# What's our Vector, Victor?

## Taking the pain out of AI with pg_vectorize

Shaun Thomas
Software Engineer, pgEdge

March 7th, 2025

pgEdge

# Airplane!

# Who are pgEdge?

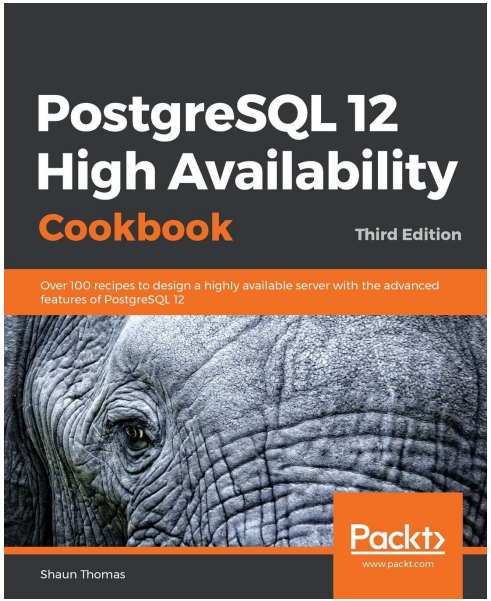
- Distributed Postgres
- Active-Active clusters
- Cloud Services
- Platform Automation
- Ultra High Availability

www.pgedge.com

# Who am I?

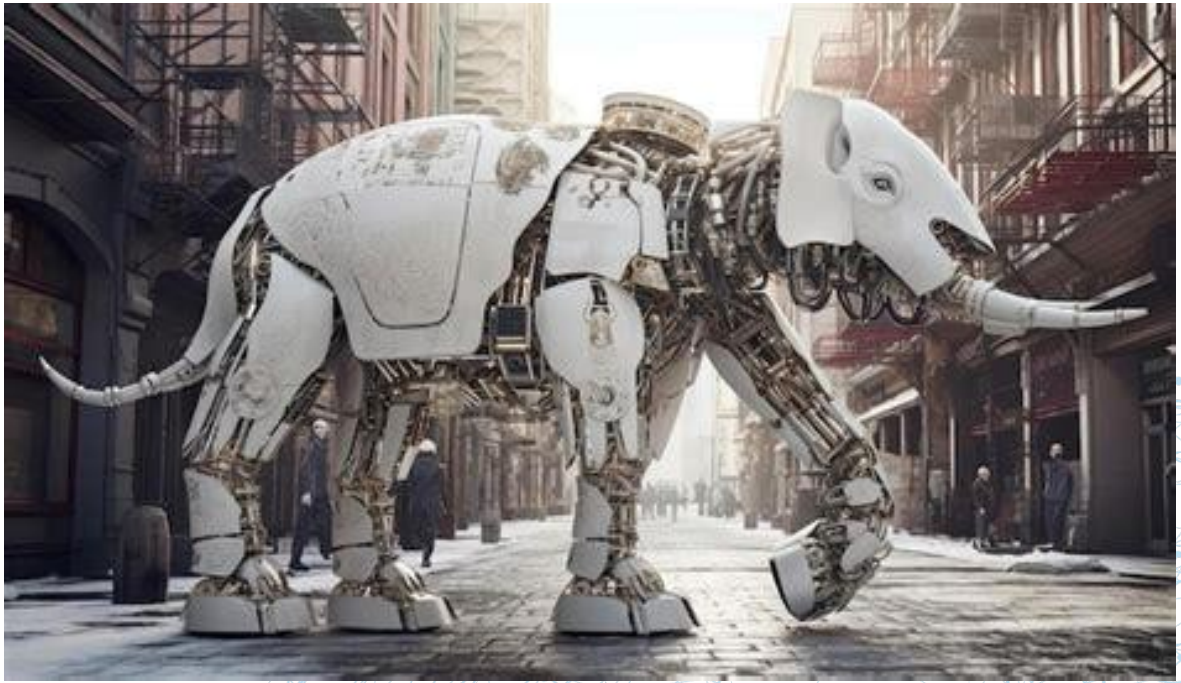
- Author
- Speaker
- Blogger
- Mentor
- Dev

