Embedding Workloads as a New Stress Test for Postgres

postgresql.istanbul 2026 | Istanbul, Turkey

Viktoriia Hrechukha

What This Talk Is

- **Not a war story**: I am not a DBA. No production pgvector battle scars.
- **A research synthesis**: Weeks reading GitHub, Reddit, HN, engineering blogs.
- **100+ sources**: Real issues, real production reports, real numbers.
- **Six categories of pain**: So you know what to watch for as you scale.

01

Why Embeddings Are a Worst Case for Postgres

Large rows . Frequent updates . Graph indexes

What Makes Embeddings Different

- **Large rows**: Each vector fills a full 8KB page on disk.
- **Frequent updates**: Re-chunk, re-embed, switch models. Constant rewrites.
- **Graph indexes**: HNSW is not a B-tree. Every write is graph surgery.

Large Rows: 8KB per Vector

1 vector(1536)

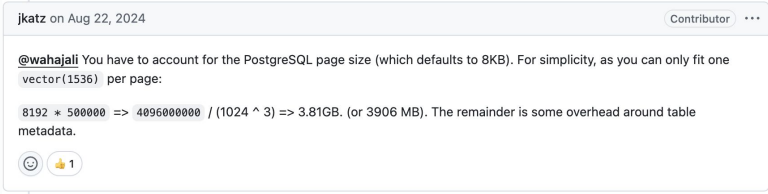
1 full 8KB page

Expected

2.86 GB

Actual

3.81 GB



SOURCE: GitHub: pgvector/pgvector Issue #653 - Table size vs actual size: https://github.com/pgvector/pgvector/issues/653

The Storage Adds Up Fast

1M documents

6 GB

raw vectors

+ HNSW index

12-18 GB

total on disk

the takeaway

x2-3

vs raw data

Frequent Updates Create Dead Rows

UPDATE does not delete
copies the row, marks the old one dead

Normal data
100 bytes dead, nobody notices

Vector data
6 KB dead per row, 500K updates = 3 GB

Non-vector UPDATEs slow too
with an HNSW index present

plagree opened on Jul 20, 2025 · edited by plagree — Edits

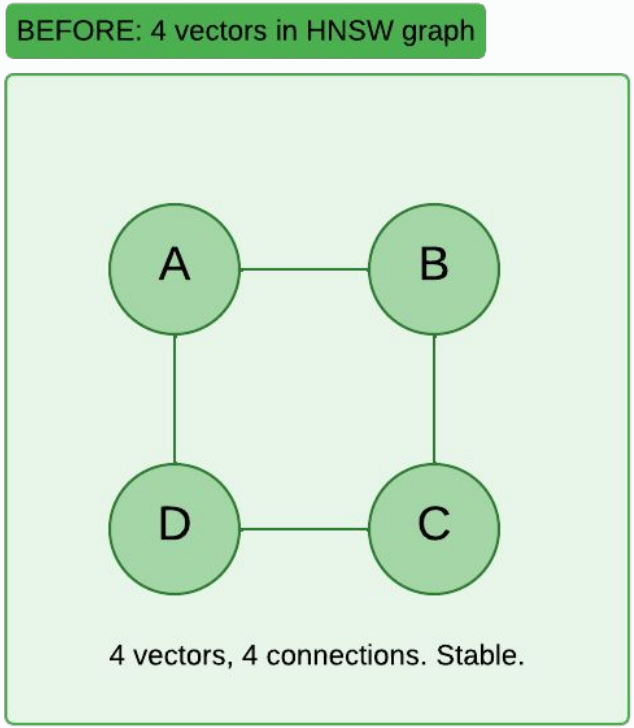
I'm experiencing severe performance issues when updating non-vector columns in tables that have HNSW vector indexes. The UPDATE statements become (very) much slower even though the vector column itself is not being modified.

