Engineer

Vektorsökning

sökning efter de vektorer som ligger närmast en fråga

Vad det är

Vektorsökning hittar de lagrade objekt vars vektorer ligger närmast sökningens vektor. På engelska heter det vector search. En vektor är här en lista med tal av fast längd som en embeddingmodell räknar fram ur ett textstycke, en bild eller en produktpost. Modellen är tränad så att objekt med liknande innehåll får vektorer som ligger nära varandra, och därför kan en fråga och stycket som besvarar den hamna tätt intill varandra även när de inte har ett enda ord gemensamt.

”Närmast” kräver ett mått. pgvector, det PostgreSQL-tillägg som exemplen i den här artikeln använder, stöder sex: L2-avstånd (euklidiskt), skalärprodukt, cosinusavstånd och L1-avstånd, plus Hammingavstånd och Jaccardavstånd för binära vektorer. Embeddingmodellens egen dokumentation anger vilket mått vektorerna är avsedda för. För vektorer som är normaliserade till längden 1 ger cosinusavstånd och skalärprodukt grannarna samma ordning, och där rekommenderar pgvector skalärprodukt eftersom den ger bäst prestanda.

Det finns två sätt att besvara sökningen. Exakt sökning jämför sökningen med varenda lagrad vektor och returnerar de verkliga närmaste grannarna. Det är vad pgvector gör som standard, och dokumentationen kallar det perfekt täckning (recall). Ungefärlig sökning kan använda ett index som bara besöker en del av datan och byter bort lite täckning mot fart. Hur många verkliga grannar som kommer med beror på datan, indexet och dess inställningar; inte ens en majoritet är garanterad. Därför varnar pgvector för att ett ungefärligt index, till skillnad från ett vanligt index, ändrar vad en fråga returnerar.

pgvector har två typer av ungefärliga index. HNSW bygger en graf i flera lager, en metod som Malkov och Yashunin först publicerade 2016. Varje lager innehåller ett urval av noderna i lagret under, en sökning börjar i det lilla översta lagret och

går nedåt till det fullständiga, och enligt deras artikel växer sökkostnaden logaritmiskt. IVFFlat delar in vektorerna i listor och söker bara i de listor som ligger närmast sökningen.

Använd begreppet i ett beslut

Använd ”Vad är vektorsökning?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Dina data samlade där AI-agenter kan nå dem, med en enda version av varje kund, utan att någon ser mer än de får, och med varje svar spårat tillbaka till källan.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Gjorde pipelinen för geodata 50 gånger snabbare — 50x snabbare datapipelines för geodata

Gjorde pipelinen för geodata 50 gånger snabbare Hör av er →

Varför det spelar roll

Fulltextsökning jämför ord. Med enbart ordmatchning och stemming hittar en kund som skriver ”skärmen förblir svart” inte avsnittet i handboken med rubriken ”displayen startar inte”, eftersom de två inte delar något ord som stemmingen kan föra samman. Konfigurerade synonymer kan överbrygga några sådana skillnader. Vektorsökning jämför betydelse så som embeddingmodellen har fångat den, så klagomålet och avsnittet kan hamna nära varandra. Samma idé fungerar bortom text: en bild som liknar den här, ett supportärende som liknar ett som redan är löst, en produkt som liknar den som ligger i varukorgen.

Det är också ett sätt att sköta hämtningen i retrieval-augmented generation, där en stor språkmodell svarar utifrån textstycken som hämtats för frågan. Frågan görs om till en vektor med samma modell som styckena, och de närmaste styckena läggs in i prompten. Svaret blir aldrig bättre än det sökningen fick med sig, så sökningen förtjänar lika mycket omsorg som modellen.

Att ha vektorerna i den databas ni redan driver har en praktisk sida, och pgvector inleder sin dokumentation med just det argumentet. Vektorerna får samma transaktioner, samma återställning till en viss tidpunkt (point-in-time recovery) och samma joinar som alla andra kolumner, och eftersom pgvector skriver till write-ahead-loggen (WAL) följer de med i replikeringen också. Ett filter på kund, datum eller behörighet är ett vanligt `WHERE` -villkor på samma rad. En ögonblicksbild som tas efter att en radering har bekräftats ser inte längre dokumentet, men en fråga med en äldre ögonblicksbild kan fortfarande se det. Ett separat vektorlager är ett system till som måste fyllas och hållas i takt med det första.

Ett team bör också veta vad vektorsökning inte gör. Den rangordnar efter närhet, inte efter sanning. Det närmaste stycket är det som mest liknar frågan; det kan ändå vara inaktuellt, fel eller hämtat ur ett dokument som läsaren inte får se. Vissa frågor hänger på ett exakt ord, som ett namn, ett artikelnummer eller en felkod, och pgvectors dokumentation föreslår att vektorsökning används tillsammans med fulltextsökningen i PostgreSQL. Den kombinationen kallas hybridsökning.

