pgvector

Open-source vector similarity search for Postgres

Store your vectors with the rest of your data. Supports:

- exact and approximate nearest neighbor search
- single-precision, half-precision, binary, and sparse vectors
- L2 distance, inner product, cosine distance, L1 distance, Hamming distance, and Jaccard distance
- any language with a Postgres client

Plus ACID compliance, point-in-time recovery, JOINs, and all of the other great features of Postgres

Installation

Linux and Mac

Compile and install the extension (supports Postgres 13+)

```
cd /tmp
git clone --branch v0.8.0 https://github.com/pgvector/pgvector.git
cd pgvector
make
make install # may need sudo
```

See the installation notes if you run into issues

You can also install it with Docker, Homebrew, PGXN, APT, Yum, pkg, or conda-forge, and it comes preinstalled with Postgres.app and many hosted providers. There are also instructions for GitHub Actions.

Windows

Ensure C++ support in Visual Studio is installed, and run:

```
call "C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Auxiliary\Build\vcvars64.ba
```

Note: The exact path will vary depending on your Visual Studio version and edition

Then use `nmake` to build:

```
set "PGROOT=C:\Program Files\PostgreSQL\16"
cd %TEMP%
git clone --branch v0.8.0 https://github.com/pgvector/pgvector.git
cd pgvector
nmake /F Makefile.win
nmake /F Makefile.win install
```

Note: Postgres 17 is not supported yet due to an upstream issue

See the installation notes if you run into issues

You can also install it with Docker or conda-forge.

Getting Started

Enable the extension (do this once in each database where you want to use it)

```
CREATE EXTENSION vector;
```

Create a vector column with 3 dimensions

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding vector(3));
```

Insert vectors

```
INSERT INTO items (embedding) VALUES ('[1,2,3]'), ('[4,5,6]');
```

Get the nearest neighbors by L2 distance

```
SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

Also supports inner product (`<#>`), cosine distance (`<=>`), and L1 distance (`<+>`, added in 0.7.0)

Note: `<#>` returns the negative inner product since Postgres only supports `ASC` order index scans on operators

Storing

Create a new table with a vector column

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding vector(3));
```

Or add a vector column to an existing table

```
ALTER TABLE items ADD COLUMN embedding vector(3);
```

Also supports half-precision, binary, and sparse vectors

Insert vectors

```
INSERT INTO items (embedding) VALUES ('[1,2,3]'), ('[4,5,6]');
```

Or load vectors in bulk using `COPY` (example)

```
COPY items (embedding) FROM STDIN WITH (FORMAT BINARY);
```

Upsert vectors

```
INSERT INTO items (id, embedding) VALUES (1, '[1,2,3]'), (2, '[4,5,6]')
    ON CONFLICT (id) DO UPDATE SET embedding = EXCLUDED.embedding;
```

Update vectors

```
UPDATE items SET embedding = '[1,2,3]' WHERE id = 1;
```

Delete vectors

```
DELETE FROM items WHERE id = 1;
```

Querying

Get the nearest neighbors to a vector

```
SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

Supported distance functions are:

- `<->` - L2 distance
- `<#>` - (negative) inner product
- `<=>` - cosine distance
- `<+>` - L1 distance (added in 0.7.0)
- `<~>` - Hamming distance (binary vectors, added in 0.7.0)
- `<%>` - Jaccard distance (binary vectors, added in 0.7.0)

Get the nearest neighbors to a row

```
SELECT * FROM items WHERE id != 1 ORDER BY embedding <-> (SELECT embedding FROM items WHERE
```

Get rows within a certain distance

```
SELECT * FROM items WHERE embedding <-> '[3,1,2]' < 5;
```

Note: Combine with `ORDER BY` and `LIMIT` to use an index

Distances Get the distance

```
SELECT embedding <-> '[3,1,2]' AS distance FROM items;
```

For inner product, multiply by -1 (since `<#>` returns the negative inner product)

```
SELECT (embedding <#> '[3,1,2]') * -1 AS inner_product FROM items;
```

For cosine similarity, use 1 - cosine distance

```
SELECT 1 - (embedding <=> '[3,1,2]') AS cosine_similarity FROM items;
```

Aggregates Average vectors

```
SELECT AVG(embedding) FROM items;
```

Average groups of vectors

```
SELECT category_id, AVG(embedding) FROM items GROUP BY category_id;
```

Indexing

By default, pgvector performs exact nearest neighbor search, which provides perfect recall.