shaun.thomas@
pgedge.com



# What is AI?

# Not This!

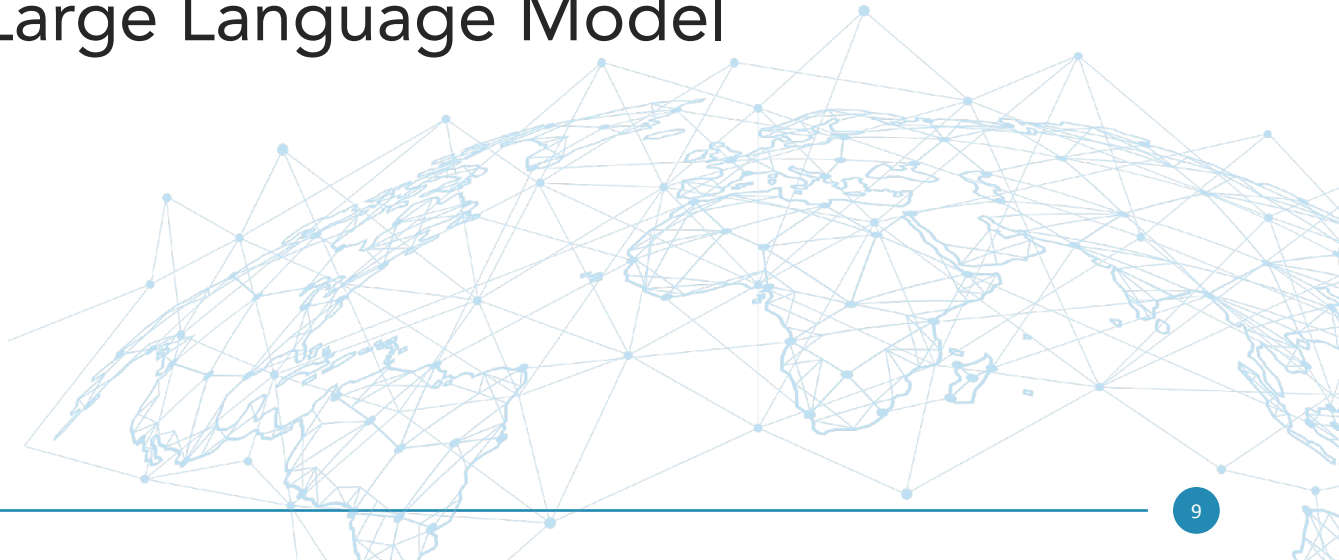


# What AI Really is

# The Lingua Franca

# LLM

Large Language Model



# RAG

Retrieval Augmented Generation

# Token

Word chunk

# Root and Stem

## Augmented Stemming

- Reduce a word to its base
- Include language metadata
- Retain inter-token relationships
- Treat as a monad

## Tokenize “cat”

- cat
- cataclysm
- catacomb
- catalog
- catalyst
- catapult
- catchup

## Tokenize “dog”

- dog
- dogged
- doggone
- doggy
- doghouse
- dogma

# Embedding

Big-ass array

# Embedding

~~Big-ass array~~

# Embedding

Vector to token coordinates

# Coordinates to Where?

# Bits and Pieces

# Postgres

Your favorite database engine

https://www.postgresql.org

# pgvector

Bestows vector abilities to Postgres

https://github.com/pgvector/pgvector

# Added by pgvector

- Vector similarity searches
- Multiple new vector types (single, half, binary, sparse)
- Vector distance operators (`<->`, `<#>`, `<=>`, `<+>`, `<~>`, `<%>`)
- New vector index types (HNSW, IVF-FLAT)

# pg_vectorize

Makes Postgres an AI powerhouse

https://github.com/tembo-io/pg_vectorize

# Parts of pg_vectorize

pg_vectorize is a combination of 3 extensions:

- pgvector (duh)
- pgmq - for queueing embedding jobs
- pg_cron - for those who'd rather wait

