pgvector: Advanced Vector Data Management in PostgreSQL

Agustín Gallego
Lead Database Performance Engineer
Support Team, Percona

PERCONA

Agenda and What is pgvector?

Setup > Types > Embeddings > Similarity > Storage > Indexing > Filtering > Tuning > Upgrades

- Open-source vector similarity search for Postgres
- Store vectors with the rest of your data
- Supports exact + approximate nearest-neighbour search, four vector types, six distance functions
- ACID, PITR, JOINs, replication — everything Postgres already gives us

https://github.com/pgvector/pgvector/blob/master/README.md

PERCONA

Setup & install

- It can be compiled from source:

https://github.com/pgvector/pgvector/blob/master/README.md#linux-and-mac

- But better yet, via a package manager from repos:
  - PGDG (versions 12+)
  - Percona Distribution for PostgreSQL (versions 13+)

```
$ yum install [percona-]pgvector_17
$ apt install [percona-]postgresql-17-pgvector
```

- After it's installed, we can enable it in all the needed databases with:

```
postgres=# CREATE EXTENSION vector;
```


PERCONA

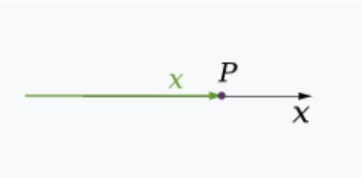
pgvector GUCs

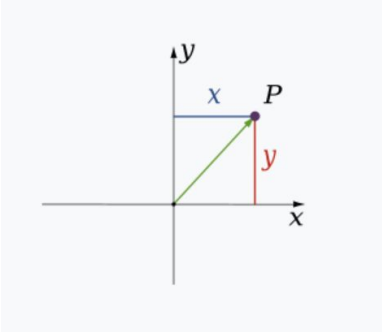
- All 7 user-settable variables available:
  - ivfflat.probes
  - ivfflat.iterative_scan
  - ivfflat.max_probes
  - hnsw.ef_search
  - hnsw.iterative_scan
  - hnsw.max_scan_tuples
  - hnsw.scan_mem_multiplier
- **Note:** to see these in *pg_settings*, add ***vector*** to ***shared_preload_libraries***. Not required for SET to work.

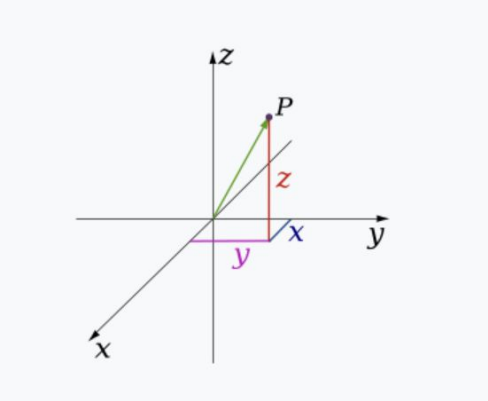
PERCONA

What is a vector?

- A vector has **magnitude** (length) and **direction**
- We group them in **vector spaces** characterised by their **dimension**
  - The dimension is the minimum number of directions needed to fully describe a point in the space
- In a database, a vector is just an array of numbers







PERCONA

Data types, operators & functions

- **Types** (storage / max dim):
  - vector — single precision floating-point number - 4·d + 8 bytes / 16k dims
  - halfvec — half precision floating-point number - 2·d + 8 bytes / 16k dims
  - sparsevec — (same as vector) - 8·nnz + 16 bytes / 16k non-zero
  - bit — d/8 + 8 bytes / 64k dims
- **Element-wise operators:** +, -, *, || (concat)
- **Useful functions:** binary_quantize, l2_normalize, subvector, vector_dims, vector_norm, etc.
- **Full reference:**

https://github.com/pgvector/pgvector/blob/master/README.md#reference

PERCONA

Embeddings

- Embeddings are **vectorial representations** of more complex objects (text, images, audio, etc.)
- They depend on the model that produced them
- Embeddings from different models are **not** interchangeable
- pgvector applies **mathematical functions only** — it does not produce embeddings

Does this hold true?

```
vector("king") - vector("man") + vector("woman") ≈ vector("queen")
```

Similarity search: distance functions

Distances:

- <-> L2 (Euclidean)
- <#> (negative) inner product (negative because PG only does ASC index scans)
- <=> cosine distance
- <+> L1 (taxicab / Manhattan) distance
- <~> Hamming distance (binary vectors)
- <%> Jaccard distance (binary vectors)

Similarities:

- euclidean = *1 / (1 + (a <-> b))*
- inner product (for normalized models) = -1 * (a <#> b)
- cosine (the semantic standard) = *1 - (a <=> b)*

Note: not all distance functions are supported by all data types.

