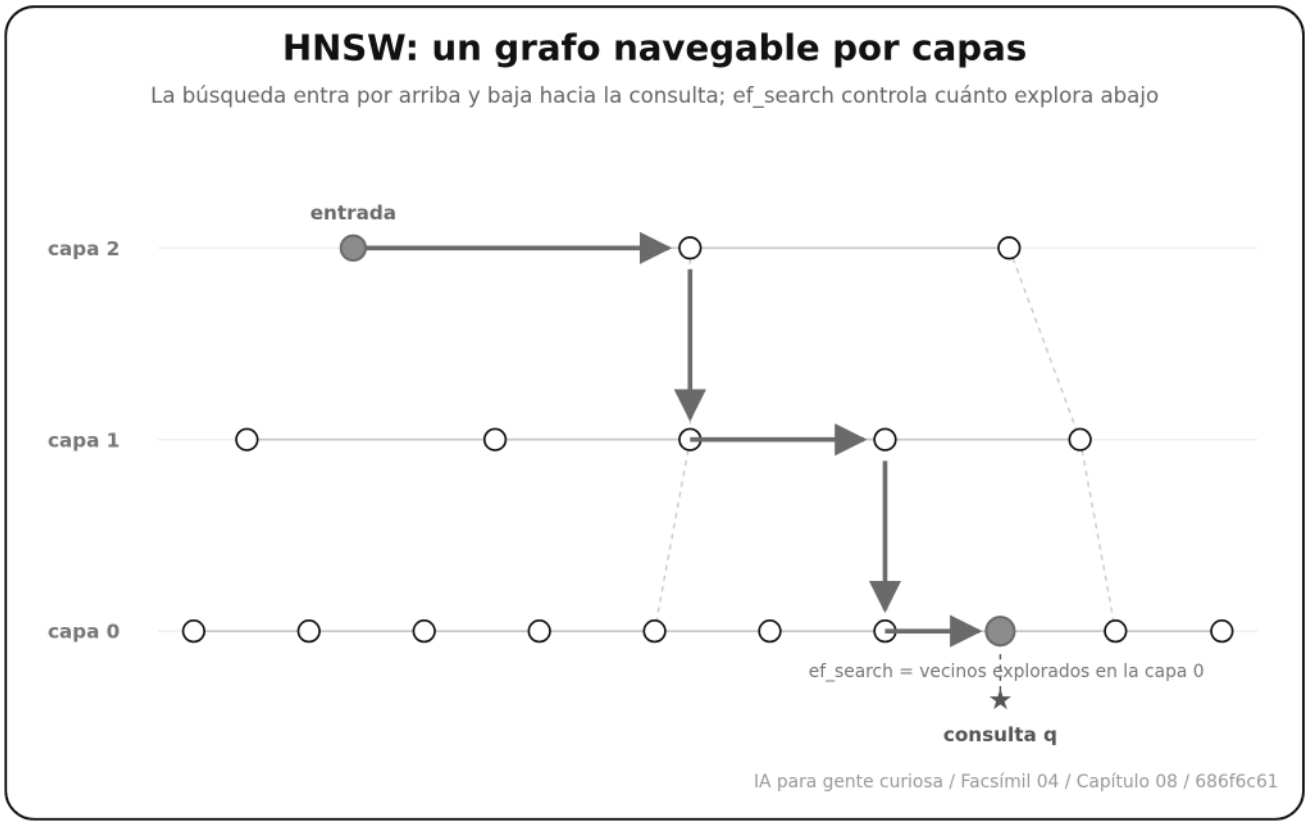
Índices: exacto, HNSW, IVFFlat y compresión

La búsqueda exacta compara la consulta con todos los vectores. Es fácil de razonar y sirve como referencia de calidad, pero su coste crece con $N \cdot d$. A partir de cierto tamaño, necesitamos índices aproximados de vecinos cercanos.

OPCIÓN	IDEA	PARÁMETROS TÍPICOS	QUÉ SE MIDE
Exacta	Comparar contra todos los vectores que pasan el filtro.	Sin índice ANN.	Calidad máxima, latencia base.
HNSW	Navegar un grafo de vecinos por capas.	`M`, `ef_construction`, `ef_search`.	Recall frente a memoria y p95.
IVFFlat	Dividir vectores en listas y buscar solo algunas.	`lists`, `probes`.	Recall frente a velocidad y coste de build.
PQ	Comprimir vectores por subespacios.	Número de subcuantizadores y bits.	Ahorro de memoria frente a pérdida de precisión.

HNSW suele dar buen equilibrio de recall y latencia, pero no es gratis: guarda conexiones entre vectores y consume memoria adicional.[8] IVFFlat puede construir más rápido y ocupar menos, pero requiere elegir listas y probes con cuidado; la documentación de pgvector lo explica como un intercambio entre rendimiento y recall.[9] La cuantización de producto reduce memoria representando subespacios con códigos compactos, pero introduce aproximación adicional.[10]

Conviene tener una imagen mental de HNSW, porque es el índice más usado. Es un grafo navegable de pequeño mundo organizado en **capas**: arriba hay pocos nodos con saltos largos, y abajo están todos los vectores con conexiones cortas. La búsqueda entra por la capa superior, se acerca a la consulta dando saltos grandes y baja de capa en capa afinando, hasta hacer una búsqueda fina en la capa 0. Ese descenso es lo que la hace rápida; `ef_search` controla cuántos vecinos explora al final.



Un criterio práctico: conserva siempre un modo de evaluación exacta, aunque sea sobre una muestra. Si no puedes comparar el índice aproximado contra el ranking exacto, no sabes si ganar latencia te está costando evidencias importantes.