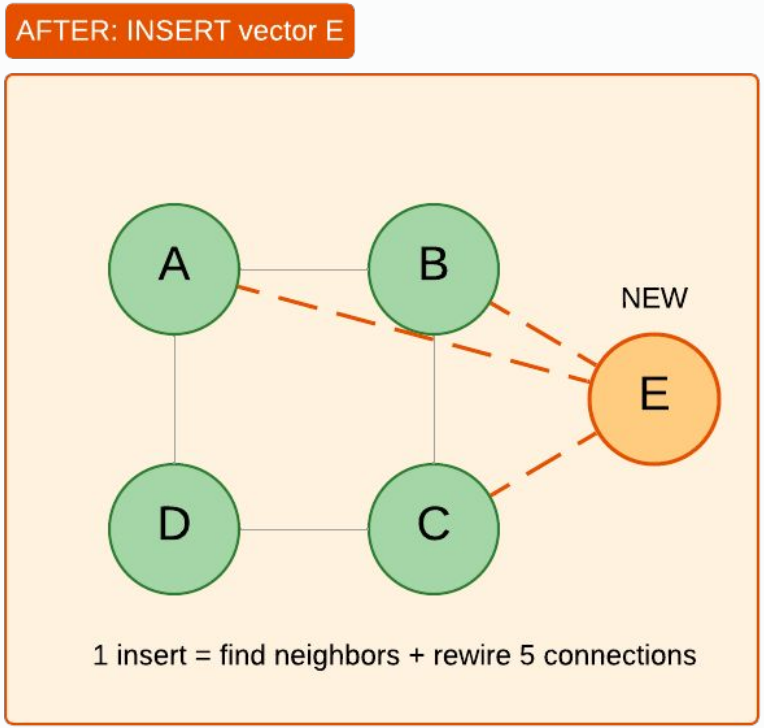
Minimal reproduction:

```
-- Table with HNSW index
CREATE TABLE test_table (
    id SERIAL PRIMARY KEY,
    name TEXT,
    embedding vector(1536)
);

INSERT INTO test_table (name, embedding)
SELECT 'item_' || i, array_fill(random(), ARRAY[1536])::vector(1536)
FROM generate_series(1, 10000) i;

SET maintenance_work_mem = '256MB';

CREATE INDEX test_embedding_idx ON test_table
USING hnsw (embedding vector_cosine_ops);

-- This UPDATE is extremely slow despite not touching the vector column
UPDATE test_table SET name = 'updated';
```

Source: pgvector Issue #875

B-tree vs Vector Index

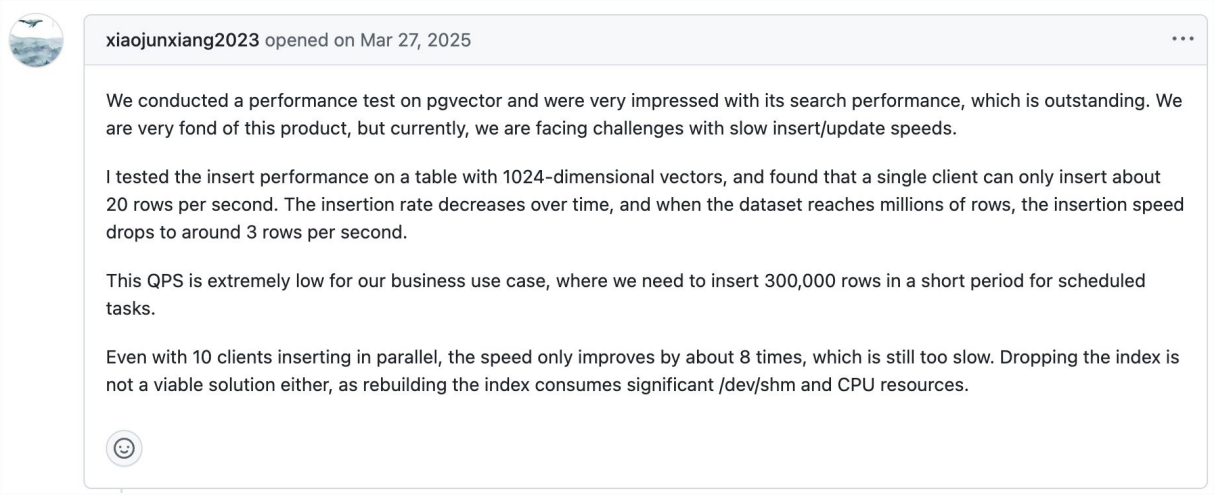




Insert Rates Collapse at Scale

3

rows / second



xiaojunxiang2023 opened on Mar 27, 2025

We conducted a performance test on pgvector and were very impressed with its search performance, which is outstanding. We are very fond of this product, but currently, we are facing challenges with slow insert/update speeds.

