Engineer

Vector search

finding stored items whose vectors lie closest to a query

What it is

Vector search finds the stored items whose vectors lie closest to the vector of a query. It is also called vector similarity search. A vector here is a fixed-length list of numbers that an embedding model produces from a piece of text, an image or a product record. The model is trained so that items with similar content come out with nearby vectors, which means a question and the passage that answers it can sit close together even when they share no words.

“Closest” needs a measure. pgvector, the PostgreSQL extension this article uses for its examples, supports six: L2 (Euclidean) distance, inner product, cosine distance and L1 distance, plus Hamming and Jaccard distance for binary vectors. The embedding model’s own documentation says which measure its vectors are meant for. For vectors normalised to length 1, cosine distance and inner product put the neighbours in the same order, and pgvector recommends inner product there for the best performance.

There are two ways to answer the question. Exact search compares the query with every stored vector and returns the true nearest neighbours; pgvector does this by default, and its documentation describes it as perfect recall. Approximate search can use an index that visits only part of the data, trading some recall for speed. How many true neighbours come back depends on the data, the index and its settings; even a majority is not guaranteed. That is why pgvector warns that, unlike an ordinary index, an approximate index changes what a query returns.

pgvector offers two approximate index types. HNSW builds a graph in several layers, a method Malkov and Yashunin first posted in 2016: each layer holds a subset of the layer below, a search starts in the small top layer and walks down

to the full one, and their paper reports logarithmic scaling of the search cost. IVFFlat divides the vectors into lists and searches only the lists nearest the query.

Use the concept in a decision

Put “What is vector search?” to work in a real decision. We can help relate the concept to the system you are building or buying. Your records where AI agents can reach them, one version of each customer, nobody shown more than they should, and every answer traced to its source.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Made a geographical data pipeline 50x faster — 50x faster geographical data pipelines

Made a geographical data pipeline 50x faster Get in touch →

Why it matters

Full-text search matches words. With word matching and stemming alone, a customer who writes “the screen stays black” does not find the manual section headed “display does not power on”, because the two share no word a stemmer can fold together. Configured synonyms can bridge some such gaps. Vector search compares meaning as the embedding model captured it, so the complaint and the section can land close together. The same idea works beyond text: an image like this one, a support ticket like one already solved, a product like the one in the basket.

It is also one way to do the retrieval in retrieval-augmented generation, where a large language model answers from passages fetched for the question. The question is embedded with the same model as the passages, and the nearest passages go into the prompt. The answer can only be as good as what that retrieval brought back, so the search deserves the same care as the model.

Keeping the vectors in the database you already run has a practical side, and pgvector opens its documentation with that argument. The vectors get the same transactions, point-in-time recovery and joins as every other column, and because pgvector writes to the write-ahead log, replication carries them too. A filter on customer, date or permission is an ordinary `WHERE` clause on the same row. A snapshot taken after a deletion commits no longer sees that document, but a query using an older snapshot may still see it. A separate vector store is a second system to fill and keep in step with the first.

A team should also know what vector search does not do. It ranks by closeness, not by truth. The nearest passage is the one most like the question; it may still be outdated, or wrong, or from a document the reader may not see. Some questions turn on an exact word, such as a name, a part number or an error code, and pgvector's documentation suggests using vector search together with PostgreSQL's full-text search. That combination is called hybrid search.

What it looks like in practice

pgvector, at version 0.8.7 in October 2026, supports PostgreSQL 13 and later. Enable the extension once in each database, then give the table a column of type `vector` with the number of dimensions the model produces:

```
CREATE EXTENSION vector;

CREATE TABLE passages (
    id        bigserial PRIMARY KEY,
    source    text NOT NULL,
    body      text NOT NULL,
    embedding vector(3)
);

CREATE INDEX ON passages USING hnsw (embedding vector_cosine_ops);
```

Three dimensions keep the example readable; a real model produces far more, and the column declares the number it produces. The application computes each embedding with the model and inserts it next to the text it came from. The index names one distance: `vector_cosine_ops` serves queries ordered by cosine distance, and a query ordered by L2 distance needs a second index built with `vector_l2_ops`. HNSW has no training step, so the index can be created on an empty table.