La métrica que de verdad mide un índice aproximado

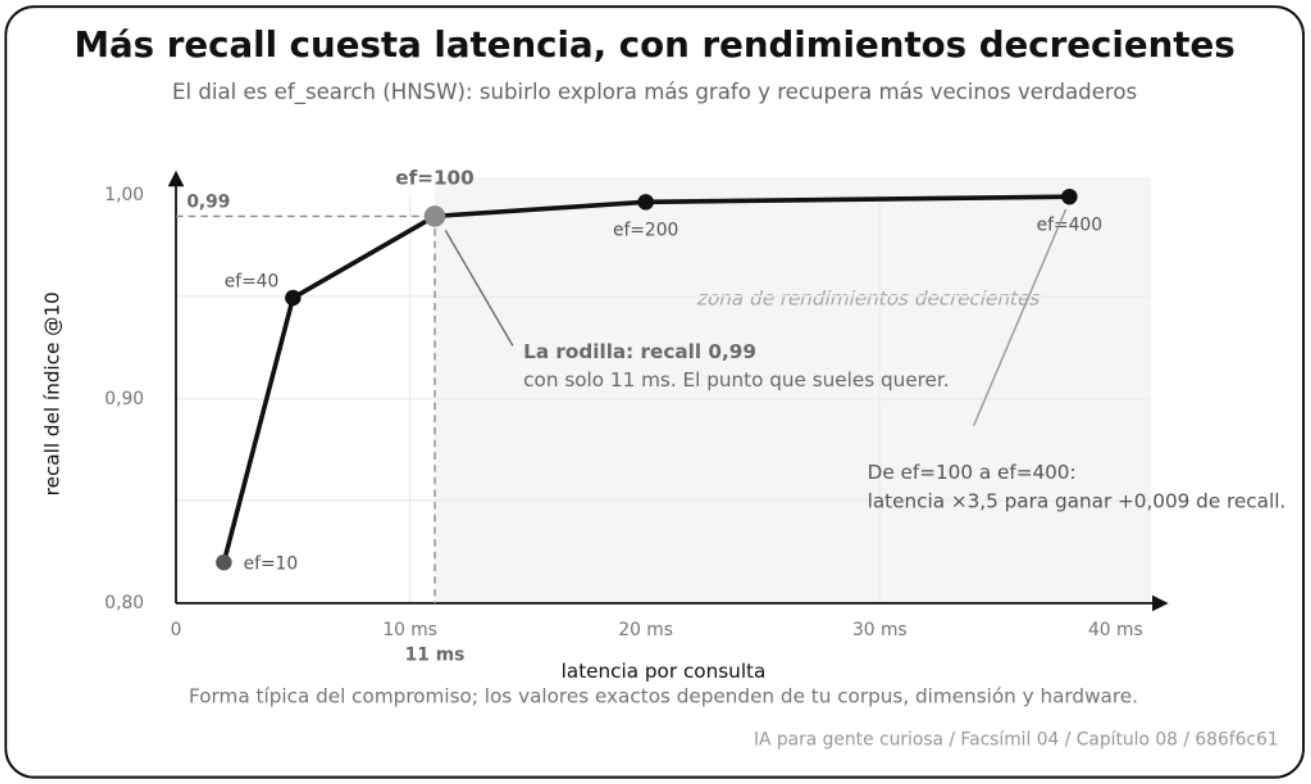
«Aproximado» tiene un número. El **recall del índice ANN** mide qué fracción de los verdaderos k vecinos más cercanos (los que daría la búsqueda exacta) recupera el índice aproximado para la misma consulta. Es la métrica estándar con la que se comparan los algoritmos de vecinos aproximados:[11]

$$\text{Recall}_{\text{ANN}}@\text{k} = \frac{|R_{\text{ANN}}(q,k) \cap R_{\text{exacto}}(q,k)|}{k}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
$R_{\text{ANN}}(q,k)$	Los k resultados que devuelve el índice aproximado.	Top-10 de HNSW.
$R_{\text{exacto}}(q,k)$	Los k verdaderos vecinos más cercanos (fuerza bruta).	Top-10 exacto sobre la misma colección.
$\cap$	Intersección: cuántos coinciden.	9 de 10.

No hay que confundir este recall con el **recall de recuperación** del capítulo 07: aquel mide si encuentras el documento relevante para una persona; este mide si el índice rápido encuentra lo mismo que el lento. Un índice puede tener `Recall_ANN@10 = 0,9` (pierde uno de cada diez vecinos verdaderos) y aun así servir bien si esos vecinos perdidos no eran los relevantes, o fatal si lo eran. Por eso se miden los dos.

El recall del índice no es fijo: es un dial. En HNSW lo gobierna `ef_search`; subirlo explora más grafo, sube el recall y cuesta latencia, con rendimientos decrecientes. La forma de ese compromiso es la que de verdad eliges al operar una base vectorial:



Comprimir para que quepa: int8, binario y rerank

La product quantization de la tabla no es la única forma de pagar menos memoria. A escala, lo más directo es bajar la precisión de cada componente del vector, y la diferencia es brutal:

REPRESENTACIÓN	BYTES POR COMPONENTE	10M VECTORES DE 768 DIMS	FACTOR
`float32`	4	30,72 GB	x1
`int8` (cuantización escalar)	1	7,68 GB	x4
Binario (1 bit por dimensión)	0,125	0,96 GB	x32

La cuantización escalar a `int8` mapea cada número real a un entero de 8 bits y suele perder muy poca calidad. La **cuantización binaria** lleva cada dimensión a un solo bit (el signo), permite comparar vectores con distancia de Hamming (operaciones sobre bits, baratísimas) y reduce la memoria 32 veces, a costa de una pérdida de precisión mayor.

El patrón de ingeniería que recupera esa precisión es el mismo de dos fases que vimos con los rerankers: **buscar en comprimido, reordenar en preciso**. Recuperas un top-k amplio (por ejemplo 200) con vectores binarios, rapidísimo y con poca memoria, y reordenas esos 200 candidatos con los vectores `float32` (o `int8`) originales para recuperar el orden fino. Así pagas memoria de índice barata para el grueso y precisión completa solo donde decide el ranking. Como siempre, la regla es medir el `Recall_ANN@k` de la versión comprimida frente a la exacta antes de comprometerte: si binario mantiene tu recall de recuperación, el ahorro de 32x es difícil de rechazar; si lo hunde, int8 suele ser el término medio sensato.