I tested the insert performance on a table with 1024-dimensional vectors, and found that a single client can only insert about 20 rows per second. The insertion rate decreases over time, and when the dataset reaches millions of rows, the insertion speed drops to around 3 rows per second.

This QPS is extremely low for our business use case, where we need to insert 300,000 rows in a short period for scheduled tasks.

Even with 10 clients inserting in parallel, the speed only improves by about 8 times, which is still too slow. Dropping the index is not a viable solution either, as rebuilding the index consumes significant /dev/shm and CPU resources.

Source: pgvector Issue #810

02

Storage Bloat: The Invisible Cost

Your table is logically clean but physically enormous

Where Did the Extra 6 GB Come From?

Load 1M vectors

6 GB

Update half

12 GB

Still 1M rows

???

Same row count. Double the size.

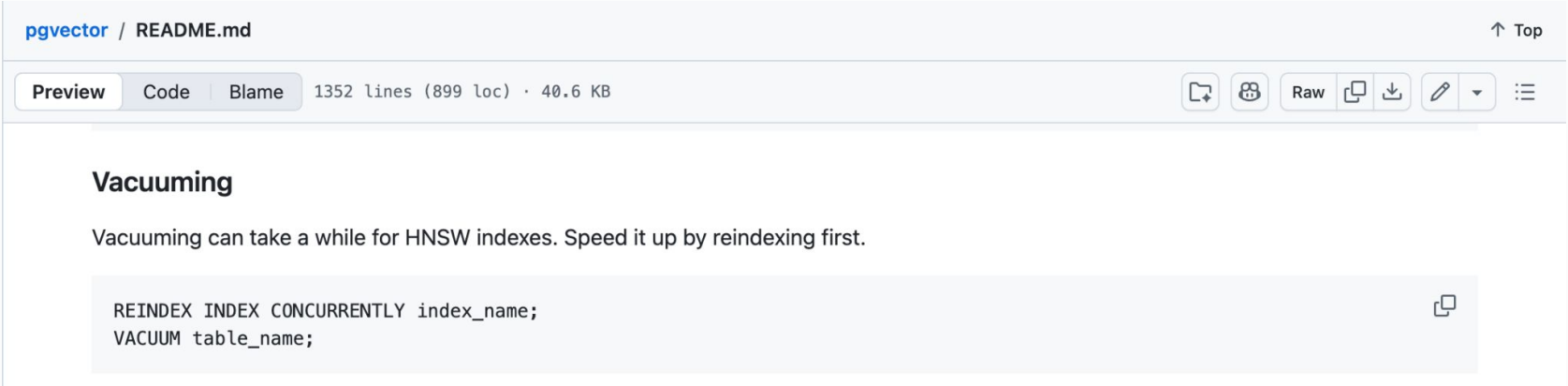
Dead Rows: Postgres' Hidden Growth

- **UPDATE copies**: Old version stays as a dead row.
- **Autovacuum marks reusable**: But space is not returned to the OS.
- **The file stays the same size**: Your disk bill does not go down.

"Waste in bytes is still too high"

- **Scale factor**: 0.2 to 0.05
- **Dead rows**: 200,000 to 0
- **Physical table size**: barely changed
- **Only fix: VACUUM FULL**: locks the table. Hours of downtime.

Even pgvector Admits It



SOURCE: pgvector README (vacuum note): https://github.com/pgvector/pgvector/blob/master/README.md
SOURCE: GitHub: pgvector/pgvector Issue #692 - Memory leak in vacuum for HNSW: https://github.com/pgvector/pgvector/issues/692

03

HNSW Indexes: Not Fire and Forget

Build times . CVE . Silent recall loss

Building the Index Takes Hours

40M rows

8+ hrs

stuck

10M vectors

24+ hrs

4 CPU machine

5M vectors

OOM

crash in seconds

Source: pgvector Issues #822, #300, #843

Parallel Builds + a Security Bug

Parallel builds (0.6.0)

30x

15 hrs to 30 min (AWS)