You can add an index to use approximate nearest neighbor search, which trades some recall for speed. Unlike typical indexes, you will see different results for queries after adding an approximate index.

Supported index types are:

- HNSW
- IVFFlat

HNSW

An HNSW index creates a multilayer graph. It has better query performance than IVFFlat (in terms of speed-recall tradeoff), but has slower build times and uses more memory. Also, an index can be created without any data in the table since there isn't a training step like IVFFlat.

Add an index for each distance function you want to use.

L2 distance

```sql
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops);
```

Note: Use `halfvec_l2_ops` for `halfvec` and `sparsevec_l2_ops` for `sparsevec` (and similar with the other distance functions)

Inner product

```sql
CREATE INDEX ON items USING hnsw (embedding vector_ip_ops);
```

Cosine distance

```sql
CREATE INDEX ON items USING hnsw (embedding vector_cosine_ops);
```

L1 distance - added in 0.7.0

```sql
CREATE INDEX ON items USING hnsw (embedding vector_l1_ops);
```

Hamming distance - added in 0.7.0

```sql
CREATE INDEX ON items USING hnsw (embedding bit_hamming_ops);
```

Jaccard distance - added in 0.7.0

```sql
CREATE INDEX ON items USING hnsw (embedding bit_jaccard_ops);
```

Supported types are:

- `vector` - up to 2,000 dimensions
- `halfvec` - up to 4,000 dimensions (added in 0.7.0)
- `bit` - up to 64,000 dimensions (added in 0.7.0)
- `sparsevec` - up to 1,000 non-zero elements (added in 0.7.0)

Index Options

Specify HNSW parameters

- `m` - the max number of connections per layer (16 by default)
- `ef_construction` - the size of the dynamic candidate list for constructing the graph (64 by default)

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops) WITH (m = 16, ef_construction = 6
```

A higher value of `ef_construction` provides better recall at the cost of index build time / insert speed.

Query Options

Specify the size of the dynamic candidate list for search (40 by default)

```
SET hnsw.ef_search = 100;
```

A higher value provides better recall at the cost of speed.

Use `SET LOCAL` inside a transaction to set it for a single query

```
BEGIN;
SET LOCAL hnsw.ef_search = 100;
SELECT ...
COMMIT;
```

Index Build Time

Indexes build significantly faster when the graph fits into `maintenance_work_mem`

```
SET maintenance_work_mem = '8GB';
```

A notice is shown when the graph no longer fits

```
NOTICE:  hnsw graph no longer fits into maintenance_work_mem after 100000 tuples
DETAIL:  Building will take significantly more time.
HINT:  Increase maintenance_work_mem to speed up builds.
```

Note: Do not set `maintenance_work_mem` so high that it exhausts the memory on the server

Like other index types, it's faster to create an index after loading your initial data

Starting with 0.6.0, you can also speed up index creation by increasing the number of parallel workers (2 by default)

```
SET max_parallel_maintenance_workers = 7; -- plus leader
```

For a large number of workers, you may also need to increase `max_parallel_workers` (8 by default)

Indexing Progress

Check indexing progress

```
SELECT phase, round(100.0 * blocks_done / nullif(blocks_total, 0), 1) AS "%" FROM pg_stat_pr
```

The phases for HNSW are:

1. `initializing`
2. `loading tuples`

IVFFlat

An IVFFlat index divides vectors into lists, and then searches a subset of those lists that are closest to the query vector. It has faster build times and uses less memory than HNSW, but has lower query performance (in terms of speed-recall tradeoff).

Three keys to achieving good recall are:

1. Create the index *after* the table has some data
2. Choose an appropriate number of lists - a good place to start is `rows / 1000` for up to 1M rows and `sqrt(rows)` for over 1M rows
3. When querying, specify an appropriate number of probes (higher is better for recall, lower is better for speed) - a good place to start is `sqrt(lists)`

Add an index for each distance function you want to use.

L2 distance

```
CREATE INDEX ON items USING ivfflat (embedding vector_l2_ops) WITH (lists = 100);
```

Note: Use `halfvec_l2_ops` for `halfvec` (and similar with the other distance functions)

Inner product

```
CREATE INDEX ON items USING ivfflat (embedding vector_ip_ops) WITH (lists = 100);
```

Cosine distance