Filtros: dónde se gana o se rompe la recuperación

Los filtros parecen sencillos hasta que crece el corpus. Filtrar por `curso=2026` es fácil; filtrar por `tenant`, `rol`, `vigente`, `idioma`, `producto`, `region`, `tipo_documento` y `fecha` ya obliga a planificar.

Hay tres patrones frecuentes:

PATRÓN	CÓMO FUNCIONA	RIESGO
Filtrar antes	Primero reduce candidatos por metadata y luego busca vectores.	Si queda muy poco, el índice ANN puede no aportar mucho.
Buscar antes	Primero recupera muchos vecinos y luego descarta por metadata.	Puede perder resultados válidos si estaban fuera del primer lote.
Filtrado integrado	El índice combina metadata y navegación vectorial.	Requiere crear buenos índices de payload y entender selectividad.

Qdrant lo dice de forma muy clara: el índice vectorial acelera la búsqueda vectorial y los índices de payload aceleran filtros; hacen trabajos distintos.[12] Milvus también separa filtrado estándar e iterativo porque filtros complejos pueden cambiar mucho la latencia.[13]

Para entenderlo, piensa en una biblioteca. Si buscas "reglamento de prácticas" en toda la biblioteca y luego tiras los libros antiguos, quizá no veas el reglamento correcto porque no entró en el primer top-k. Si primero entras en la estantería "2026" y después buscas por significado, reduces ruido. Pero si la estantería tiene solo tres documentos, un recorrido exacto puede ser mejor que un índice sofisticado.

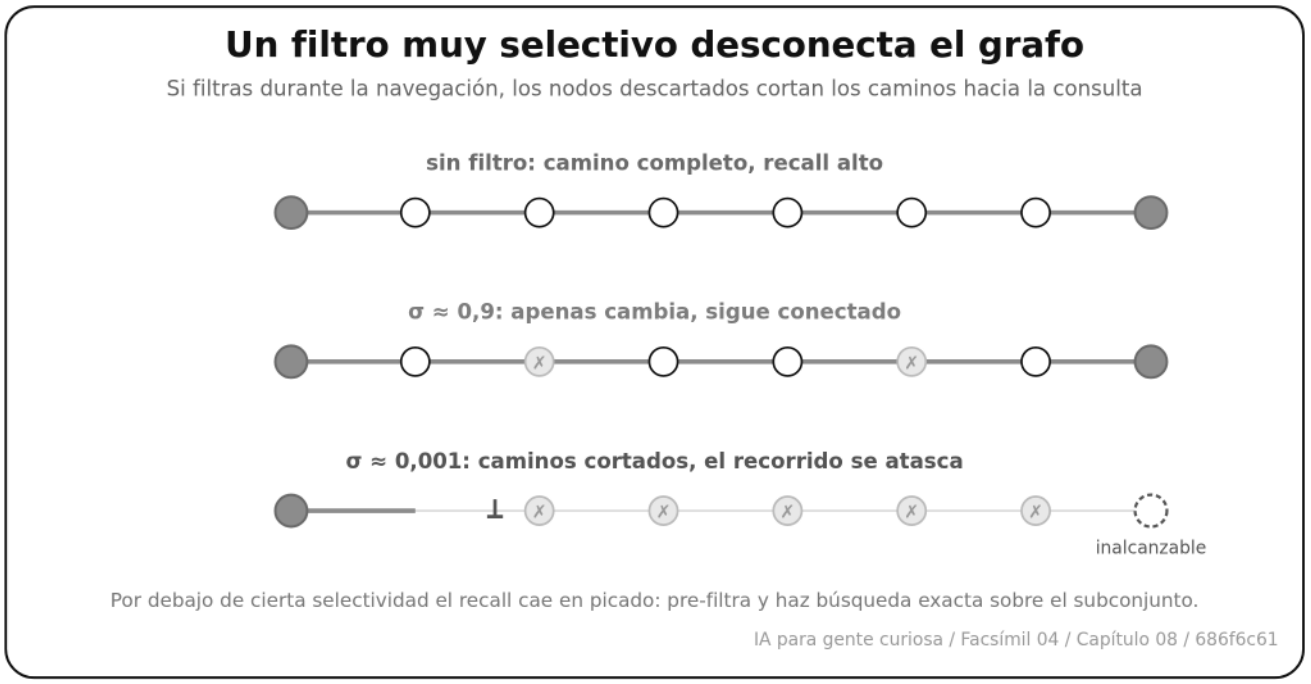
Por qué un filtro muy selectivo rompe HNSW

La frase del capítulo anterior («un filtro con $\sigma = 0{,}001$ puede romper supuestos del índice») tiene un mecanismo concreto, y conviene entenderlo. HNSW busca **navegando un grafo**: salta de vecino en vecino acercándose a la consulta. Ese recorrido funciona porque el grafo está bien conectado. Cuando aplicas un filtro muy selectivo *durante* la navegación (filtrado integrado), la mayoría de los nodos por los que el algoritmo querría pasar quedan descartados, el camino se corta y la búsqueda se queda atascada en una región sin poder llegar a los verdaderos vecinos que sí cumplen el filtro. El resultado es un **precipicio de recall**: por debajo de cierta selectividad, el recall del índice cae en picado aunque la respuesta correcta exista en la colección.

Por eso la selectividad del filtro decide la estrategia, no es un detalle:

SELECTIVIDAD Σ	QUÉ PASA	ESTRATEGIA RAZONABLE
Alta ($\sigma \approx 0{,}9$)	El filtro apenas reduce; el grafo casi no cambia.	Filtrado integrado; el ANN funciona bien.
Media ($\sigma \approx 0{,}1$)	Zona delicada; el recall puede empezar a sufrir.	Medir recall con y sin filtro; ajustar `ef_search`.
Baja ($\sigma \approx 0{,}001$)	Quedan tan pocos candidatos que el grafo se desconecta.	Pre-filtrar y hacer búsqueda exacta sobre el subconjunto: con pocos vectores, la fuerza bruta gana al grafo.

