pgvector

Open-source vector similarity search for Postgres

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

- exact and approximate nearest neighbor search
- L2 distance, inner product, and cosine 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

Compile and install the extension (supports Postgres 11+)

```
cd /tmp
git clone --branch v0.5.1 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, or conda-forge, and it comes preinstalled with Postgres.app and many hosted providers

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 (`<#>`) and cosine distance (`<=>`)

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);
```

Insert vectors

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

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;
```

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:

- IVFFlat
- HNSW - added in 0.5.0

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);
```

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);
```

Vectors with up to 2,000 dimensions can be indexed.

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;
```

HNSW

An HNSW index creates a multilayer graph. It has slower build times and uses more memory than IVFFlat, but has better query performance (in terms of speed-recall tradeoff). There's no training step like IVFFlat, so the index can be created without any data in the table.

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

L2 distance

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

Inner product

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

Cosine distance

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

Vectors with up to 2,000 dimensions can be indexed.

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
```

Query Options

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

```sql
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

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

Indexing Progress

Check indexing progress with Postgres 12+

```sql
SELECT phase, tuples_done, tuples_total FROM pg_stat_progress_create_index;
```

The phases are:

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

Note: `tuples_done` and `tuples_total` are only populated during the `loading tuples` phase

Filtering

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

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

Create an index on one or more of the `WHERE` columns for exact search

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

Or a partial index on the vector column for approximate search

```sql
CREATE INDEX ON items USING ivfflat (embedding vector_l2_ops) WITH (lists = 100)
    WHERE (category_id = 123);
```

Use partitioning for approximate search on many different values of the `WHERE` columns

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

Hybrid Search

Use together with Postgres full-text search for hybrid search (Python example).

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

Performance

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);
```

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-cpp
C#	pgvector-dotnet
Crystal	pgvector-crystal
Dart	pgvector-dart
Elixir	pgvector-elixir
Go	pgvector-go
Haskell	pgvector-haskell
Java, Scala	pgvector-java
Julia	pgvector-julia
Lua	pgvector-lua
Node.js	pgvector-node
Perl	pgvector-perl
PHP	pgvector-php
Python	pgvector-python
R	pgvector-r
Ruby	pgvector-ruby, Neighbor
Rust	pgvector-rust
Swift	pgvector-swift

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'll need to use dimensionality reduction at the moment.

Troubleshooting

Why isn't a query using an index? The cost estimation in pgvector < 0.4.3 does not always work well with the planner. You can encourage the planner to use an index for a query with:

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

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 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;
```

Reference

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
<->	Euclidean distance	
<#>	negative inner product	
<=>	cosine distance	

Vector Functions

Function	Description	Added
cosine_distance(vector, vector) → double precision	cosine distance	
inner_product(vector, vector) → double precision	inner product	
l2_distance(vector, vector) → double precision	Euclidean distance	
l1_distance(vector, vector) → double precision	taxicab distance	0.5.0
vector_dims(vector) → integer	number of dimensions	
vector_norm(vector) → double precision	Euclidean norm	

Aggregate Functions

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

Installation Notes

Postgres Location

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

```
export PG_CONFIG=/Applications/Postgres.app/Contents/Versions/latest/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
```

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-15
```

Note: Replace `15` with your Postgres server version

Windows

Support for Windows is currently experimental. 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\15"
git clone --branch v0.5.1 https://github.com/pgvector/pgvector.git
cd pgvector
nmake /F Makefile.win
nmake /F Makefile.win install
```

Additional Installation Methods

Docker

Get the Docker image with:

```
docker pull ankane/pgvector
```

This adds pgvector to the Postgres image (run it the same way).

You can also build the image manually:

```
git clone --branch v0.5.1 https://github.com/pgvector/pgvector.git
cd pgvector
docker build --build-arg PG_MAJOR=15 -t myuser/pgvector .
```

Homebrew

With Homebrew Postgres, you can use:

```
brew install pgvector
```

Note: This only adds it to the `postgresql@14` formula

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-15-pgvector
```

Note: Replace `15` 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_15
# or
sudo dnf install pgvector_15
```

Note: Replace `15` with your Postgres server version

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. 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

0.4.0

If upgrading with Postgres < 13, remove this line from `sql/vector--0.3.2--0.4.0.sql`:

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

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

0.3.1

If upgrading from 0.2.7 or 0.3.0, recreate all `ivfflat` indexes after upgrading to ensure all data is indexed.

```
-- Postgres 12+
REINDEX INDEX CONCURRENTLY index_name;

-- Postgres < 12
CREATE INDEX CONCURRENTLY temp_name ON table USING ivfflat (column opclass);
DROP INDEX CONCURRENTLY index_name;
ALTER INDEX temp_name RENAME TO index_name;
```

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_wal.pl  # TAP test
```

To enable benchmarking:

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

Resources for contributors

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