And its own functionality

# What Does pg_vectorize Do?

# Transform Individual Phrases

```
SELECT vectorize.encode(
    input  => 'Is Postgres the best database engine?',
    model  => 'sentence-transformers/all-MiniLM-L12-v2'
);
```

- Easily transform a prompt to search terms in compatible vectors
- Output is compatible with pgvector vector search

# Create and Maintain Embeddings

```
SELECT vectorize.table(
    job_name      => 'rt_article_embed',
    "table"       => 'blog_article',
    primary_key   => 'article_id',
    update_col    => 'last_updated',
    columns       => ARRAY['author', 'title', 'content'],
    transformer   => 'sentence-transformers/all-MiniLM-L12-v2',
    schedule      => 'realtime'
);
```

Embeddings are maintained by pg_cron job, or pgmq live updates

# Important Latency Note!

"Realtime" spawns embeddings via queue

This dramatically reduces write latency!

# Search Content Semantically

```
SELECT vectorize.search(
    job_name        => 'rt_article_embed',
    query           => 'Is Postgres the best database engine?',
    return_columns  => ARRAY['author', 'title', 'content'],
    num_results     => 5
);
```

Automatically uses the same transformer as existing embeddings

# Interrogate an LLM

```
SELECT vectorize.generate(
    input  => 'Is Postgres the best database?',
    model  => 'ollama/llama3.1'
);
```

Good for quick one-off responses for various purposes

# Natural Language Search

```
SELECT * FROM vectorize.search(
    job_name        => 'rt_article_embed',
    query           => 'Is Postgres the best database engine?',
    return_columns  => ARRAY['author', 'title', 'content'],
    num_results     => 5
);
```

Consider this like Full Text Search, but better

# Bootstrap a RAG Stack

```
SELECT vectorize.init_rag(
    agent_name          => 'rt_article_embed',
    table_name          => 'blog_article',
    "column"            => 'article',
    unique_record_id    => 'article_id',
    transformer         => 'sentence-transformers/all-MiniLM-L12-v2',
    schedule            => 'realtime'
);
```

Realtime embeddings are queued to avoid write latency

# Perform a RAG Request

```
SELECT vectorize.rag(
    agent_name  => 'blog_chat',
    query       => 'Is Postgres the best database?',
    chat_model  => 'ollama/llama3.1'
) -> 'chat_response';
```

The result is a JSON object that includes context if we need it

# Works with OpenAI

Just supply your OpenAI token:

```
ALTER SYSTEM SET vectorize.openai_key TO '<your api key>';
```

# Or Roll Your Own

Search using Ollama or vLLM instead:

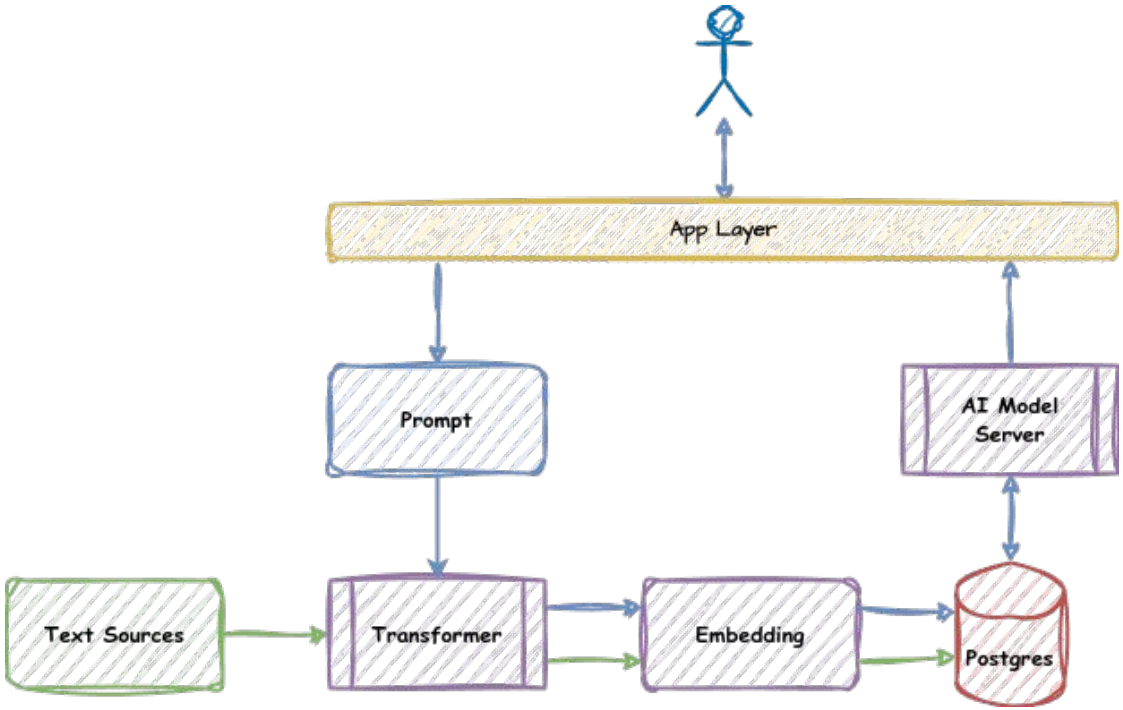
```
ALTER SYSTEM SET vectorize.openai_service_url
   TO 'https://api.myserver.com/v1';
```