La regla de ingeniería: no elijas pre, post o integrado por defecto; elígelo según la selectividad esperada de tus filtros, y mide el `Recall_ANN@k` **con el filtro puesto**, no solo sin él.



Búsqueda híbrida: cuando exactitud léxica y semántica se necesitan

Los embeddings son fuertes con sinónimos e intención. BM25 es fuerte con palabras raras, siglas, códigos, nombres propios y errores donde la palabra exacta importa. La búsqueda híbrida combina ambas señales.

BM25, simplificado, puntúa una consulta Q sobre un documento D así:

$$\mathrm{BM25}(D,Q) = \sum_{t \in Q} \mathrm{IDF}(t) \frac{f(t,D)(k_1+1)}{f(t,D) + k_1\left(1 - b + b\frac{|D|}{avgdl}\right)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
t	Término de la consulta.	`MFA`, `Moodle`, `matrícula`.
$f(t,D)$	Veces que aparece el término en el documento.	3 apariciones de `Moodle`.
$\mathrm{IDF}(t)$	Peso de rareza del término en la colección.	`MFA` pesa más que `el`.
$	D	$
$avgdl$	Longitud media de documentos.	220 tokens.
k_1, b	Parámetros de saturación y longitud.	Valores habituales: $k_1 = 1{,}2$, $b = 0{,}75$.

BM25 viene de la familia probabilística de recuperación de información y sigue siendo una base fuerte para búsqueda léxica.[14] La parte densa recupera significado; la parte léxica protege términos que no deberían diluirse.

Una forma sencilla de fusionar rankings es RRF:

$$\mathrm{RRF}(d) = \sum_{j=1}^{S} \frac{1}{k_0 + \mathrm{rank}_j(d)}$$

SÍMBOLO	SIGNIFICADO	EJEMPLO
d	Documento candidato.	`doc-01`.
S	Número de sistemas que devuelven ranking.	Vectorial y BM25: $S = 2$.
$\mathrm{rank}_j(d)$	Posicion del documento en el sistema j.	1 en BM25, 5 en vectorial.
k_0	Constante que suaviza el peso de la posición.	60 es un valor comun en RRF.

RRF funciona bien porque no exige que las puntuaciones de BM25 y embeddings estén en la misma escala.[15] Eso es práctico: un coseno de 0,72 y un BM25 de 11,4 no son directamente comparables, pero sus posiciones en rankings sí pueden combinarse.

CONSULTA	VECTOR DENSO AYUDA	BM25 AYUDA	HIBRIDO EVITA
"no puedo entrar al campus"	Encuentra "restablecer acceso a Moodle".	Poco, si no comparte palabras.	Quedarse solo con sinónimos.
"error SAML 403 Moodle"	Puede entender "login".	Protege `SAML` y `403`.	Perder códigos exactos.
"matrícula 2026 septiembre"	Relaciona matrícula con calendario.	Protege `2026` y `septiembre`.	Devolver normativa antigua.
"API pagos webhook reintentos"	Relaciona integración y eventos.	Protege `webhook`.	Mezclar artículos de producto.

Weaviate expone búsqueda híbrida como combinacion de resultados vectoriales y BM25F por fusión configurable.[16] Pinecone documenta dos caminos: índice híbrido único o índices densos y dispersos separados, cada uno con sus ventajas operativas.[17] La idea pedagógica es la misma: no hay que elegir religión entre vector y palabras exactas; hay que medir cuál combina mejor en tu corpus.

Más allá de un vector por documento: interacción tardía

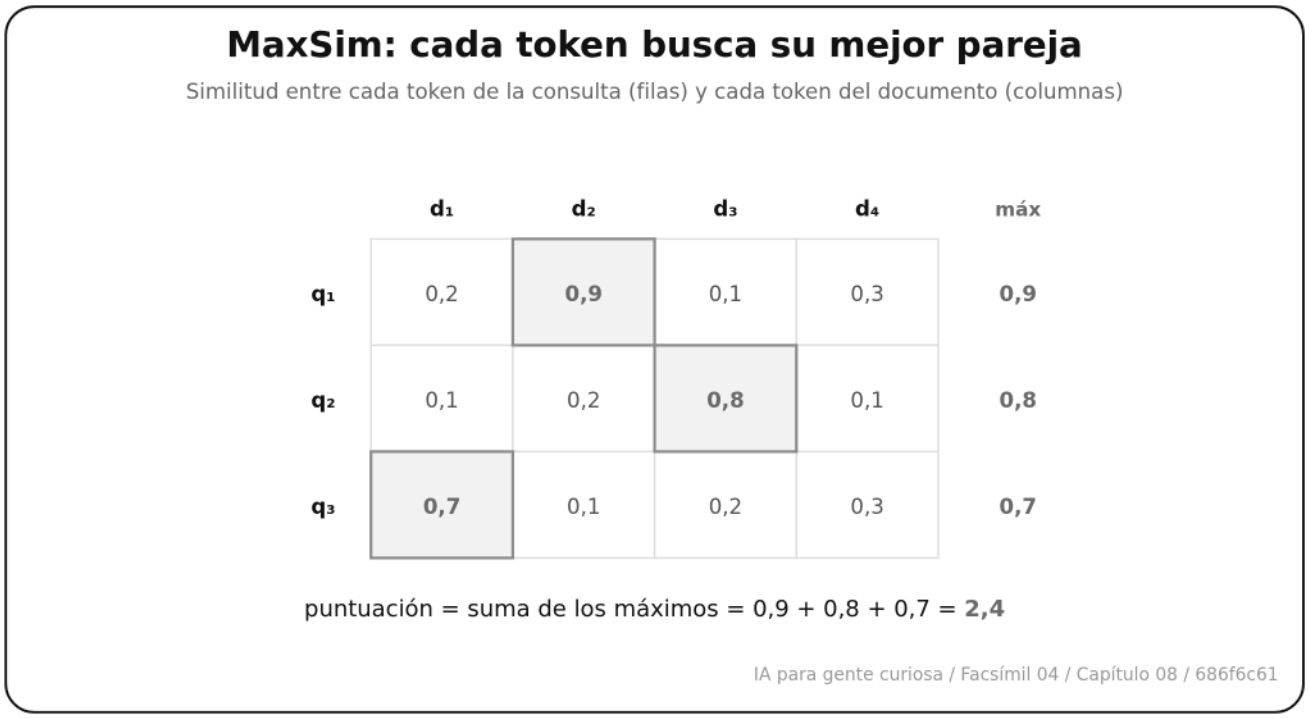
Hasta aquí hemos asumido **un vector por documento**: el bi-encoder comprime todo el texto en un único punto del espacio. Es eficiente, pero pierde detalle, porque un párrafo largo no cabe bien en una sola coordenada. Hay un paradigma alternativo que conviene conocer: la **interacción tardía** (late interaction), popularizada por ColBERT.[18]