CVE-2026-3172

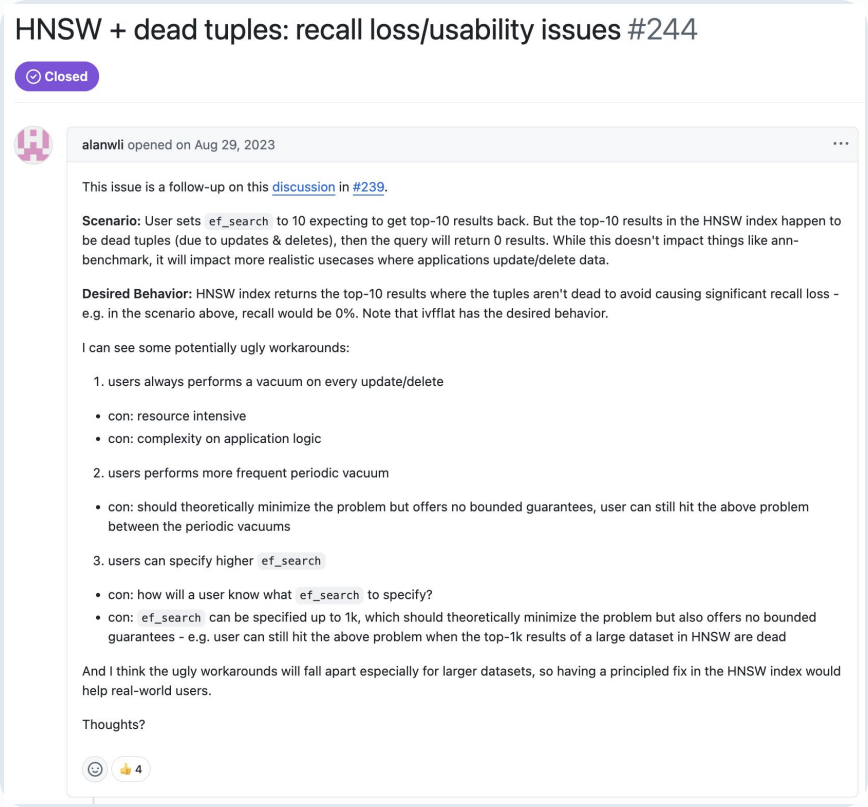
Buffer overflow in parallel index builds.
Affects 0.6.0 - 0.8.1

BUT Fixed in 0.8.2 🥳

Silent Recall Loss

"HNSW returned zero results" - Issue #244

- **Dead rows stay in the graph**
  They point to deleted data.
- **Ask for top 10**
  Get 7. Or 3. Or zero.
- **No error. No warning.**
  Just wrong answers.



HNSW + dead tuples: recall loss/usability issues #244

Closed

alanwli opened on Aug 29, 2023

This issue is a follow-up on this discussion in #239.

Scenario: User sets `ef_search` to 10 expecting to get top-10 results back. But the top-10 results in the HNSW index happen to be dead tuples (due to updates & deletes), then the query will return 0 results. While this doesn't impact things like ann-benchmark, it will impact more realistic usecases where applications update/delete data.

Desired Behavior: HNSW index returns the top-10 results where the tuples aren't dead to avoid causing significant recall loss - e.g. in the scenario above, recall would be 0%. Note that ivfflat has the desired behavior.

I can see some potentially ugly workarounds:

1. users always performs a vacuum on every update/delete
   - con: resource intensive
   - con: complexity on application logic
2. users performs more frequent periodic vacuum
   - con: should theoretically minimize the problem but offers no bounded guarantees, user can still hit the above problem between the periodic vacuums
3. users can specify higher `ef_search`
   - con: how will a user know what `ef_search` to specify?
   - con: `ef_search` can be specified up to 1k, which should theoretically minimize the problem but also offers no bounded guarantees - e.g. user can still hit the above problem when the top-1k results of a large dataset in HNSW are dead

And I think the ugly workarounds will fall apart especially for larger datasets, so having a principled fix in the HNSW index would help real-world users.

Thoughts?

4

Monitor the Dead Row Ratio

```
n_dead_tup / (n_live_tup + n_dead_tup)
```

> 10%

search quality drops

> 15%

critical, recall degrading

04

The Query Planner Was Not Built for Vectors

The 50ms to 5-second jump

One WHERE Clause. 48x Slower.