Så ser det ut i praktiken

pgvector, i version 0.8.7 i oktober 2026, stöder PostgreSQL 13 och senare. Tillägget aktiveras en gång i varje databas, och sedan får tabellen en kolumn av typen `vector` med det antal dimensioner som modellen ger:

```
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);
```

Tre dimensioner håller exemplet läsbart; en verklig modell ger betydligt fler, och kolumnen anger det antal som modellen ger. Applikationen räknar fram varje embedding med modellen och lägger in den bredvid texten den kom från. Indexet gäller ett enda avstånd: `vector_cosine_ops` betjänar frågor som sorteras efter cosinusavstånd, och en fråga som sorteras efter L2-avstånd behöver ett index till, byggt med `vector_l2_ops`. HNSW har inget träningssteg, så indexet kan skapas på en tabell utan rader.

Vid sökning görs frågan om till en vektor med samma modell, och SQL-frågan ber om de närmaste styckena:

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

`<=>` är cosinusavstånd, och `$1` är sökvektorn, skickad som parameter. Två detaljer gör att planeraren kan använda indexet: frågan har både `ORDER BY` och `LIMIT`, och `ORDER BY` står på själva avståndsoperatorn i stigande ordning. Ett likhetsvärde att visa, `1 - (embedding <=> $1)`, hör hemma i `SELECT`-listan. De andra operatorerna är `<->` för L2-avstånd, `<+>` för L1 och `<#>` för skalärprodukt. Den sista returnerar den negativa skalärprodukten, eftersom PostgreSQL bara skannar ett index på en operator i stigande ordning.

HNSW-indexet har två byggparametrar: `m`, det största antalet förbindelser per lager, 16 som standard, och `ef_construction`, storleken på kandidatlistan under bygget, 64 som standard. Vid sökning sätter `hnsw.ef_search` storleken på kandidatlistan, 40 som standard. Ett högre värde ger bättre täckning på bekostnad av fart, och `SET LOCAL` höjer det fram till slutet av den aktuella transaktionen. För att begränsa inställningen till en fråga ska transaktionen bara innehålla den frågan. Vill du veta hur mycket täckning indexet kostar kör du samma fråga i en transaktion med `SET LOCAL enable_indexscan = off`, som ger det exakta svaret, och jämför de två listorna.

Företagets arbete med skräddarsydda AI-agenter lägger agentens minne av tidigare ärenden i kundens egen databas, inte i en leverantörs chatthistorik. En tabell som `passages` är en form ett sådant minne kan ta: när ett nytt ärende kommer in ligger de tidigare ärenden som liknar det mest en fråga bort, i samma databas och under samma behörigheter som resten av datan.

Hybridsökning kör båda sökningarna och slår ihop de två listorna. pgvectors dokumentation nämner Reciprocal Rank Fusion, från en artikel av Cormack, Clarke och Büttcher från 2009: varje dokument får 1 / (k + rang) för varje lista det finns med i, och andelarna läggs ihop. Författarna satte k till 60, och deras provkörningar visade att 60 var ”near-optimal, but that the choice was not critical”. Med en `tsvector`-kolumn `search_vec` i samma tabell, byggd som för fulltextsökning, ser den sammanslagna frågan ut så här:

```
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('swedish', $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;
```

Varje sökning behåller sina egna 20 bästa och numrerar dem från 1. Ett stycke som båda sökningarna placerar högt får två andelar och stiger till toppen; ett stycke som bara en sökning hittade behåller en. Sammanslagningen använder rangordning i stället för poäng, eftersom ett cosinusavstånd och en textrang ligger på skalor som inte går att jämföra. Språket i `plainto_tsquery` måste vara detsamma som i kolumnen, här `swedish` . pgvector nämner en cross-encoder som det andra sättet att kombinera listorna.

Var det går fel

Färre rader än `LIMIT` bad om. Med ett ungefärligt index tillämpas `WHERE` -villkoret först efter att indexet har skannats. pgvector visar räkneexemplet: om ett villkor träffar 10 % av raderna ger ett HNSW-index med standardvärdet 40 för `hnsw.ef_search` i genomsnitt bara omkring 4 rader. Från och med version 0.8.0 skannar iterativa indexskanningar mer av indexet tills tillräckligt många rader har passerat filtret eller skanningsgränserna har nåtts. Slå på dem med `SET hnsw.iterative_scan = strict_order` , eller `relaxed_order` för bättre täckning med resultat som kan stå något i fel ordning. För HNSW är `hnsw.max_scan_tuples` , 20 000 som standard, en ungefärlig gräns för ytterligare besökta tupler och påverkar inte den första skanningen. En separat minnesgräns kan också stoppa fortsatt skanning. För ett filter med bara ett fåtal olika värden fungerar också ett partiellt index per värde.