```
CREATE INDEX ON items USING ivfflat (embedding vector_cosine_ops) WITH (lists = 100);
```

Hamming distance - added in 0.7.0

```
CREATE INDEX ON items USING ivfflat (embedding bit_hamming_ops) WITH (lists = 100);
```

Supported types are:

- `vector` - up to 2,000 dimensions
- `halfvec` - up to 4,000 dimensions (added in 0.7.0)
- `bit` - up to 64,000 dimensions (added in 0.7.0)

Query Options

Specify the number of probes (1 by default)

```
SET ivfflat.probes = 10;
```

A higher value provides better recall at the cost of speed, and it can be set to the number of lists for exact nearest neighbor search (at which point the planner won't use the index)

Use `SET LOCAL` inside a transaction to set it for a single query

```
BEGIN;
SET LOCAL ivfflat.probes = 10;
SELECT ...
COMMIT;
```

Index Build Time

Speed up index creation on large tables by increasing the number of parallel workers (2 by default)

```
SET max_parallel_maintenance_workers = 7; -- plus leader
```

For a large number of workers, you may also need to increase `max_parallel_workers` (8 by default)

Indexing Progress

Check indexing progress

```
SELECT phase, round(100.0 * tuples_done / nullif(tuples_total, 0), 1) AS "%" FROM pg_stat_pr
```

The phases for IVFFlat are:

1. `initializing`
2. `performing k-means`
3. `assigning tuples`
4. `loading tuples`

Note: `%` is only populated during the `loading tuples` phase

Filtering

There are a few ways to index nearest neighbor queries with a `WHERE` clause.

```
SELECT * FROM items WHERE category_id = 123 ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

A good place to start is creating an index on the filter column. This can provide fast, exact nearest neighbor search in many cases. Postgres has a number of index types for this: B-tree (default), hash, GiST, SP-GiST, GIN, and BRIN.

```
CREATE INDEX ON items (category_id);
```

For multiple columns, consider a multicolumn index.

```
CREATE INDEX ON items (location_id, category_id);
```

Exact indexes work well for conditions that match a low percentage of rows. Otherwise, approximate indexes can work better.

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops);
```

With approximate indexes, filtering is applied *after* the index is scanned. If a condition matches 10% of rows, with HNSW and the default `hnsw.ef_search` of 40, only 4 rows will match on average. For more rows, increase `hnsw.ef_search`.

```
SET hnsw.ef_search = 200;
```

Starting with 0.8.0, you can enable iterative index scans, which will automatically scan more of the index when needed.

```
SET hnsw.iterative_scan = strict_order;
```

If filtering by only a few distinct values, consider partial indexing.

```
CREATE INDEX ON items USING hnsw (embedding vector_l2_ops) WHERE (category_id = 123);
```

If filtering by many different values, consider partitioning.

```
CREATE TABLE items (embedding vector(3), category_id int) PARTITION BY LIST(category_id);
```

Iterative Index Scans

Added in 0.8.0

With approximate indexes, queries with filtering can return less results since filtering is applied *after* the index is scanned. Starting with 0.8.0, you can enable iterative index scans, which will automatically scan more of the index until enough results are found (or it reaches `hnsw.max_scan_tuples` or `ivfflat.max_probes`).

Iterative scans can use strict or relaxed ordering.

Strict ensures results are in the exact order by distance

```
SET hnsw.iterative_scan = strict_order;
```

Relaxed allows results to be slightly out of order by distance, but provides better recall

```
SET hnsw.iterative_scan = relaxed_order;
# or
SET ivfflat.iterative_scan = relaxed_order;
```

With relaxed ordering, you can use a materialized CTE to get strict ordering

```
WITH relaxed_results AS MATERIALIZED (
    SELECT id, embedding <-> '[1,2,3]' AS distance FROM items WHERE category_id = 123 ORDER
) SELECT * FROM relaxed_results ORDER BY distance;
```

For queries that filter by distance, use a materialized CTE and place the distance filter outside of it for best performance (due to the current behavior of the Postgres executor)

```
WITH nearest_results AS MATERIALIZED (
    SELECT id, embedding <-> '[1,2,3]' AS distance FROM items ORDER BY distance LIMIT 5
) SELECT * FROM nearest_results WHERE distance < 5 ORDER BY distance;
```

Note: Place any other filters inside the CTE

Iterative Scan Options

Since scanning a large portion of an approximate index is expensive, there are options to control when a scan ends.

HNSW Specify the max number of tuples to visit (20,000 by default)