En lugar de un vector por documento, ColBERT guarda **un vector por token** y puntúa con MaxSim: para cada token de la consulta busca el token del documento más parecido y suma esas máximas similitudes.

$$s(q, d) = \sum_{i \in q} \max_{j \in d} \operatorname{sim}(q_i, d_j)$$

SÍMBOLO	SIGNIFICADO
q_i	Vector del token i de la consulta.
d_j	Vector del token j del documento.
$\max_{j \in d}$	Para cada token de la consulta, su mejor coincidencia en el documento.
$\sum_{i \in q}$	Suma de esas coincidencias sobre todos los tokens de la consulta.

La ventaja es precisión: al comparar token a token, capta coincidencias finas que un solo vector promediaría y perdería. El coste es memoria e índice: guardar decenas de vectores por documento multiplica el almacenamiento y exige un índice especializado, por lo que en la práctica suele usarse como **reordenador** sobre los candidatos de un bi-encoder, igual que el cross-encoder del capítulo 07, pero más barato de servir. La decisión es la de siempre: la interacción tardía compensa cuando tu evaluación muestra que el bi-encoder pierde matices que importan; si no, su coste de almacenamiento no se justifica.



Diseñar el esquema de una colección

Antes de indexar, conviene escribir el contrato. Una colección debería responder estas preguntas:

DECISIÓN	PREGUNTA TÉCNICA	MALA SEÑAL
Identificador	Qué id estable permite reindexar sin duplicar?	IDs aleatorios sin relación con fuente y sección.
Texto	Guardamos texto completo o puntero?	Resultados sin cita recuperable.
Metadata	Qué campos se filtran de verdad?	Guardar metadata bonita que nunca se indexa.
Versión	Qué modelo, dimensión y fecha genero el vector?	Mezclar embeddings incompatibles.
Permisos	El filtro de acceso vive en la consulta?	Filtrar después de mostrar candidatos.
Vigencia	Cómo caduca o se reemplaza un documento?	Resultados de años anteriores en top-k.
Borrado	Qué significa borrar: vector, payload, texto y cache?	Quedan fragmentos recuperables por accidente.

Si usas PostgreSQL con `pgvector`, el esquema puede ser explícito:

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE chunks (
  id text PRIMARY KEY,
  source_id text NOT NULL,
  chunk_text text NOT NULL,
  embedding vector(768) NOT NULL,
  curso integer NOT NULL,
  rol text NOT NULL,
  vigente boolean NOT NULL,
  embedding_model text NOT NULL,
  indexed_at timestamptz NOT NULL DEFAULT now()
);

CREATE INDEX chunks_embedding_hnsw
ON chunks USING hnsw (embedding vector_cosine_ops);

CREATE INDEX chunks_metadata
ON chunks (curso, rol, vigente);