Use a custom transformer service:

```
ALTER SYSTEM SET vectorize.embedding_service_url
   TO 'https://api.myserver.com/v1';
```

# How Does RAG Work?

# Anatomy of a RAG App



# How it Works

## Data Side

1. Gather content
2. Pass through a transformer
3. Store vector in database

## User Side

1. Asks a question
2. Pass through transformer
3. Match against stored vectors
4. Question + results sent to AI
5. Send answer to user

# The Full Monty

To build a RAG app, we need to:

1. Parse and load the content and metadata into Postgres
2. Generate the embeddings and save in Postgres
3. Transform user input into an embedding
4. Match results from user search vector
5. Build new prompt from results and user search
6. Send full instructions to model server
7. Return results to user

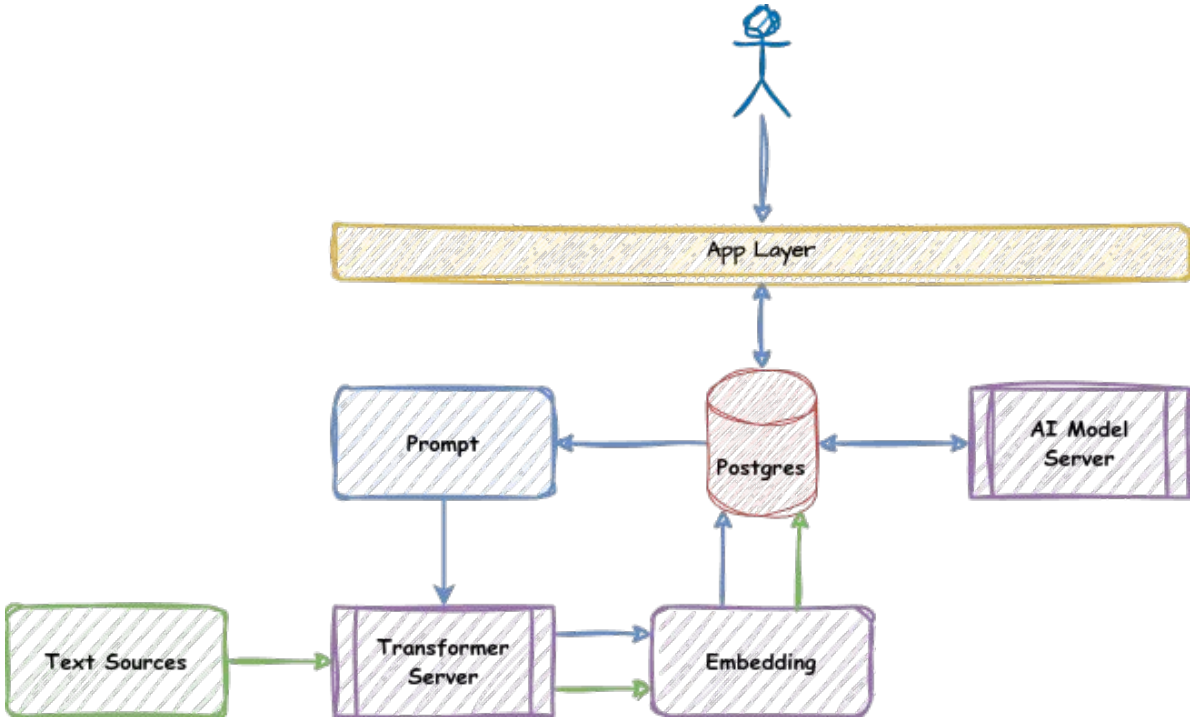