```
SET hnsw.max_scan_tuples = 20000;
```

Note: This is approximate and does not affect the initial scan

Specify the max amount of memory to use, as a multiple of `work_mem` (1 by default)

```
SET hnsw.scan_mem_multiplier = 2;
```

Note: Try increasing this if increasing `hnsw.max_scan_tuples` does not improve recall

IVFFlat Specify the max number of probes

```
SET ivfflat.max_probes = 100;
```

Note: If this is lower than `ivfflat.probes`, `ivfflat.probes` will be used

Half-Precision Vectors

Added in 0.7.0

Use the `halfvec` type to store half-precision vectors

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding halfvec(3));
```

Half-Precision Indexing

Added in 0.7.0

Index vectors at half precision for smaller indexes

```sql
CREATE INDEX ON items USING hnsw ((embedding::halfvec(3)) halfvec_l2_ops);
```

Get the nearest neighbors

```sql
SELECT * FROM items ORDER BY embedding::halfvec(3) <-> '[1,2,3]' LIMIT 5;
```

Binary Vectors

Use the `bit` type to store binary vectors (example)

```sql
CREATE TABLE items (id bigserial PRIMARY KEY, embedding bit(3));
INSERT INTO items (embedding) VALUES ('000'), ('111');
```

Get the nearest neighbors by Hamming distance (added in 0.7.0)

```sql
SELECT * FROM items ORDER BY embedding <~> '101' LIMIT 5;
```

Or (before 0.7.0)

```sql
SELECT * FROM items ORDER BY bit_count(embedding # '101') LIMIT 5;
```

Also supports Jaccard distance (`<%>`)

Binary Quantization

Added in 0.7.0

Use expression indexing for binary quantization

```sql
CREATE INDEX ON items USING hnsw ((binary_quantize(embedding)::bit(3)) bit_hamming_ops);
```

Get the nearest neighbors by Hamming distance

```sql
SELECT * FROM items ORDER BY binary_quantize(embedding)::bit(3) <~> binary_quantize('[1,-2,3
```

Re-rank by the original vectors for better recall

```sql
SELECT * FROM (
    SELECT * FROM items ORDER BY binary_quantize(embedding)::bit(3) <~> binary_quantize('[1,
) ORDER BY embedding <=> '[1,-2,3]' LIMIT 5;
```

Sparse Vectors

Added in 0.7.0

Use the `sparsevec` type to store sparse vectors

```sql
CREATE TABLE items (id bigserial PRIMARY KEY, embedding sparsevec(5));
```

Insert vectors

```sql
INSERT INTO items (embedding) VALUES ('{1:1,3:2,5:3}/5'), ('{1:4,3:5,5:6}/5');
```

The format is `{index1:value1,index2:value2}/dimensions` and indices start at 1 like SQL arrays

Get the nearest neighbors by L2 distance

```
SELECT * FROM items ORDER BY embedding <-> '{1:3,3:1,5:2}/5' LIMIT 5;
```

Hybrid Search

Use together with Postgres full-text search for hybrid search.

```
SELECT id, content FROM items, plainto_tsquery('hello search') query
    WHERE textsearch @@ query ORDER BY ts_rank_cd(textsearch, query) DESC LIMIT 5;
```

You can use Reciprocal Rank Fusion or a cross-encoder to combine results.

Indexing Subvectors

Added in 0.7.0

Use expression indexing to index subvectors

```
CREATE INDEX ON items USING hnsw ((subvector(embedding, 1, 3)::vector(3)) vector_cosine_ops
```

Get the nearest neighbors by cosine distance

```
SELECT * FROM items ORDER BY subvector(embedding, 1, 3)::vector(3) <=> subvector('[1,2,3,4,5
```

Re-rank by the full vectors for better recall

```
SELECT * FROM (
    SELECT * FROM items ORDER BY subvector(embedding, 1, 3)::vector(3) <=> subvector('[1,2,3
) ORDER BY embedding <=> '[1,2,3,4,5]' LIMIT 5;
```

Performance

Tuning

Use a tool like PgTune to set initial values for Postgres server parameters. For instance, `shared_buffers` should typically be 25% of the server's memory. You can find the config file with:

```
SHOW config_file;
```

And check individual settings with:

```
SHOW shared_buffers;
```

Be sure to restart Postgres for changes to take effect.