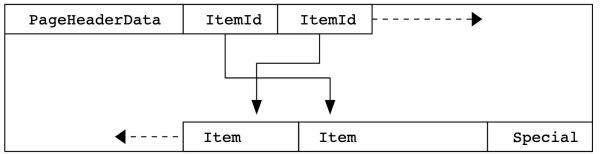
PERCONA

Exact vs Approximate Nearest Neighbors

- Do I need to get the exact results from the whole dataset?
- Exact Nearest Neighbors:
  - Compares every vector in the relation (full table scan)
  - Tuning does **not** change the result set
  - Cost: O(N) per query
- Approximate Nearest Neighbors:
  - Uses an index, doesn't compare everything
  - Introduces **recall**: the percentage of expected results we actually return
  - Tuning **can** change the result set: recall ↔ performance trade

Vector storage: TOAST math

- Postgres storage strategies: PLAIN, EXTENDED, **EXTERNAL** (pgvector default), MAIN
- MAXALIGN = 8 on Linux x86_64
- Page = 8192 bytes. Header 24, line pointer 4, tuple header 23 (24 aligned)
- TOAST triggers above TOAST_TUPLE_THRESHOLD ≈ 2 KB:
  - TOAST_TUPLES_PER_PAGE = 4
  - 8192 - 24 - 4*4 = 8152 ; / 4 = 2038 (2032 aligned); - 24 tuple header = 2008 B
- Dimension limits before TOAST kicks in (single column, no PK):
  - vector: 4·d + 8 ≤ 2008 → **d ≤ 500**
  - halfvec: 2·d + 8 ≤ 2008 → **d ≤ 1000**
  - sparsevec: 8·nnz + 16 ≤ 2008 → **nnz ≤ 249**
- **Note:** adding a PK drops these by 1 → 499 / 999 / 248; adding more columns can drop them even further



PERCONA

Vector storage: worked example

- Two tables, 100k rows each. *table_400* has vector(400), *table_500* has vector(500). The PK pushes *table_500* over the 2008 byte threshold, so it TOASTs

```
table_name | table_pages | toast_pages | total_size
-----------+-------------+-------------+-----------
 table_400 |       25000 |           0 |     198 MB
 table_500 |         637 |       33334 |     272 MB
```

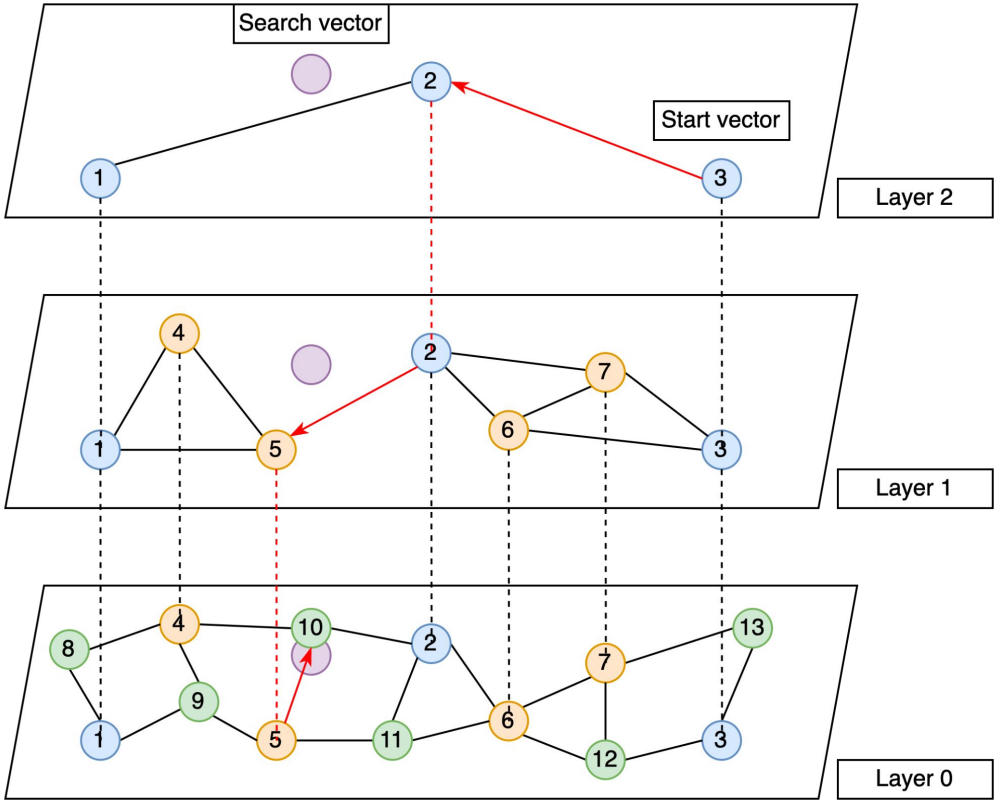
- Exact-NN query on:
  - *table_400* → Parallel Seq Scan, ~39 ms
  - *table_500* → Seq Scan, ~686 ms (planner under-costs out-of-line)
- Solutions:
  - `SET min_parallel_table_scan_size = 1` and `parallel_setup_cost = 1`
  - `ALTER TABLE ... SET STORAGE PLAIN` (with caveats)