# From the Perspective of pg_vectorize

Or if we're using pg_vectorize:

1. Parse and load the content and metadata into Postgres
2. Call **`vectorize.init_rag( … )`**
3. Call **`vectorize.rag( … )`**

Which would you rather do?

# Better, Faster, Stronger



# Edge Cases

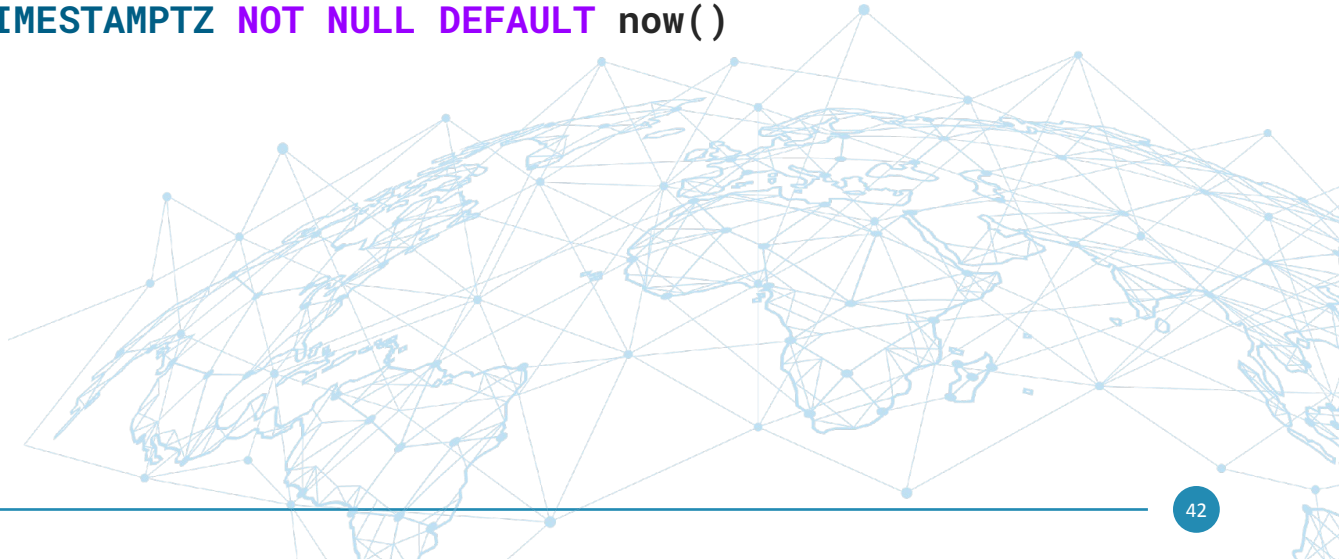
The AI model and transformer servers can be *local*

- Use anything API compatible with OpenAI or OLLaMa
- Now data never leaves your local network
- No latency to remote model servers
- No need to return source matches to the app layer

# Let's Make a RAG APP

# A Place for Blogs

```
CREATE TABLE blog_articles (
    article_id    BIGINT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    author        TEXT,
    title         TEXT,
    content       TEXT,
    publish_date  DATE,
    last_updated  TIMESTAMPTZ NOT NULL DEFAULT now()
);
```