Vector search only

1.5 ms

index used

+ WHERE is_active

807 ms

seq scan, index ignored

psql ⌥⌘2

```
embeddings=# SET enable_indexscan = off;
  EXPLAIN (ANALYZE, FORMAT TEXT)
  SELECT id, content, embedding <=> (SELECT embedding FROM documents WHERE id = 42) AS distance
  FROM documents
  WHERE is_active = true
  ORDER BY embedding <=> (SELECT embedding FROM documents WHERE id = 42)
  LIMIT 10;
  SET enable_indexscan = on;
SET
                                                             QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=17888.78..17889.97 rows=10 width=31) (actual time=784.678..807.122 rows=10 loops=1)
   InitPlan 1
     ->  Bitmap Heap Scan on documents documents_1  (cost=4.43..8.45 rows=1 width=18) (actual time=0.034..0.035 rows=1 loops=1)
           Recheck Cond: (id = 42)
           Heap Blocks: exact=2
           ->  Bitmap Index Scan on documents_pkey  (cost=0.00..4.43 rows=1 width=0) (actual time=0.023..0.024 rows=2 loops=1)
                 Index Cond: (id = 42)
   ->  Gather Merge  (cost=17880.34..95079.70 rows=652389 width=31) (actual time=784.676..807.116 rows=10 loops=1)
         Workers Planned: 3
         Workers Launched: 3
         ->  Sort  (cost=16880.30..17423.95 rows=217463 width=31) (actual time=755.352..755.358 rows=8 loops=4)
               Sort Key: ((documents.embedding <=> (InitPlan 1).col1))
               Sort Method: top-N heapsort  Memory: 26kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 26kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 26kB
               Worker 2:  Sort Method: top-N heapsort  Memory: 25kB
               ->  Parallel Seq Scan on documents  (cost=0.00..12181.00 rows=217463 width=31) (actual time=2.200..738.020 rows=112484 loops=4)
                     Filter: is_active
                     Rows Removed by Filter: 12516
 Planning Time: 0.202 ms
 Execution Time: 807.178 ms
(21 rows)

SET
embeddings=#
```

This Is the Most Common Problem

8

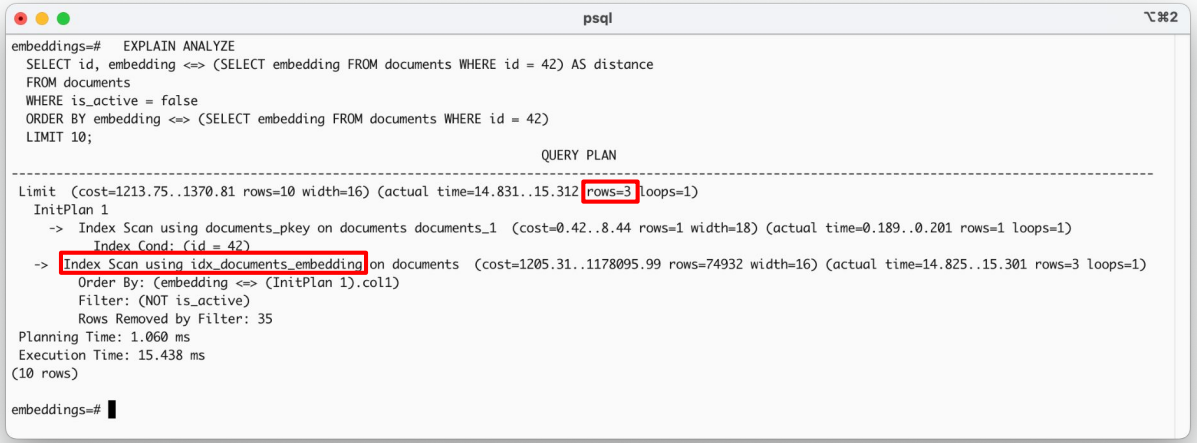
separate GitHub issues

document the planner
ignoring the vector index.

Post-Filtering: Ask for 10, Get 3

Step 1: pgvector finds the 10 nearest vectors.