PERCONA

Indexing overview: HNSW vs IVFFlat

- **HNSW** — Hierarchical Navigable Small Worlds
  - Multilayer graph grouping vectors into neighbourhoods
  - Better query performance / recall ratio
  - Slower builds, more memory
  - Can be created on an empty table (no training step)
  - Easier to manage — doesn't need to be rebuilt as data grows
- **IVFFlat** — Inverted File Flat
  - Uses k-means to group vectors into lists
  - Opposite trade-offs: faster builds, less memory, lower recall
  - Centers to be recalculated after substantial data churn (drop & recreate)
  - Faster and more consistent inserts as the dataset grows

Indexing: HNSW



PERCONA

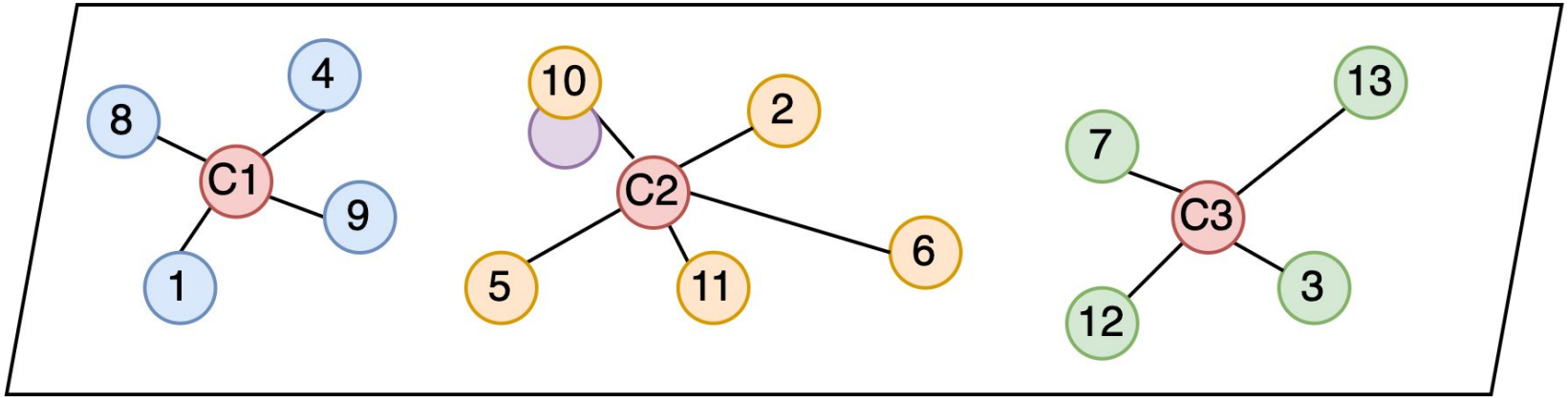
Indexing: HNSW

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops) WITH (m = 16, ef_construction = 64);
```

- Supported types: **vector** (≤2,000 dim) · **halfvec** (≤4,000) · **bit** (≤64,000) · **sparsevec** (≤1,000 nz)
- Distances: l2 · ip · cosine · l1 · hamming · jaccard
- Build options:
  - *m = 16* — neighbours per vector
  - *ef_construction = 64* — size of the dynamic candidate list at build time
- **Query knob:** `hnsw.ef_search` = 40 — size of the candidate list at query time
- Note: start by increasing *ef_construction* before *m* — it has a more predictable effect on build time. Keep *ef_construction* ≥ 2 · m.

PERCONA

Indexing: IVFFlat



PERCONA

Indexing: IVFFlat

