RAG Retrieval

Version Nov 2026 · pgvector 0.8 · verified 2026-11-14

Build the retrieval half of RAG — chunking, metadata, embeddings, pgvector search, filters, hybrid search with RRF and reranking.

Contents

Mental model 2

The retrieval pipeline 2

Chunking 3

Chunk metadata and parent context 4

Embeddings 6

Similarity search with pgvector 7

Filters and short results 9

Hybrid search and reciprocal rank fusion 10

Reranking 11

Further reading 12

Mental model

Retrieval-augmented generation is a search engine with a language model at the end, and the model can only answer from the chunks the search hands it. So most of the quality is decided before the model runs: how documents were cut into chunks, whether the right chunk ranks in the top k, and whether you can trace it back to a source. Treat retrieval as its own system with its own measurements — see the retrieval evaluation sheet.

The retrieval pipeline

Stage	Does	Runs
Chunk	Split documents into passages	At ingest
Embed	One vector per chunk	At ingest
Index	Vector index plus full-text index	At ingest
Retrieve	Top candidates for the query	Per query
Rerank	Re-score candidates, keep top k	Per query
Generate	Answer from the kept chunks, cite them	Per query

The ingest side runs once per document version; the query side runs on every request and is where latency lives. Keep the stages separate in code, so you can change chunking or the ranker and re-measure without touching the rest.

```
query -> embed -> vector search, top 50
query -> full-text search, top 50
```

```
both  -> fuse -> rerank -> keep top 10
top 10 -> prompt -> answer with citations
```

> **Tip:** If the whole corpus fits in the model's context — Anthropic suggests under about 200,000 tokens, roughly 500 pages — skip retrieval and put it all in the prompt, with prompt caching to keep repeated calls cheap.

Chunking

Rule	Starting point
One idea per chunk	Split on structure, not byte count
Size	300–800 tokens; tune on your eval set
Overlap	10–20 percent, so ideas at a boundary survive
Split order	Headings, then paragraphs, then sentences
Fixed windows	Only when there is no structure

A chunk is the unit you retrieve and the unit the model reads, so it must make sense alone. Too small and it loses the context that makes it answerable; too large and its embedding averages several topics and matches none of them well. Respect the document's own boundaries first, and only fall back to fixed-size windows when the text has no structure.

```
def chunk(paras, limit=300, overlap=1):
    out, cur = [], []
```

```
    for p in paras:
        n = sum(len(x.split()) for x in cur)
        if cur and n + len(p.split()) > limit:
            out.append("\n\n".join(cur))
            cur = cur[-overlap:]  # overlap
        cur.append(p)
    if cur:
        out.append("\n\n".join(cur))
    return out
```

> **Gotcha:** Word counts are not token counts. Tokenizers usually produce more tokens than words, and far more for code, tables and non-English text. Measure chunk size with the embedding model's own tokenizer before trusting a limit.

Chunk metadata and parent context

Store with every chunk	Used for
`doc_id`, chunk position	Parent lookup, deduplication
Section heading path	Context, display, filters
Page or character offset	Citations that point somewhere
Source URL, `updated_at`	Citations, freshness filters
Embedding model and version	Knowing what to re-embed

Metadata is what turns a matched passage into a citable answer and lets you filter by language, tenant or date. Keep the parent document or

section too: retrieve on small, precise chunks, then hand the model the surrounding section when one chunk is not enough context. Anthropic's contextual retrieval goes a step further and prepends a 50–100 token summary of where the chunk sits in its document before embedding and keyword-indexing it.

```
{
  "doc_id": 812,
  "position": 4,
  "heading": "Billing > Refunds",
  "source": "https://example.com/help/refunds",
  "page": 3,
  "updated_at": "2026-10-02",
  "model": "voyage-4"
}
```

> **Gotcha:** A chunk like "Revenue grew 3% over the prior quarter" embeds well but cannot be matched to the question "ACME Q2 2025 revenue", because the company and the quarter were in a heading the chunk lost. Put the heading path into the text you embed.

Embeddings

Rule	Why
Same model for documents and queries	Vectors from different models are not comparable
Set the query/document input type	Many retrieval models embed each side differently
Batch calls	Fewer requests, higher throughput
Store model name and version	A model change means re-embedding everything
Normalize, or use cosine distance	Dot product equals cosine only at length 1

An embedding model maps text to a fixed-length vector — 384, 1024 and 1536 dimensions are common, and some models let you choose — so that similar meanings land close together. The column type fixes the dimension, so switching to a model with a different size is a schema change as well as a full re-embed.

```
import math


def normalize(v):
    n = math.sqrt(sum(x * x for x in v))
    return [x / n for x in v]
```

```
def batches(items, size=128):
    for i in range(0, len(items), size):
        yield items[i:i + size]


normalize([3.0, 4.0])  # [0.6, 0.8]
```

> **Gotcha:** Voyage prepends a different prompt per `input_type` — "Represent the query for retrieving supporting documents" versus "Represent the document for retrieval" — and its docs say not to omit it. Embedding the question as a document still returns results, just worse ones, so nothing flags the mistake.