The query embeds the question with the same model and asks for the nearest passages:

```
SELECT id, source, body
FROM   passages
ORDER  BY embedding <=> $1
LIMIT  5;
```

`<=>` is cosine distance and `$1` is the query vector, passed as a parameter. Two details let the planner use the index: the query has both `ORDER BY` and `LIMIT`, and the `ORDER BY` is the bare distance operator in ascending order. A similarity score for display, `1 - (embedding <=> $1)`, belongs in the select list. The other operators are `<->` for L2 distance, `<+>` for L1 and `<#>` for inner product. That last one returns the negative inner product, because PostgreSQL scans an index on an operator in ascending order only.

The HNSW index has two build parameters: `m`, how many neighbours a node links to on each layer (16 unless set), and `ef_construction`, the size of the candidate list while building, 64 by default. At query time, `hnsw.ef_search` sets the size of the candidate list, 40 by default. Raising it finds more of the true nearest neighbours and makes each query slower, and `SET LOCAL` raises it until the end of the current transaction. To limit the setting to one query, put just that query in the transaction. To learn what the index costs in recall, run the same query in a transaction with `SET LOCAL enable_indexscan = off`, which gives the exact answer, and compare the two lists.

The company's work on custom AI agents keeps an agent's memory of past cases in the client's own database rather than in a vendor's chat history. A table like `passages` is one shape such memory can take: when a new case arrives, the past cases most like it are one query away, in the same database and under the same permissions as the rest of the data.

Hybrid search runs both searches and merges the two lists. pgvector's documentation names Reciprocal Rank Fusion, from a 2009 paper by Cormack, Clarke and Büttcher: every document scores 1 / (k + rank) for each list it appears in, and the scores are added. The authors fixed k at 60, and their pilot runs showed that 60 was "near-optimal, but that the choice was not critical". Given a `tsvector` column `search_vec` on the same table, built as for full-text search, the fused query reads:

```
WITH semantic AS (
    SELECT id, row_number() OVER (ORDER BY dist) AS r
    FROM  (SELECT id, embedding <=> $1 AS dist
           FROM   passages
           ORDER  BY dist
           LIMIT  20) AS nearest
),
keyword AS (
    SELECT id, row_number() OVER (ORDER BY score DESC) AS r
    FROM  (SELECT id, ts_rank_cd(search_vec, query) AS score
```

```
           FROM   passages, plainto_tsquery('english', $2) AS query
           WHERE  search_vec @@ query
           ORDER  BY score DESC
           LIMIT  20) AS matches
)
SELECT id,
       coalesce(1.0 / (60 + s.r), 0)
     + coalesce(1.0 / (60 + k.r), 0) AS rrf
FROM   semantic AS s
FULL   JOIN keyword AS k USING (id)
ORDER  BY rrf DESC
LIMIT  5;
```

Each search keeps its own top 20 and numbers them from 1. A passage that both searches place high collects two shares and rises to the top; one found by only one search keeps one share. The fusion uses ranks rather than scores because a cosine distance and a text rank sit on scales that cannot be compared. pgvector names a cross-encoder as the other way to combine the lists.

Where it goes wrong

Fewer rows than `LIMIT` asked for. With an approximate index, the `WHERE` clause is applied after the index scan. pgvector gives the arithmetic: if a condition matches 10% of rows, an HNSW index with the default `hnsw.ef_search` of 40 leaves about 4 rows on average. Since version 0.8.0, iterative index scans scan more of the index until enough rows pass the filter or scan limits are reached. Turn them on with `SET hnsw.iterative_scan = strict_order`, or `relaxed_order` for better recall with results slightly out of order. For HNSW, `hnsw.max_scan_tuples`, 20,000 by default, is an approximate limit on additional tuples visited and does not affect the initial scan. A separate memory limit can also stop further scanning. For a filter with only a few distinct values, a partial index per value also works.