Loading

Use COPY for bulk loading data (example).

```
COPY items (embedding) FROM STDIN WITH (FORMAT BINARY);
```

Add any indexes *after* loading the initial data for best performance.

Indexing

See index build time for HNSW and IVFFlat.

In production environments, create indexes concurrently to avoid blocking writes.

```
CREATE INDEX CONCURRENTLY ...
```

Querying

Use EXPLAIN ANALYZE to debug performance.

```
EXPLAIN ANALYZE SELECT * FROM items ORDER BY embedding <-> '[3,1,2]' LIMIT 5;
```

Exact Search To speed up queries without an index, increase max_parallel_workers_per_gather.

```
SET max_parallel_workers_per_gather = 4;
```

If vectors are normalized to length 1 (like OpenAI embeddings), use inner product for best performance.

```
SELECT * FROM items ORDER BY embedding <#> '[3,1,2]' LIMIT 5;
```

Approximate Search To speed up queries with an IVFFlat index, increase the number of inverted lists (at the expense of recall).

```
CREATE INDEX ON items USING ivfflat (embedding vector_l2_ops) WITH (lists = 1000);
```

Vacuuming

Vacuuming can take a while for HNSW indexes. Speed it up by reindexing first.

```
REINDEX INDEX CONCURRENTLY index_name;
VACUUM table_name;
```

Monitoring

Monitor performance with pg_stat_statements (be sure to add it to shared_preload_libraries).

```
CREATE EXTENSION pg_stat_statements;
```

Get the most time-consuming queries with:

```
SELECT query, calls, ROUND((total_plan_time + total_exec_time) / calls) AS avg_time_ms,
    ROUND((total_plan_time + total_exec_time) / 60000) AS total_time_min
    FROM pg_stat_statements ORDER BY total_plan_time + total_exec_time DESC LIMIT 20;
```

Note: Replace `total_plan_time + total_exec_time` with `total_time` for Postgres < 13

Monitor recall by comparing results from approximate search with exact search.

```
BEGIN;
SET LOCAL enable_indexscan = off; -- use exact search
SELECT ...
COMMIT;
```

Scaling

Scale pgvector the same way you scale Postgres.

Scale vertically by increasing memory, CPU, and storage on a single instance. Use existing tools to tune parameters and monitor performance.

Scale horizontally with replicas, or use Citus or another approach for sharding (example).

Languages

Use pgvector from any language with a Postgres client. You can even generate and store vectors in one language and query them in another.

Language	Libraries / Examples
C	pgvector-c
C++	pgvector-cpp
C#, F#, Visual Basic	pgvector-dotnet
Crystal	pgvector-crystal
Dart	pgvector-dart
Elixir	pgvector-elixir
Go	pgvector-go
Haskell	pgvector-haskell
Java, Kotlin, Groovy, Scala	pgvector-java
JavaScript, TypeScript	pgvector-node
Julia	pgvector-julia
Lisp	pgvector-lisp
Lua	pgvector-lua
Nim	pgvector-nim
OCaml	pgvector-ocaml
Perl	pgvector-perl
PHP	pgvector-php
Python	pgvector-python

Language	Libraries / Examples
R	pgvector-r
Ruby	pgvector-ruby, Neighbor
Rust	pgvector-rust
Swift	pgvector-swift
Zig	pgvector-zig

Frequently Asked Questions

How many vectors can be stored in a single table? A non-partitioned table has a limit of 32 TB by default in Postgres. A partitioned table can have thousands of partitions of that size.

Is replication supported? Yes, pgvector uses the write-ahead log (WAL), which allows for replication and point-in-time recovery.

What if I want to index vectors with more than 2,000 dimensions? You can use half-precision indexing to index up to 4,000 dimensions or binary quantization to index up to 64,000 dimensions. Another option is dimensionality reduction.