SELECT id, chunk_text
FROM chunks
WHERE curso = 2026
  AND rol IN ('estudiante', 'publico')
  AND vigente = true
ORDER BY embedding <=> $1
LIMIT 8;
```

El detalle importante no es memorizar esta sintaxis. El detalle es que el campo vectorial, los filtros y la versión viven juntos. Si cambias el modelo de embedding o la dimensión, no estás "actualizando una columna"; estás cambiando el espacio de búsqueda.

Cómo trabajar con bases vectoriales con criterio

Una base vectorial en producción necesita disciplina operativa. La parte difícil no es insertar el primer vector; es mantener el sistema correcto cuando cambian documentos, permisos, modelos y volumen.

PRÁCTICA	QUÉ HACES	POR QUÉ IMPORTA
IDs deterministas	Derivas el id de fuente, sección y versión.	Permite `upsert` idempotente y evita duplicados.
Versionado de embeddings	Guardas modelo, dimensión, normalización y fecha.	Permite reindexar y comparar variantes.
Doble índice temporal	Construyes el nuevo índice junto al anterior.	Evita cortar servicio mientras migras.
Borrado verificable	Compruebas que ids borrados no vuelven en top-k.	Evita respuestas con contenido retirado.
Filtros obligatorios	El backend añade filtros de permisos siempre.	El cliente no decide qué puede ver.
Evaluación continua	Mides recall, p95, coste y errores por segmento.	Detecta degradación antes que usuarios y usuarias.
Trazas de retrieval	Guardas consulta, filtros, ids, scores y versión.	Permite explicar por qué se recuperó algo.

Una buena pregunta de ingeniería: si mañana cambiamos de `all-MiniLM-L6-v2` a otro modelo de 1024 dimensiones, ¿qué pasos exactos hay que hacer? Si la respuesta no incluye reindexado, evaluación, cambio de versión y plan de retirada del índice anterior, falta diseño.

Frescura: por qué borrar es más difícil que insertar

Un índice no es una foto: cambia cada día. Insertar vectores nuevos es relativamente fácil, porque HNSW admite añadir nodos al grafo de forma incremental. **Borrar es el problema.** En un grafo HNSW, los demás nodos pueden tener aristas que apuntan al vector que quieres eliminar; quitarlo de golpe dañaría caminos de navegación. Por eso la mayoría de los sistemas no borran de verdad: marcan el vector con un **tombstone** (una lápida) y lo excluyen de los resultados, pero sigue ocupando memoria y participando en el grafo. Cuando se acumulan muchos tombstones, el índice se degrada (más memoria, peor recall) y hay que **reconstruirlo** periódicamente.

Las consecuencias prácticas: un documento «retirado» puede seguir vivo en el índice si tu flujo de borrado solo lo quita de la base de datos pero no del índice vectorial; actualizar un documento es en realidad borrar más insertar (con re-embedding si cambió el texto); y la consistencia entre tu fuente de verdad y el índice es eventual, no inmediata. Por eso el «borrado verificable» de la tabla anterior no es paranoia: es la única forma de saber que el contenido retirado no vuelve en un top-k.

Cuando no cabe en una máquina: sharding e índice en disco

A escala de cientos de millones o miles de millones de vectores, el índice deja de caber en la RAM de un solo servidor, y aparecen dos estrategias. La primera es el **sharding**: partir la colección entre varios nodos, lanzar la consulta a todos en paralelo y fusionar sus resultados (un patrón *scatter-gather*), con réplicas para disponibilidad y para absorber más consultas por segundo. La segunda es mover el índice al **disco**: sistemas como DiskANN guardan el grafo en SSD y usan la RAM solo para la parte caliente de la navegación, lo que permite servir miles de millones de vectores desde una máquina a cambio de algo más de latencia.[19] Para la mayoría de proyectos esto es prematuro (con millones de vectores, una sola máquina sobra), pero conviene saber que el techo no es la API de embeddings, sino la memoria del índice, y que existe un plan cuando se alcanza.

Evaluar una base vectorial

Aquí evaluamos dos capas: la calidad de recuperación y la calidad operativa. La primera pregunta es "encuentro lo correcto?". La segunda es "lo encuentro dentro del contrato de producto?".

MÉTRICA	QUÉ MIDE	CÓMO SE CALCULA
Recall ANN@k	Cuánto se parece el índice aproximado al exacto.	Resultados ANN frente a búsqueda exacta.
Recall con filtro@k	Si aparecen documentos correctos cumpliendo metadata.	Casos con filtros obligatorios.
nDCG@k	Si los documentos más útiles suben arriba.	Relevancia graduada por posición.
p50, p95, p99	Latencia normal y de cola.	Tiempos por consulta real.
Tasa de resultados retirados	Cuántos resultados ya no deberían aparecer.	Hits con `vigente=false` o versión antigua.
Cobertura de permisos	Si cada consulta aplica el filtro correcto.	Trazas con usuario, rol y condición.
Coste por consulta	CPU, memoria, GPU o precio cloud.	Coste mensual dividido por consultas útiles.

