# Vector columns

Supabase offers a number of different ways to store and query vectors within Postgres. The SQL included in this guide is applicable for clients in all programming languages. If you are a Python user see your Python client options after reading the `Learn` section.

Vectors in Supabase are enabled via pgvector, a PostgreSQL extension for storing and querying vectors in Postgres. It can be used to store embeddings.

## Usage

### Enable the extension

Dashboard SQL

1. Go to the Database page in the Dashboard.
2. Click on **Extensions** in the sidebar.
3. Search for "vector" and enable the extension.

### Create a table to store vectors

After enabling the `vector` extension, you will get access to a new data type called `vector`. The size of the vector (indicated in parenthesis) represents the number of dimensions stored in that vector.

```
1 create table documents (
```

```
  id serial primary key,
  title text not null,
  body text not null,
  embedding vector(384)
);
```

In the above SQL snippet, we create a `documents` table with a column called `embedding` (note this is just a regular Postgres column - you can name it whatever you like). We give the `embedding` column a `vector` data type with 384 dimensions. Change this to the number of dimensions produced by your embedding model. For example, if you are generating embeddings using the open source `gte-small` model, you would set this number to 384 since that model produces 384 dimensions.

> In general, embeddings with fewer dimensions perform best. See our analysis on fewer dimensions in pgvector.

## Storing a vector / embedding

In this example we'll generate a vector using Transformers.js, then store it in the database using the Supabase JavaScript client.

```
import { pipeline } from '@xenova/transformers'
const generateEmbedding = await pipeline('feature-extraction', 'Supabase/gte-small')

const title = 'First post!'
const body = 'Hello world!'

// Generate a vector using Transformers.js
const output = await generateEmbedding(body, {
  pooling: 'mean',
  normalize: true,
})

// Extract the embedding output
const embedding = Array.from(output.data)

// Store the vector in Postgres
const { data, error } = await supabase.from('documents').insert({
  title,
  body,
  embedding,
})
```

```
});
```

This example uses the JavaScript Supabase client, but you can modify it to work with any supported language library.

## Querying a vector / embedding

Similarity search is the most common use case for vectors. `pgvector` support 3 new operators for computing distance:

| Operator | Description |
| --- | --- |
| `<->` | Euclidean distance |
| `<#>` | negative inner product |
| `<=>` | cosine distance |

Choosing the right operator depends on your needs. Dot product tends to be the fastest if your vectors are normalized. For more information on how embeddings work and how they relate to each other, see What are Embeddings?.

Supabase client libraries like `supabase-js` connect to your Postgres instance via PostgREST. PostgREST does not currently support `pgvector` similarity operators, so we'll need to wrap our query in a Postgres function and call it via the `rpc()` method:

```sql
create or replace function match_documents (
  query_embedding vector(384),
  match_threshold float,
  match_count int
)
returns table (
  id bigint,
  title text,
  body text,
  similarity float
)
language sql stable
as $$
  select
    documents.id,
```

```
    documents.title,
    documents.body,
    1 - (documents.embedding <=> query_embedding) as similarity
  from documents
  where 1 - (documents.embedding <=> query_embedding) > match_threshold
  order by (documents.embedding <=> query_embedding) asc
  limit match_count;
$$;
```

This function takes a `query_embedding` argument and compares it to all other embeddings in the `documents` table. Each comparison returns a similarity score. If the similarity is greater than the `match_threshold` argument, it is returned. The number of rows returned is limited by the `match_count` argument.

Feel free to modify this method to fit the needs of your application. The `match_threshold` ensures that only documents that have a minimum similarity to the `query_embedding` are returned. Without this, you may end up returning documents that subjectively don't match. This value will vary for each application - you will need to perform your own testing to determine the threshold that makes sense for your app.

If you index your vector column, ensure that the `order by` sorts by the distance function directly (rather than sorting by the calculated `similarity` column, which may lead to the index being ignored and poor performance).

To execute the function from your client library, call `rpc()` with the name of your Postgres function:

```
const { data: documents } = await supabaseClient.rpc('match_documents', {
  query_embedding: embedding, // Pass the embedding you want to compare
  match_threshold: 0.78, // Choose an appropriate threshold for your data
  match_count: 10, // Choose the number of matches
})
```

In this example `embedding` would be another embedding you wish to compare against your table of pre-generated embedding documents. For example if you were building a search engine, every time the user submits their query you would first generate an embedding on the search query itself, then pass it into the above `rpc()` function to match.

> Be sure to use embeddings produced from the same embedding model when calculating distance. Comparing embeddings from two different models will produce no meaningful result.

Vectors and embeddings can be used for much more than search. Learn more about embeddings at What are Embeddings?.

## Indexes

Once your vector table starts to grow, you will likely want to add an index to speed up queries. See Vector indexes to learn how vector indexes work and how to create them.

Edit this page on GitHub