Can I store vectors with different dimensions in the same column? You can use `vector` as the type (instead of `vector(3)`).

```
CREATE TABLE embeddings (model_id bigint, item_id bigint, embedding vector, PRIMARY KEY (mod
```

However, you can only create indexes on rows with the same number of dimensions (using expression and partial indexing):

```
CREATE INDEX ON embeddings USING hnsw ((embedding::vector(3)) vector_l2_ops) WHERE (model_ic
```

and query with:

```
SELECT * FROM embeddings WHERE model_id = 123 ORDER BY embedding::vector(3) <-> '[3,1,2]' LI
```

Can I store vectors with more precision? You can use the `double precision[]` or `numeric[]` type to store vectors with more precision.

```
CREATE TABLE items (id bigserial PRIMARY KEY, embedding double precision[]);

-- use {} instead of [] for Postgres arrays
INSERT INTO items (embedding) VALUES ('{1,2,3}'), ('{4,5,6}');
```

Optionally, add a check constraint to ensure data can be converted to the `vector` type and has the expected dimensions.

```
ALTER TABLE items ADD CHECK (vector_dims(embedding::vector) = 3);
```

Use expression indexing to index (at a lower precision):

```
CREATE INDEX ON items USING hnsw ((embedding::vector(3)) vector_l2_ops);
```

and query with:

```
SELECT * FROM items ORDER BY embedding::vector(3) <-> '[3,1,2]' LIMIT 5;
```

Do indexes need to fit into memory? No, but like other index types, you'll likely see better performance if they do. You can get the size of an index with:

```
SELECT pg_size_pretty(pg_relation_size('index_name'));
```

Troubleshooting

Why isn't a query using an index? The query needs to have an `ORDER BY` and `LIMIT`, and the `ORDER BY` must be the result of a distance operator (not an expression) in ascending order.

```
-- index
ORDER BY embedding <=> '[3,1,2]' LIMIT 5;

-- no index
ORDER BY 1 - (embedding <=> '[3,1,2]') DESC LIMIT 5;
```

You can encourage the planner to use an index for a query with:

```
BEGIN;
SET LOCAL enable_seqscan = off;
SELECT ...
COMMIT;
```

Also, if the table is small, a table scan may be faster.

Why isn't a query using a parallel table scan? The planner doesn't consider out-of-line storage in cost estimates, which can make a serial scan look cheaper. You can reduce the cost of a parallel scan for a query with:

```
BEGIN;
SET LOCAL min_parallel_table_scan_size = 1;
SET LOCAL parallel_setup_cost = 1;
SELECT ...
COMMIT;
```

or choose to store vectors inline:

```
ALTER TABLE items ALTER COLUMN embedding SET STORAGE PLAIN;
```

Why are there less results for a query after adding an HNSW index? Results are limited by the size of the dynamic candidate list (`hnsw.ef_search`). There may be even less results due to dead tuples or filtering conditions in the query. We recommend setting `hnsw.ef_search` to at least twice the `LIMIT` of the query. If you need more than 500 results, use an IVFFlat index instead.

Also, note that `NULL` vectors are not indexed (as well as zero vectors for cosine distance).

Why are there less results for a query after adding an IVFFlat index? The index was likely created with too little data for the number of lists. Drop the index until the table has more data.

```
DROP INDEX index_name;
```

Results can also be limited by the number of probes (`ivfflat.probes`).

Also, note that `NULL` vectors are not indexed (as well as zero vectors for cosine distance).

Reference

- Vector
- Halfvec
- Bit
- Sparsevec

Vector Type

Each vector takes `4 * dimensions + 8` bytes of storage. Each element is a single-precision floating-point number (like the `real` type in Postgres), and all elements must be finite (no `NaN`, `Infinity` or `-Infinity`). Vectors can have up to 16,000 dimensions.

Vector Operators

Operator	Description	Added
+	element-wise addition	
-	element-wise subtraction	
*	element-wise multiplication	0.5.0
||	concatenate	0.7.0
<->	Euclidean distance	
<#>	negative inner product	
<=>	cosine distance	
<+>	taxicab distance	0.7.0

Vector Functions

Function	Description	Added
binary_quantize(vector) → bit	binary quantize	0.7.0
cosine_distance(vector, vector) → double precision	cosine distance	
inner_product(vector, vector) → double precision	inner product	
l1_distance(vector, vector) → double precision	taxicab distance	0.5.0
l2_distance(vector, vector) → double precision	Euclidean distance	
l2_normalize(vector) → vector	Normalize with Euclidean norm	0.7.0
subvector(vector, integer, integer) → vector	subvector	0.7.0
vector_dims(vector) → integer	number of dimensions	
vector_norm(vector) → double precision	Euclidean norm	

Vector Aggregate Functions

Function	Description	Added
avg(vector) → vector	average	
sum(vector) → vector	sum	0.5.0

Halfvec Type

Each half vector takes `2 * dimensions + 8` bytes of storage. Each element is a half-precision floating-point number, and all elements must be finite (no `NaN`, `Infinity` or `-Infinity`). Half vectors can have up to 16,000 dimensions.