Una prueba mínima compara tres rankings para las mismas consultas: exacto filtrado, ANN filtrado e híbrido filtrado. Si el ANN pierde documentos que el exacto encuentra, ajustas parámetros o cambias índice. Si el híbrido mejora consultas con siglas pero empeora consultas naturales, ajustas fusión o decides cuándo activarlo.

En el día a día

En una universidad, una base vectorial es lo que permite que el buscador de la intranet encuentre la normativa correcta aunque la persona no use las palabras exactas, y que solo le muestre la versión vigente de su curso. En soporte interno, es donde viven los miles de fragmentos de documentación que un asistente recupera antes de responder, con filtros por producto, idioma y permisos. En cualquier RAG de producción, la base vectorial es la pieza que decide qué evidencia llega al modelo: si recupera mal, el resto da igual.

La parte delicada es que casi nunca falla el coseno. Falla el filtro que no se aplicó, la versión antigua que nadie retiró, el documento que el índice aproximado no encontró porque alguien subió `ef_search` demasiado bajo, o el término exacto (`SAML` , `2026` , un número de factura) que la búsqueda solo densa diluyó. Por eso una base vectorial seria es tanto un problema de recuperación como de gobernanza de datos.

Por qué debería importarte

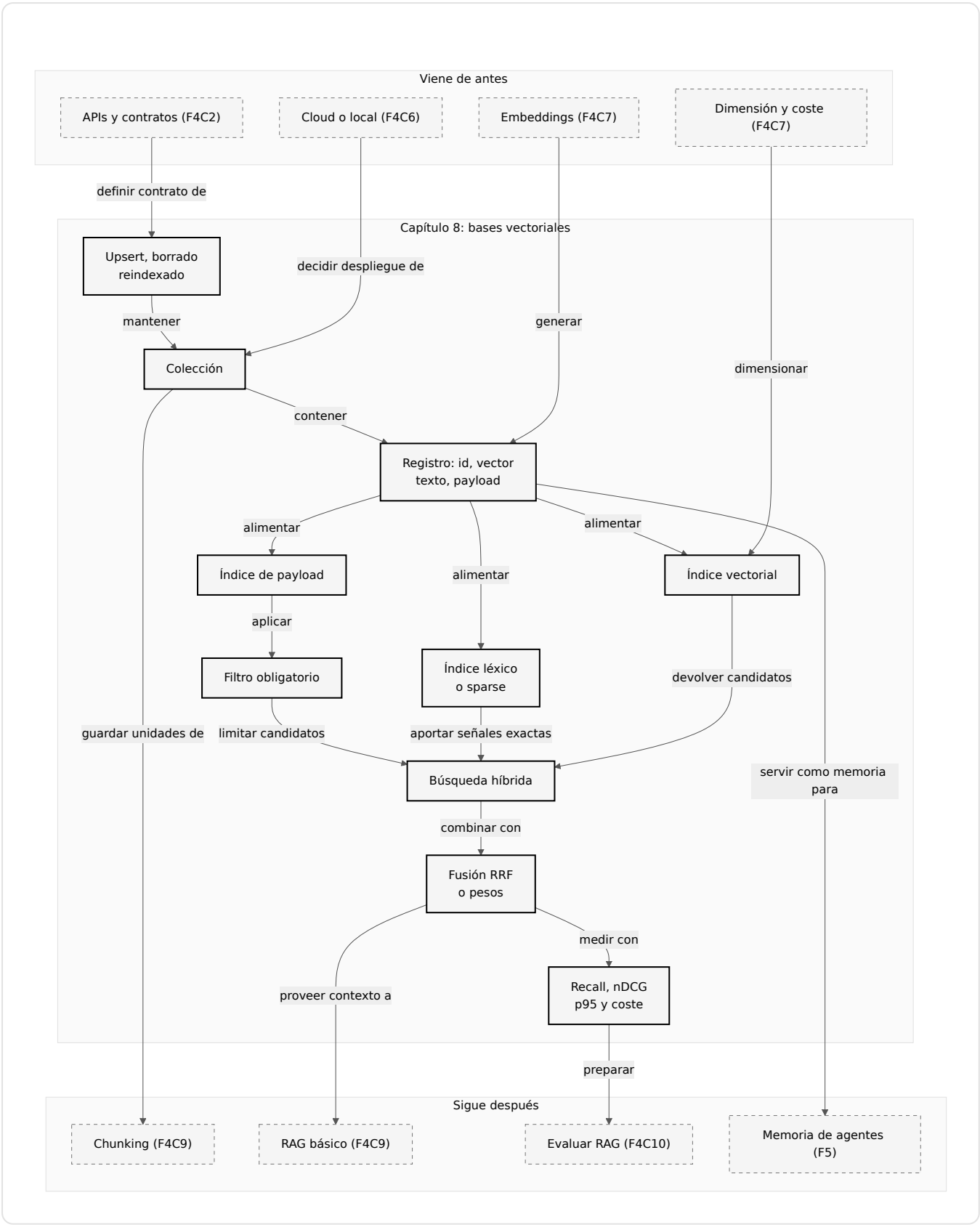
Porque es donde un RAG se rompe en silencio. Un modelo generativo excelente con una base vectorial mal diseñada produce respuestas seguras y equivocadas, y desde fuera parece un problema del modelo. Si entiendes esta pieza, sabes diagnosticar: si el fallo es de recuperación (no apareció el documento), de filtro (apareció uno que no debía), de índice (el ANN perdió un vecino), de fusión (lo léxico tapó lo semántico) o de versión (contenido retirado que sigue vivo).

También importa por coste y por escala. Diez millones de vectores no son «llamar a una API»: son decenas de gigas de memoria, índices que tardan en construirse, réplicas, copias de seguridad y una factura que crece con la dimensión y con cada filtro mal planificado. Elegir índice, cuantización y estrategia de filtrado con criterio es lo que separa un piloto que funciona en una demo de un sistema que aguanta usuarios reales.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Creer que vector DB equivale a RAG	La base recupera candidatos; no decide chunking, citas, abstención ni respuesta final.	Separar retrieval, contexto y generación.
Filtrar después de recuperar poco	Si pides top-10 global y luego descartas por permisos, puedes quedarte sin el resultado correcto.	Aplicar filtros como parte del plan de búsqueda.
No versionar embeddings	Mezclar modelos o dimensiones hace que las distancias dejen de significar lo mismo.	Guardar `embedding_model`, dimensión y fecha en cada registro.
Olvidar términos exactos	Siglas, códigos, IDs y nombres propios pueden perderse en búsqueda solo densa.	Probar búsqueda híbrida con BM25 o sparse vectors.
Medir solo latencia media	p95 o p99 pueden ser malos aunque la media parezca aceptable.	Medir percentiles y separar consultas con filtros complejos.
No probar borrados	Un índice puede seguir devolviendo contenido retirado si el flujo de borrado falla.	Crear tests de ids retirados y verificar que no aparecen.
Elegir herramienta por moda	Cada base cambia filtros, operación, costes, backup y SQL disponible.	Comparar con una matriz de requisitos reales.

Cómo encaja todo



Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Base vectorial	Sistema que guarda vectores y permite recuperarlos por similitud.
Colección	Conjunto de registros con el mismo contrato de vector y metadata.
Payload	Metadata asociada al vector, usada para filtrar y explicar resultados.
Índice vectorial	Estructura que acelera la búsqueda de vecinos cercanos.
HNSW	Índice por grafo navegable para búsqueda aproximada.
IVFFlat	Índice que divide el espacio en listas y busca solo algunas.
Filtro	Condición que limita qué registros pueden entrar en el ranking.
BM25	Ranking léxico basado en frecuencia, rareza y longitud de documento.
Búsqueda híbrida	Combinación de ranking vectorial y ranking léxico.
RRF	Fusión de rankings basada en posiciones.
Upsert	Inserción o actualización idempotente de un registro.

Antes de pasar página

- [ ] ¿Puedo explicar por qué una base vectorial no arregla embeddings malos? (Si no, vuelve a «Qué no es una base vectorial».)
- [ ] ¿Sé qué debe guardar un registro vectorial además del vector? (Si no, vuelve a «Qué sí es: un contrato de recuperación».)
- [ ] ¿Puedo calcular el coste bruto de memoria para N, d y bytes por componente? (Si no, vuelve a «El coste real: vectores, índices y payload».)
- [ ] ¿Sé explicar qué mide la selectividad de un filtro? (Si no, vuelve a «El coste real: vectores, índices y payload».)
- [ ] ¿Sé distinguir búsqueda exacta, HNSW, IVFFlat y PQ? (Si no, vuelve a «Índices: exacto, HNSW, IVFFlat y compresión».)
- [ ] ¿Sé por qué filtrar después de recuperar poco puede fallar? (Si no, vuelve a «Filtros: dónde se gana o se rompe la recuperación».)