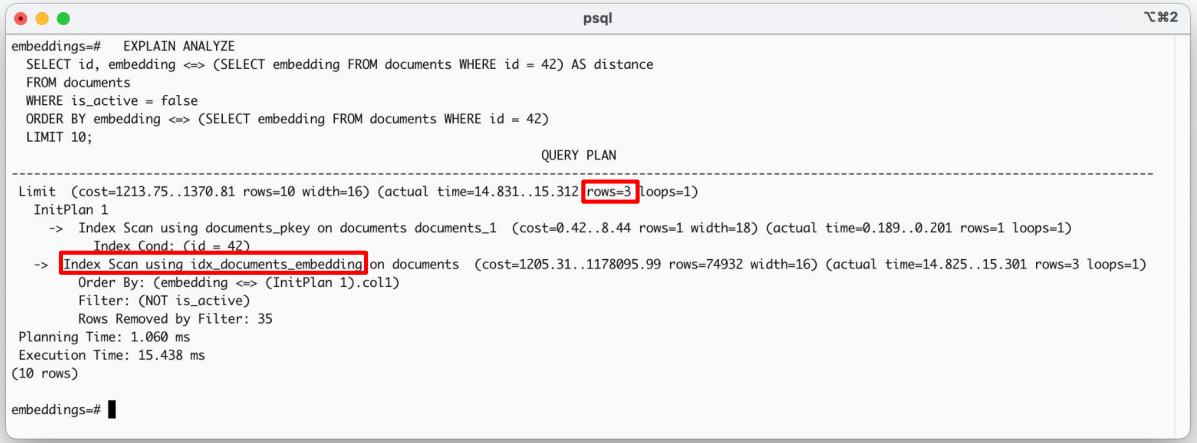
Step 2: Throws away the ones that fail your WHERE.



Post-Filtering: Ask for 10, Get 3

If 3 match, you get 3. Not 10.

Your app asked for 10. Got fewer. No error.



Iterative Scans (0.8.0): a Partial Fix

The Iterative Approach

pgvector 0.8.0 tried to fix this with **iterative scans**. Instead of searching once, it keeps searching until enough results pass the filter.

AWS tested it and found it to be **up to 5.7 times faster** in the best case.

But with strict filters, latency can still jump from 229 milliseconds to over 3 seconds—which is **14 times worse**.

Loose filter

5.7x

faster (best case)

Strict filter

14x

229ms to 3,228ms

05

Model Migrations: The Schema Nightmare

When your embedding model changes, everything breaks

Models Don't Stay the Same

- **They get updated**: New version, new output.
- **They get deprecated**: Overnight, sometimes.
- **They get replaced**: By something better.
- **Everything downstream breaks**: Fixed column dimension, no ALTER.

vector(768) to vector(1024)

01
Add new column

02
Re-embed all docs

03
Build new index

04
Drop old column

Migration Resource Impact Warning
During migration: 2x storage, 2x index memory. For days or weeks.

The Horror Story

"Our embedding model got deprecated overnight.Every RAG query started returning 404s."
— DEV Community

The Hard Ceiling: 2,000 Dimensions

- **HNSW index limit**: 2,000 dimensions (full precision)
- **OpenAI embed-3-large**: 3,072 dims, cannot index
- **Qwen3**: 4,096 dims, cannot index

06

The Real Total Cost of Ownership

"We already have Postgres, so it's free." Let me challenge that.

Infrastructure: pgvector Wins

pgvector + pgvectorscale

$835

per month, 50M vectors

Pinecone

$3.2-5.2K

per month, same workload

But Infrastructure Is Not the Whole Story

- **DBA time**: Tuning autovacuum for vector tables.
- **Index rebuilds**: Scheduled maintenance windows.
- **Cold cache penalties**: After every deploy or failover.
- **Custom hybrid search**: Combining full-text + vector is on you.

The Honest Answer

pgvector

Cheaper in dollars

Dedicated Vector DB

Cheaper in engineering hours

The Survival Checklist

1. **autovacuum_vacuum_scale_factor = 0.01** default 0.2 too slow for vectors
2. **Alert when dead rows > 10%** at 15% quality already dropping
3. **EXPLAIN ANALYZE every WHERE + vector** the planner will surprise you
4. **Upgrade to pgvector 0.8.2** CVE-2026-3172 fix
5. **Name columns embedding_v1** be ready for model changes

07

When pgvector Is Enough (and When It Is Not)

The right tool for the right scale

Decision Framework

< 1M vectors	pgvector. No question.
1M - 10M	pgvector + serious tuning, or evaluate alternatives.
> 10M	Dedicated vector DB. Reddit: 340M to Milvus.

Alternatives

- **Qdrant**: Rust, single binary. Filters during search.
- **Milvus**: Billion-scale. GPU. Distributed.
- **Weaviate**: Best hybrid search (BM25 + vector).
- **ClickHouse**: GA in 25.8. Zero new infra if you run it.
- **pgvectorscale**: DiskANN. Stays in Postgres. Not on RDS.

Thank you!

Questions?