Halfvec Operators

Operator	Description	Added
+	element-wise addition	0.7.0
-	element-wise subtraction	0.7.0
*	element-wise multiplication	0.7.0
||	concatenate	0.7.0
<->	Euclidean distance	0.7.0
<#>	negative inner product	0.7.0
<=>	cosine distance	0.7.0
<+>	taxicab distance	0.7.0

Halfvec Functions

Function	Description	Added
binary_quantize(halfvec) → bit	binary quantize	0.7.0
cosine_distance(halfvec, halfvec) → double precision	cosine distance	0.7.0
inner_product(halfvec, halfvec) → double precision	inner product	0.7.0
l1_distance(halfvec, halfvec) → double precision	taxicab distance	0.7.0
l2_distance(halfvec, halfvec) → double precision	Euclidean distance	0.7.0
l2_norm(halfvec) → double precision	Euclidean norm	0.7.0
l2_normalize(halfvec) → halfvec	Normalize with Euclidean norm	0.7.0
subvector(halfvec, integer, integer) → halfvec	subvector	0.7.0
vector_dims(halfvec) → integer	number of dimensions	0.7.0

Halfvec Aggregate Functions

Function	Description	Added
avg(halfvec) → halfvec	average	0.7.0
sum(halfvec) → halfvec	sum	0.7.0

Bit Type

Each bit vector takes `dimensions / 8 + 8` bytes of storage. See the Postgres docs for more info.

Bit Operators

Operator	Description	Added
<~>	Hamming distance	0.7.0
<%>	Jaccard distance	0.7.0

Bit Functions

Function	Description	Added
hamming_distance(bit, bit) → double precision	Hamming distance	0.7.0
jaccard_distance(bit, bit) → double precision	Jaccard distance	0.7.0

Sparsevec Type

Each sparse vector takes `8 * non-zero elements + 16` bytes of storage. Each element is a single-precision floating-point number, and all elements must be finite (no `NaN`, `Infinity` or `-Infinity`). Sparse vectors can have up to 16,000 non-zero elements.

Sparsevec Operators

Operator	Description	Added
<->	Euclidean distance	0.7.0
<#>	negative inner product	0.7.0
<=>	cosine distance	0.7.0
<+>	taxicab distance	0.7.0

Sparsevec Functions

Function	Description	Added
cosine_distance(sparsevec, sparsevec) → double precision	cosine distance	0.7.0
inner_product(sparsevec, sparsevec) → double precision	inner product	0.7.0
l1_distance(sparsevec, sparsevec) → double precision	taxicab distance	0.7.0
l2_distance(sparsevec, sparsevec) → double precision	Euclidean distance	0.7.0
l2_norm(sparsevec) → double precision	Euclidean norm	0.7.0
l2_normalize(sparsevec) → sparsevec	Normalize with Euclidean norm	0.7.0

Installation Notes - Linux and Mac

Postgres Location

If your machine has multiple Postgres installations, specify the path to pg_config with:

```
export PG_CONFIG=/Library/PostgreSQL/17/bin/pg_config
```

Then re-run the installation instructions (run `make clean` before `make` if needed). If `sudo` is needed for `make install`, use:

```
sudo --preserve-env=PG_CONFIG make install
```

A few common paths on Mac are:

- EDB installer - `/Library/PostgreSQL/17/bin/pg_config`
- Homebrew (arm64) - `/opt/homebrew/opt/postgresql@17/bin/pg_config`

- Homebrew (x86-64) - `/usr/local/opt/postgresql@17/bin/pg_config`

Note: Replace `17` with your Postgres server version

Missing Header

If compilation fails with `fatal error: postgres.h: No such file or directory`, make sure Postgres development files are installed on the server.

For Ubuntu and Debian, use:

```
sudo apt install postgresql-server-dev-17
```

Note: Replace `17` with your Postgres server version

Missing SDK

If compilation fails and the output includes `warning: no such sysroot directory` on Mac, reinstall Xcode Command Line Tools.

Portability

By default, pgvector compiles with `-march=native` on some platforms for best performance. However, this can lead to `Illegal instruction` errors if trying to run the compiled extension on a different machine.

To compile for portability, use:

```
make OPTFLAGS=""
```

Installation Notes - Windows

Missing Header

If compilation fails with `Cannot open include file: 'postgres.h': No such file or directory`, make sure `PGROOT` is correct.