- [ ] ¿Sé cuándo BM25 aporta algo que el embedding puede perder? (Si no, vuelve a «Búsqueda híbrida: cuando exactitud léxica y semántica se necesitan».)
- [ ] ¿Puedo explicar RRF sin comparar puntuaciones incompatibles? (Si no, vuelve a «Búsqueda híbrida: cuando exactitud léxica y semántica se necesitan».)
- [ ] ¿Sé qué campos versionar para reindexar con seguridad? (Si no, vuelve a «Diseñar el esquema de una colección».)
- [ ] ¿Sé qué métricas mirar además de recall: p95, coste y resultados retirados? (Si no, vuelve a «Evaluar una base vectorial».)

En resumen

IDEA FUERZA	DETALLE
Una base vectorial es un contrato operativo.	Guarda vectores, texto, ids, metadata, versiones e índices.
El filtro forma parte de la respuesta correcta.	Un resultado cercano pero fuera de curso, rol o vigencia no es válido.
El índice aproximado debe compararse con exacto.	Sin baseline exacto, no sabes cuánto recall pierdes por ganar latencia.
La búsqueda híbrida une dos señales.	Embeddings capturan significado; BM25 protege términos exactos.
Operar importa tanto como buscar.	Upsert, borrado, snapshots, reindexado y trazas deciden si el sistema aguanta.
La evaluación debe incluir producto.	Recall, nDCG, filtros, p95, coste y documentos retirados cuentan juntos.

Para saber más

Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). *Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods*. SIGIR, 758-759. https://doi.org/10.1145/1571941.1572114

Jégou, H., Douze, M. y Schmid, C. (2011). *Product Quantization for Nearest Neighbor Search*. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 33*(1), 117-128. https://doi.org/10.1109/TPAMI.2010.57

Johnson, J., Douze, M. y Jégou, H. (2019). *Billion-Scale Similarity Search with GPUs*. *IEEE Transactions on Big Data, 7*(3), 535-547. https://doi.org/10.1109/TBDATA.2019.2921572

Malkov, Y. A. y Yashunin, D. A. (2020). *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs*. *IEEE TPAMI, 42*(4), 824-836. https://doi.org/10.1109/TPAMI.2018.2889473

Milvus. (2026). *Filtered Search*. https://milvus.io/docs/filtered-search.md

pgvector. (2026). *pgvector: Open-source vector similarity search for Postgres*. https://github.com/pgvector/pgvector

Pinecone. (2026). *Hybrid search*. https://docs.pinecone.io/docs/hybrid-search-and-sparse-vectors

Qdrant. (2026). *Indexing*. https://qdrant.tech/documentation/manage-data/indexing/

Robertson, S. y Zaragoza, H. (2009). *The Probabilistic Relevance Framework: BM25 and Beyond*. *Foundations and Trends in Information Retrieval, 3*(4), 333-389. https://doi.org/10.1561/1500000019

Weaviate. (2026). *Hybrid search*. https://docs.weaviate.io/weaviate/search/hybrid

---

Notas

1. Johnson, J., Douze, M. y Jégou, H. (2019). *Billion-Scale Similarity Search with GPUs*. *IEEE Transactions on Big Data, 7*(3), 535-547. https://doi.org/10.1109/TBDATA.2019.2921572. ↩

2. Malkov, Y. A. y Yashunin, D. A. (2020). *Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs*. *IEEE TPAMI, 42*(4), 824-836. https://doi.org/10.1109/TPAMI.2018.2889473. ↩

3. Qdrant. (2026). *Indexing*. https://qdrant.tech/documentation/manage-data/indexing/. Consultado el 25 de mayo de 2026. ↩

4. pgvector. (2026). *pgvector: Open-source vector similarity search for Postgres*. https://github.com/pgvector/pgvector. Consultado el 25 de mayo de 2026. ↩

5. Weaviate. (2026). *Hybrid search*. https://docs.weaviate.io/weaviate/search/hybrid. Consultado el 25 de mayo de 2026. ↩

6. Milvus. (2026). *Filtered Search*. https://milvus.io/docs/filtered-search.md. Consultado el 25 de mayo de 2026. ↩

7. Pinecone. (2026). *Hybrid search*. https://docs.pinecone.io/docs/hybrid-search-and-sparse-vectors. Consultado el 25 de mayo de 2026. ↩

8. Malkov y Yashunin, 2020. ↩

9. pgvector, 2026. ↩

.0. Jégou, H., Douze, M. y Schmid, C. (2011). *Product Quantization for Nearest Neighbor Search*. *IEEE TPAMI, 33*(1), 117-128. https://doi.org/10.1109/TPAMI.2010.57. ↩

.1. Martin Aumüller, Erik Bernhardsson y Alexander Faithfull. (2020). *ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms*. Information Systems, 87. https://doi.org/10.1016/j.is.2019.02.006. ↩

.2. Qdrant, 2026. ↩

.3. Milvus, 2026. ↩

.4. Robertson, S. y Zaragoza, H. (2009). *The Probabilistic Relevance Framework: BM25 and Beyond*. *Foundations and Trends in Information Retrieval, 3*(4), 333-389. https://doi.org/10.1561/1500000019. ↩

.5. Cormack, G. V., Clarke, C. L. A. y Buettcher, S. (2009). *Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods*. SIGIR, 758-759. https://doi.org/10.1145/1571941.1572114. ↩

.6. Weaviate, 2026. ↩

.7. Pinecone, 2026. ↩

.8. Omar Khattab y Matei Zaharia. (2020). *ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT*. SIGIR. https://arxiv.org/abs/2004.12832. ↩

.9. Suhas Jayaram Subramanya y otros (2019). *DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node*. NeurIPS 2019. Implementación de referencia: https://github.com/microsoft/DiskANN. ↩