# Chunky Style

```
CREATE TABLE blog_article_chunks (
    chunk_id      BIGINT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    article_id    BIGINT NOT NULL REFERENCES blog_articles,
    chunk         TEXT,
    last_updated  TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

- Embeddings are usually “fuzzy” (only 384 coordinates)
- We need chunks for sharper context

# More than Meets the Eye

```
SELECT vectorize.init_rag(
    agent_name          => 'blog_chat',
    table_name          => 'blog_article_chunks',
    "column"            => 'chunk',
    unique_record_id    => 'chunk_id',
    transformer         => 'sentence-transformers/all-MiniLM-L12-v2',
    schedule            => 'realtime'
);
```

Look familiar? Now we're indexing chunks rather than full articles.

# Slice and Dice

Here's a closer look at a chunk splitter in Python:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    separators = ["\n\n", "\n", ' ', '.', '```'],
    chunk_size = 500,
    chunk_overlap = 20,
    length_function = len,
    is_separator_regex = False
)

def chunk_content(content):
    return splitter.split_text(content)
```

# Where are the Embeddings?

Let's check our schema:

```
postgres=# \dt
                List of relations
 Schema |         Name          | Type  |  Owner
--------+-----------------------+-------+----------
 public | blog_article_chunks | table | postgres
 public | blog_articles       | table | postgres
```

# Where are the Embeddings?

```
postgres=# set search_path to vectorize;
postgres=# \dt
                  List of relations
  Schema   |          Name           | Type  |  Owner
-----------+-------------------------+-------+----------
 vectorize | _embeddings_blog_chat   | table | postgres
 vectorize | example_products        | table | postgres
 vectorize | job                     | table | postgres
 vectorize | prompts                 | table | postgres
```

# Another Brick in the Wall

```
postgres=# \d _embeddings_blog_chat
                Table "vectorize._embeddings_blog_chat"
   Column   |           Type           | Collation | Nullable | Default
------------+--------------------------+-----------+----------+---------
 chunk_id   | bigint                   |           | not null |
 embeddings | public.vector(384)       |           | not null |
 updated_at | timestamp with time zone |           | not null | now()
```

# Trigger me Timbers

On our blog_article_chunks table

```
Triggers:
    vectorize_insert_trigger_blog_chat AFTER INSERT ON
blog_article_chunks REFERENCING NEW TABLE AS new_table FOR EACH
STATEMENT EXECUTE FUNCTION vectorize.handle_update_blog_chat()
    vectorize_update_trigger_blog_chat AFTER UPDATE ON
blog_article_chunks REFERENCING NEW TABLE AS new_table FOR EACH
STATEMENT EXECUTE FUNCTION vectorize.handle_update_blog_chat()
```

# What is handle_update_blog_chat ?

This chunk calls a rust function:

```
PERFORM vectorize._handle_table_update(
    'blog_chat',
    record_id_array::TEXT[],
    inputs_array
);
```

# Starting to get Rusty

```
-[ RECORD 1 ]-------+---------------------------------------------
Schema              | vectorize
Name                | _handle_table_update
Result data type    | void
Argument data types | job_name text, record_ids text[], inputs text[]
Type                | func
Volatility          | volatile
Parallel            | unsafe
Owner               | postgres
Security            | invoker
Access privileges   |
Language            | c
Source code         | _handle_table_update_wrapper
Description         |
```

# Please Form an Orderly Queue

We see this fateful line:

```
let query = "select pgmq.send($1, $2::jsonb);";
```

# What's in the Queue?

Oh… you know… stuff

```
-[ RECORD 1 ]-------------------------------------
msg_id      | 44
read_ct     | 1
enqueued_at | 2025-02-28 20:49:34.212479+00
archived_at | 2025-02-28 20:49:42.824572+00
vt          | 2025-02-28 20:52:42.761824+00
message     | {... huge JSON blob ...}
```