Similarity search with pgvector

Operator	Distance	Index ops
`<=>`	Cosine	`vector_cosine_ops`
`<->`	Euclidean (L2)	`vector_l2_ops`
`<#>`	Negative inner product	`vector_ip_ops`

Order by the distance operator and `LIMIT` the result; with an HNSW index that query becomes an approximate nearest-neighbour search. The operator in `ORDER BY` must match the index's operator class, or the index is not used. `<#>` returns the **negative** inner product, so smaller is still better.

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE TABLE chunks (
  id bigserial PRIMARY KEY,
  doc_id bigint NOT NULL,
  content text NOT NULL,
  metadata jsonb NOT NULL DEFAULT '{}',
  embedding vector(1536) NOT NULL
);
CREATE INDEX ON chunks
  USING hnsw (embedding vector_cosine_ops);
```

```
SELECT id, content,
  embedding <=> $1 AS distance
FROM chunks
WHERE metadata->>'lang' = 'en'
ORDER BY embedding <=> $1
LIMIT 10;
```

> **Note:** HNSW has build-time `m` (default 16) and `ef_construction` (default 64), and a per-session `hnsw.ef_search` (default 40); higher values raise recall and latency. IVFFlat builds faster but must be created after the table has data. Tuning detail is in the pgvector sheet.

Filters and short results

Situation	Fix
Filter keeps most rows	Plain query is fine
Selective filter, HNSW	`SET hnsw.iterative_scan = relaxed_order` (0.8+)
One filter value, very selective	Partial index, or partition by it
Exact results required	No vector index for that query

An approximate index finds its nearest candidates first and the `WHERE` clause runs on them afterwards. If a filter matches 10 percent of rows, HNSW with the default `ef_search` of 40 returns about 4 rows for a `LIMIT 10`. pgvector 0.8 added iterative index scans, which keep scanning until enough rows pass the filter or a limit (`hnsw.max_scan_tuples`) is reached.

```
BEGIN;
SET LOCAL hnsw.iterative_scan = relaxed_order;
SELECT id FROM chunks
WHERE metadata->>'tenant' = 'acme'
ORDER BY embedding <=> $1
LIMIT 10;
COMMIT;
```

> **Gotcha:** A short result is not an error. The query silently returns fewer than `LIMIT` rows, and the model answers from too little context. Log the result count per query and alert when it drops below `LIMIT`.

Hybrid search and reciprocal rank fusion

Signal	Strong at
Vector similarity	Paraphrases, synonyms, vague questions
Full-text (`tsvector`, BM25)	Exact terms: IDs, error codes, names
Reciprocal rank fusion	Merging both without tuning score scales

Embeddings find meaning but miss exact strings — a query for error `TS-999` returns chunks about errors in general. Keyword search finds the string but misses paraphrases. Run both, then fuse the two ranked lists. Reciprocal rank fusion adds `1 / (k + rank)` for each list a result appears in, with `k = 60` from the original paper; it needs only ranks, so the two incompatible score scales never meet.

```
def rrf(*rankings, k=60):
    score = {}
    for ids in rankings:
        for rank, doc in enumerate(ids, 1):
            prev = score.get(doc, 0)
            score[doc] = prev + 1 / (k + rank)
```

```
    return sorted(score, key=score.get,
                  reverse=True)


vector_ids = [7, 3, 9]
keyword_ids = [3, 5, 7]
rrf(vector_ids, keyword_ids)  # [3, 7, 5, 9]
```

> **Tip:** Fetch more candidates from each list than you will return — 50 each for a final 10 — so a result ranked low by one method can still win on the other.

Reranking

Stage	Anthropic's setup
First-stage retrieval	Top 150 chunks
Rerank	Score each chunk against the query
Keep	Top 20 into the prompt

Embedding search compares a query vector with chunk vectors computed in advance, which is fast but coarse. A reranker (a cross-encoder) reads the query and each candidate together and scores relevance directly, which is more accurate but costs a model call per candidate — so run it only on the first-stage shortlist. In Anthropic's tests, contextual embeddings plus contextual BM25 cut top-20 retrieval failures by 49 percent, and adding reranking by 67 percent.

```
def retrieve(query, search, rerank, keep=20):
    cands = search(query, limit=150)
    texts = [c["text"] for c in cands]
    scores = rerank(query, texts)
    order = sorted(range(len(cands)),
                   key=lambda i: -scores[i])
    return [cands[i] for i in order[:keep]]
```

> **Tip:** Reranking adds latency on every query. Measure recall and precision with and without it on your eval set before shipping; fewer, better chunks can also lower generation cost.

Further reading

- Anthropic — Introducing contextual retrieval
- Claude Docs — Embeddings
- pgvector README
- Reciprocal rank fusion paper (PDF)
- PostgreSQL — Ranking search results