En fråga som missar indexet för närmaste grannar. `ORDER BY 1 - (embedding <=> $1) DESC` sorterar efter ett uttryck, och en fråga utan `LIMIT` begär inte ett avgränsat resultat med de närmaste grannarna. Inget av fallen uppfyller pgvectors krav för att använda det indexet. Hur många rader den valda planen läser beror också på filter och andra val som planeraren gör. Ett index byggt med

`vector_l2_ops` hjälper inte en fråga som sorteras efter `<=>`. Sortera efter själva operatorn, stigande, med `LIMIT`; bygg ett index per avstånd du söker med; och kontrollera planen med `EXPLAIN (ANALYZE, BUFFERS)`.

Ett IVFFlat-index byggt för tidigt. IVFFlat kör k-means över de rader som finns när indexet byggs och delar in vektorerna i listor utifrån resultatet. Ett index som skapats med för lite data för sitt antal listor returnerar färre rader än frågan bad om, och pgvector råder dig att ta bort det tills tabellen har fått mer data. Läs in datan först och bygg sedan. pgvector föreslår att börja med rader / 1000 listor upp till en miljon rader och kvadratroten ur antalet rader därutöver, och att sätta probes, 1 som standard, till kvadratroten ur antalet listor. HNSW har inget sådant träningssteg.

Rader som aldrig hittas. NULL-vektorer indexeras inte, och det gör inte heller nollvektorer vid cosinusavstånd, eftersom de saknar en riktning att jämföra. En rad som saknar embedding är osynlig för en sökning via indexet. Lägg till `NOT NULL` när inläsningen är pålitlig, och kör en schemalagd räkning av rader som fortfarande saknar embedding.

Vektorer från två modeller i en kolumn. Två embeddingmodeller, eller två versioner av samma modell, placerar samma text på olika ställen, och deras vektorer går inte att jämföra ens när antalet dimensioner stämmer. Gör om frågan till en vektor med den modell som tog fram vektorerna för styckena, spara modellen i en kolumn och räkna om alla vektorer när modellen byts. För en kolumn med vektorer av flera storlekar visar pgvector ett uttrycksindex med ett villkor per modell, `WHERE (model_id = 123)`.

För många dimensioner för indexet. Typen `vector` lagrar upp till 16 000 dimensioner, men HNSW och IVFFlat indexerar högst 2 000. Indexera en `halfvec`-konvertering för att nå upp till 4 000, använd binär kvantisering för upp till 64 000 och rangordna kortlistan på nytt med de fullständiga vektorerna, eller indexera en delvektor när modellen stöder det.

En kunds vektorer saktar ner en annans. pgvector varnar för att när flera kunder delar ett ungefärligt index kan vektorerna hos en av dem påverka täckning och fart för de andra. Partitionera tabellen per kund med listpartitionering, eller ge varje kund en egen tabell.

Ett indexbygge som blockerar applikationen. HNSW byggs mycket snabbare när grafen får plats i `maintenance_work_mem`, och pgvector skriver ut ett meddelande när den inte längre gör det; att sätta det minnet så högt att servern får för lite minne orsakar ett eget driftstopp. På ett system i drift bygger du med `CREATE INDEX CONCURRENTLY`, som inte blockerar skrivningar.

Stycken som läsaren inte får se. En sökning över alla stycken kan ge en språkmodell text ur ett dokument som användaren inte har rätt att läsa, och modellen kan citera den. Lägg behörighetsvillkoret i samma `WHERE`-villkor som sökningen, så att en sådan rad aldrig lämnar databasen, och slå på iterativa skanningar för att leta efter fler tillåtna stycken när filtreringen tar bort kandidater, inom skanningsgränserna.

Ett avstånd som tolkas som visshet. Ett cosinusavstånd på 0,2 säger att ett stycke ligger närmare än ett på 0,4, inte att det besvarar frågan. En tröskel som är inställd för en modell betyder ingenting för nästa. Sortera efter avstånd, begränsa med `LIMIT` och bedöm sökningen mot en uppsättning verkliga frågor där de rätta styckena är kända.

Läs mer

- pgvector — tilläggets egen dokumentation: typer, avståndsoperatorer, inställningar för HNSW och IVFFlat, filtrering, iterativa indexskanningar, hybridsökning och felsökning.
- Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs — artikeln av Malkov och Yashunin som introducerade HNSW.
- Reciprocal Rank Fusion outperforms Condorcet and individual Rank Learning Methods — SIGIR-artikeln från 2009 med formeln för sammanslagningen och provkörningarna bakom dess konstant.
- Introduction to full text search — ordbaserad sökning i PostgreSQL, den andra halvan av en hybridfråga.
- Controlling Text Search — `plainto_tsquery` och `ts_rank_cd`, så som de används i ordsökningen ovan.
- Transaction Isolation — när PostgreSQLs ögonblicksbilder ser bekräftade ändringar, inklusive varför en äldre ögonblicksbild fortfarande kan se ett dokument efter att dess radering har bekräftats.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med ett representativt dataset, frågorna det ska besvara och dagens flaskhals. Vi kan avtala mått på kvalitet, genomströmning och återställning.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är vektorsökning?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/vector-search/