A query misses the nearest-neighbour index. `ORDER BY 1 - (embedding <=> $1) DESC` sorts by an expression, and a query without `LIMIT` does not request a bounded nearest-neighbour result. Neither meets pgvector's requirements for using that index. How many rows the chosen plan reads also depends on filters and other planner choices. An index built with `vector_l2_ops` does nothing for a query ordered by `<=>`. Order by the bare operator, ascending, with a `LIMIT`; build one index per distance you query; and check the plan with `EXPLAIN (ANALYZE, BUFFERS)`.

An IVFFlat index built too early. IVFFlat runs k-means over the rows present when the index is built and divides the vectors into lists from the result. An index created with too little data for its number of lists returns fewer rows than the query asked for, and pgvector's advice is to drop it until the table has more data. Load the data first, then build. pgvector suggests starting with rows / 1000 lists up to a million rows and the square root of the row count above that, and probes, 1 by default, at the square root of the number of lists. HNSW has no such training step.

Rows that are never found. NULL vectors are not indexed, and neither are zero vectors under cosine distance, which has no direction to compare. A row whose embedding is still missing is invisible to an indexed search. Add `NOT NULL` once loading is reliable, and run a scheduled count of rows that still lack an embedding.

Vectors from two models in one column. Two embedding models, or two versions of one, place the same text in different spots, and their vectors cannot be compared even when the dimension counts agree. Embed the question with the model that embedded the passages, store the model in a column, and re-embed everything when the model changes. For a column that holds vectors of several sizes, pgvector shows an expression index with a condition per model, `WHERE (model_id = 123)` .

Too many dimensions for the index. The `vector` type stores up to 16,000 dimensions, but HNSW and IVFFlat index at most 2,000. Index a `halfvec` cast for up to 4,000, use binary quantization for up to 64,000 and re-rank the shortlist by the full vectors, or index a subvector when the model supports it.

One customer's vectors slowing another's. pgvector warns that an approximate index shared between tenants lets one tenant's vectors affect recall and speed for the others. Partition the table by customer with list partitioning, or give each customer its own table.

An index build that blocks the application. HNSW builds much faster when the graph fits in `maintenance_work_mem` , and pgvector prints a notice when it no longer does; setting that memory so high that the server runs short is its own outage. On a live system, build with `CREATE INDEX CONCURRENTLY` , which does not block writes.

Passages the reader may not see. A search over all passages can hand a language model text from a document the user has no right to read, and the model can quote it. Put the permission condition in the same `WHERE` clause as the search, so such a row never leaves the database, and turn on iterative scans to look for more permitted passages when filtering removes candidates, subject to the scan limits.

A distance read as confidence. A cosine distance of 0.2 says a passage is nearer than one at 0.4, not that it answers the question. A threshold tuned for one model means nothing for the next. Order by distance, cap with `LIMIT`, and judge the search by a set of real questions whose right passages are known.

Further reading

- pgvector — the extension's own documentation: types, distance operators, HNSW and IVFFlat options, filtering, iterative index scans, hybrid search and troubleshooting.
- Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs — Malkov and Yashunin's paper that introduced HNSW.
- Reciprocal Rank Fusion outperforms Condorcet and individual Rank Learning Methods — the SIGIR 2009 paper with the fusion formula and the pilot runs behind its constant.
- Introduction to full text search — PostgreSQL's word-based search, the other half of a hybrid query.
- Controlling Text Search — `plainto_tsquery` and `ts_rank_cd`, as used in the keyword half above.
- Transaction Isolation — when PostgreSQL snapshots see committed changes, including why an older snapshot may still see a document after its deletion commits.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is vector search?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/vector-search/