# What Does an Embedding Look Like?


[-0.02595623,0.04631714,-0.053539883,0.011895365,0.0758225,-0.04304593,-0.006637965,-0.08208234,-0.04918979,-0.020363959,-0.038359903,0.01744871,-0.057595164,0.034
587763,-0.020651337,0.002429941,0.0018788559,-0.018510725,-0.09920806,0.12411486,-0.09942987,0.038612444,0.057046242,-0.015014563,0.03681107,0.029042058,-0.0561162
35,-0.007918157,0.06834828,-0.027709357,-0.012434633,-0.0062096403,-0.015024162,-0.0882817,-0.010005957,0.0217961,0.020747224,0.00043326707,0.029898426,0.063303724
,-0.023971524,-0.035034273,0.12894247,0.03956573,0.04099617,-0.036992185,0.039790176,-0.038303692,0.03762054,-0.016138878,-0.026407361,0.010406044,0.031098412,-0.0
59915572,0.0296487,0.018585488,-0.0127668455,0.0698031,-0.023116358,-0.03830573,0.058555316,0.053015016,0.009442912,0.065988995,-0.025956836,0.0072427755,-0.035602
763,0.049767125,0.027460659,0.011989594,1.022185e-05,-0.04103233,-0.017008793,-0.026518255,-0.057895917,0.02913324,-0.007884655,-0.036250923,0.018677657,-0.0518168
77,0.036574055,-0.018310225,0.10684758,0.015361703,-0.0068149795,-0.002467204,0.045794293,-0.03188524,-0.014328101,-0.04377825,-0.02258047,-0.05837506,0.008181678,
-0.07910704,0.03463214,-0.020189477,-0.092740774,-0.0002254515,-0.00661493,0.1312322,0.02023139,0.016226936,0.050397724,0.0049572135,0.009400744,-0.045763697,-0.07
1638376,-0.014594109,-0.018446293,0.028820504,0.0023369463,0.053181294,0.058653817,-0.06454964,0.049355283,0.07178324,0.027783332,-0.067031115,-0.06841928,0.015850
065,-0.002914686,0.009294329,-0.078147724,-0.01781891,-0.07263269,0.017262291,-0.0061519933,0.014498569,0.07934687,-0.011039961,-0.014350844,0.009714252,0.07571004
,-0.059741423,-0.061780307,-0.07044488,0.0017138183,-0.03665142,0.06618329,-0.056741964,0.024425812,0.043776017,-0.05947509,0.02473815,-0.033279914,0.06721659,0.01
2232149,0.0015699323,0.007885537,0.00707865,0.013194744,-0.068191566,-0.12272909,0.06650073,-0.02412729,0.04940419,0.08976135,0.016346294,-0.042974483,0.0075128144
,0.13506782,0.013340274,0.013941901,-0.0135494545,0.019012375,-0.045056634,-0.024806282,-0.025400957,0.009210025,-0.085539885,-0.0014276546,-0.047662564,0.02840303
4,-0.031291023,0.00994239,0.013966853,0.029291267,-0.06537566,-0.0023040709,-0.022339806,0.05957562,0.0032288802,-0.026567612,0.054026626,0.07418133,-0.11601187,0.
14578743,0.06701949,0.089334145,0.013379732,0.039292034,-0.029553873,0.020182345,-0.027620139,0.033731233,0.029958928,0.021263465,0.0116131,0.024114138,0.036053922
,0.010862184,-0.11032744,0.029497253,0.03680072,0.015323135,-0.02569687,0.020646175,-0.00309678,0.075037666,-0.012467476,-0.012603479,0.05536957,0.06923356,0.04137
6483,-0.05493469,0.07284344,-0.0024210871,0.024228476,-0.054416776,0.09758099,0.015991757,-0.026029492,0.005204354,2.1359543e-32,0.02700274,-0.06537937,-0.05798246
7,-0.058108676,0.024990669,0.008049355,0.016007772,-0.019222062,0.055540632,0.014360761,0.02189043,-0.039927147,-0.06621141,-0.007778538,-0.032505617,-0.015146801,
0.030141199,0.047050603,-0.0278275,0.04865551,0.07719417,-0.048471287,-0.069588214,-0.050331596,0.041957315,0.12916774,0.10859817,0.009190485,-0.05403324,-0.085586
93,0.04856777,0.010237227,-0.09778996,0.032434497,-0.05686069,0.11311847,0.0040654135,-0.055423062,0.044098742,-0.08351652,-0.0066194735,0.0051483805,-0.013018369,
0.09141706,-0.011138346,0.03484014,-0.09798947,0.009890583,0.052184697,-0.016177202,-0.12128752,-0.05317396,-0.038664415,0.053813018,0.025762321,-0.010391627,-0.02
7447335,-0.09687913,-0.040417686,0.05761224,-0.0049005016,-0.03860952,-0.10431886,0.09482661,0.08394817,-0.05782826,-0.023384307,-0.033743203,0.01319146,0.02000948
4,-0.06339285,0.008339009,-0.10972377,-0.09203553,0.02314593,-0.026981864,-0.0098597845,-0.00695105,-0.04888554,-0.054383095,-0.0033409367,-0.016765589,-0.02066507
2,0.03518574,-0.0975508,0.03954543,-0.027971495,0.022485066,-0.03068828,0.044939965,0.014050996,-0.02814454,-0.056048892,-0.027148627,-0.022608032,7.592085e-32,0.0
24655852,-0.03308234,-0.119617596,-0.020011967,0.08908686,-0.10233242,0.041305285,-0.019912839,0.008432649,0.08246976,-0.007695544,-0.013220983,0.04306117,-0.06137
5756,0.10317889,-0.0032164725,-0.06101632,-0.054768626,0.06190977,0.020685453,0.091767095,-0.030094603,-0.010625265,0.011956352,-0.001202916,0.081404224,0.00017668
601,0.053858735,0.11105762,0.03965099,0.055190314,-0.0008298795,-0.03585047,0.02358887,-0.07300523,-0.09976991,0.04071222,-0.017766878,0.083444,-0.014780061,0.1179
988,-0.047808193,0.027711963,0.010073332,0.06527614,-0.081142455,-0.04021762,0.07025154,0.06898177,-0.022367012,-0.06016291,0.020527564,-0.0048388843,-0.015055914,
0.06347836,-0.028675102,-0.04353604,0.0039767306,0.0139750345,-0.10406179,0.03652024,0.05376024,-0.07579619,0.003702582]