```
CREATE INDEX ON items USING ivfflat (embedding vector_cosine_ops) WITH (lists = 200);
```

- Supported types: **vector** (≤2,000) · **halfvec** (≤4,000) · **bit** (≤64,000)
- Distances: l2 · ip · cosine · hamming
- Build option: *lists = 100*
  - ≤ 1M rows → rows / 1000
  - > 1M rows → sqrt(rows)
- **Query knob:** `ivfflat.probes = 1` (start with sqrt(lists))
- Note: build the IVFFlat index after loading data. Empty-table builds give us useless cluster centres.

PERCONA

Index build tuning

- `SET maintenance_work_mem = '8GB';`
  - Must fit the HNSW graph in memory or we get:

```
NOTICE: hnsw graph no longer fits into maintenance_work_mem after N tuples
```

- `SET max_parallel_maintenance_workers = 7;` (plus bump max_parallel_workers)
- `CREATE/REINDEX INDEX CONCURRENTLY`
- Track build progress:

```
SELECT phase, round(100.0 * blocks_done / nullif(blocks_total, 0), 1) AS pct
FROM pg_stat_progress_create_index;
```

Filtering, iterative scans & partial indexes

- Filtering is applied **after** the HNSW or IVFFlat scan
  - `WHERE category_id = 123 ORDER BY embedding <-> ... LIMIT 5` can return fewer than 5 rows
- Fix 1: bump up ef_search, probes and LIMITs
- Fix 2: iterative scans — works for both index types
  - SET hnsw.iterative_scan = strict_order; (or relaxed_order)
  - SET ivfflat.iterative_scan = relaxed_order; (plus ivfflat.max_probes)
- Fix 3: partial indexing

```
CREATE INDEX ON movies USING hnsw (overview_embedding vector_l2_ops)
WITH (m = 8, ef_construction = 32) WHERE (category_id = 123);
```


PERCONA

Query & ops tuning

- Querying & ops:
  - Bulk load: `COPY items (embedding) FROM STDIN WITH (FORMAT BINARY);`
  - Debug: `EXPLAIN (ANALYZE, BUFFERS, ...)` on every new query
  - Monitor recall: `BEGIN; SET LOCAL enable_indexscan = off;` then compare to the indexed result
  - HNSW vacuum: `REINDEX INDEX [CONCURRENTLY] idx;` then `VACUUM` table;
- Postgres GUCs to keep in mind:
  - `shared_buffers` (25–40 % RAM) and `max_parallel_workers_per_gather`
  - `min_parallel_table_scan_size = 1` and `parallel_setup_cost = 1` (the TOAST fix)
  - `random_page_cost`, `work_mem` and `maintenance_work_mem`

PERCONA

Upgrades

```
postgres=# ALTER EXTENSION vector UPDATE;
postgres=# SELECT extversion FROM pg_extension WHERE extname = 'vector';
```

- Install the new binaries the same way we installed the old ones
- Then `ALTER EXTENSION` in each database
- Always read release notes and changelogs!

Percona for **PostgreSQL**

DEMO

PERCONA

Percona for **PostgreSQL**

DEMO

PERCONA

Percona for **PostgreSQL**

DEMO

PERCONA

PGVector Movies Lab

Find movies by semantic similarity of descriptions.

farm animals adventure funny | 10 | Search

Farm Frolics 0.7486
1941 · ID 187102
A series of wacky vignettes involving farm animals.

Les tout-petits de la ferme 0.6952
1943 · ID 577772
The animal life of on a farm.

Concentrate 0.6683
1929 · ID 1148883
Funny animals learning how to do tricks by concentrating. Then suddenly, the farmer wants to try it.

Fresh Air 0.6539
1921 · ID 1327750
A hunting comedy full of cartoonish gags.

Speaking of Animals No. Y6-1: Stork Crazy 0.6495

Percona for **PostgreSQL**

DEMO

PERCONA

PERCONA®


PERCONA®