Permissions

If installation fails with `Access is denied`, re-run the installation instructions as an administrator.

Additional Installation Methods

Docker

Get the Docker image with:

```
docker pull pgvector/pgvector:pg17
```

This adds pgvector to the Postgres image (replace `17` with your Postgres server version, and run it the same way).

You can also build the image manually:

```
git clone --branch v0.8.0 https://github.com/pgvector/pgvector.git
cd pgvector
docker build --pull --build-arg PG_MAJOR=17 -t myuser/pgvector .
```

Homebrew

With Homebrew Postgres, you can use:

```
brew install pgvector
```

Note: This only adds it to the `postgresql@17` and `postgresql@14` formulas

PGXN

Install from the PostgreSQL Extension Network with:

```
pgxn install vector
```

APT

Debian and Ubuntu packages are available from the PostgreSQL APT Repository. Follow the setup instructions and run:

```
sudo apt install postgresql-17-pgvector
```

Note: Replace `17` with your Postgres server version

Yum

RPM packages are available from the PostgreSQL Yum Repository. Follow the setup instructions for your distribution and run:

```
sudo yum install pgvector_17
# or
sudo dnf install pgvector_17
```

Note: Replace `17` with your Postgres server version

pkg

Install the FreeBSD package with:

```
pkg install postgresql15-pgvector
```

or the port with:

```
cd /usr/ports/databases/pgvector
make install
```

conda-forge

With Conda Postgres, install from conda-forge with:

```
conda install -c conda-forge pgvector
```

This method is community-maintained by [@mmcauliffe](https://github.com/mmcauliffe)

Postgres.app

Download the latest release with Postgres 15+.

Hosted Postgres

pgvector is available on these providers.

Upgrading

Install the latest version (use the same method as the original installation). Then in each database you want to upgrade, run:

```
ALTER EXTENSION vector UPDATE;
```

You can check the version in the current database with:

```
SELECT extversion FROM pg_extension WHERE extname = 'vector';
```

Upgrade Notes

0.6.0

Postgres 12 If upgrading with Postgres 12, remove this line from `sql/vector--0.5.1--0.6.0.sql`:

```
ALTER TYPE vector SET (STORAGE = external);
```

Then run `make install` and `ALTER EXTENSION vector UPDATE;`.

Docker The Docker image is now published in the `pgvector` org, and there are tags for each supported version of Postgres (rather than a `latest` tag).

```
docker pull pgvector/pgvector:pg16
# or
docker pull pgvector/pgvector:0.6.0-pg16
```

Also, if you've increased `maintenance_work_mem`, make sure `--shm-size` is at least that size to avoid an error with parallel HNSW index builds.

```
docker run --shm-size=1g ...
```

Thanks

Thanks to:

- PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension
- Faiss: A Library for Efficient Similarity Search and Clustering of Dense Vectors
- Using the Triangle Inequality to Accelerate k-means
- k-means++: The Advantage of Careful Seeding
- Concept Decompositions for Large Sparse Text Data using Clustering
- Efficient and Robust Approximate Nearest Neighbor Search using Hierarchical Navigable Small World Graphs

History

View the changelog

Contributing

Everyone is encouraged to help improve this project. Here are a few ways you can help:

- Report bugs
- Fix bugs and submit pull requests
- Write, clarify, or fix documentation
- Suggest or add new features

To get started with development:

```
git clone https://github.com/pgvector/pgvector.git
cd pgvector
make
make install
```

To run all tests:

```
make installcheck        # regression tests
make prove_installcheck  # TAP tests
```

To run single tests:

```
make installcheck REGRESS=functions                            # regression test
make prove_installcheck PROVE_TESTS=test/t/001_ivfflat_wal.pl  # TAP test
```

To enable assertions:

```
make clean && PG_CFLAGS="-DUSE_ASSERT_CHECKING" make && make install
```

To enable benchmarking:

```
make clean && PG_CFLAGS="-DIVFFLAT_BENCH" make && make install
```

To show memory usage:

```
make clean && PG_CFLAGS="-DHNSW_MEMORY -DIVFFLAT_MEMORY" make && make install
```

To get k-means metrics:

```
make clean && PG_CFLAGS="-DIVFFLAT_KMEANS_DEBUG" make && make install
```

Resources for contributors

- Extension Building Infrastructure
- Index Access Method Interface Definition
- Generic WAL Records