# Remember This?

# A Pleasing Result

```
SELECT vectorize.rag(
    agent_name => 'blog_chat',
    query      => 'Is Postgres the best database?',
    chat_model => 'ollama/llama3.1'
) -> 'chat_response';
```

```
"Four times since 2017, it has won the DB-Engines \"DBMS of the Year\"
award."
```

# What Happens Without RAG?

```
SELECT vectorize.generate(
    input  => 'Is Postgres the best database?',
    model  => 'ollama/llama3.1'
);
```

```
Postgres (also known as PostgreSQL) is an excellent database engine,
but whether it's the "best" depends on your specific needs.
```

# Advanced Techniques

# How Do Window Functions Work?

```
SELECT
  (jsonb_array_elements(chat_results->'context'))->'content' as chunk
FROM
  vectorize.rag(
    agent_name => 'blog_chat',
    query      => 'How do window functions work?',
    chat_model => 'ollama/llama3.1'
  ) -> 'chat_response';
```

# Is This Really The Best Augmentation?

**"The more advanced use cases for window functions are a topic for another day. Consider this a very high-level introduction to how they work and their inherent limitations instead of a comprehensive guide. There's a lot of material here that deserves closer inspection, so there's no need to rush. Either way, don't let window functions confuse you more than necessary. Like any independent agent, you just need to know what they're doing behind the scenes."**

**"with window functions." <- I need to work on my chunker!**

# What Does the User Really Want?

You are a front-end to a retrieval augmented generation search. Rewrite this user prompt into an appropriate series of semantic search terms to match against a corpus of reference documents related to PostgreSQL which has been indexed with a simple transformer assuming low token context granularity and small extract chunks less than 1024 tokens. Do not try to answer the question yourself, produce only the appropriate search revision, and do not add introductory text.

Prompt: How do window functions work?

# Perhaps a Better Prompt?

```
postgres window function definition
window function syntax postgres
row over partition by clause
rows between unbounded preceding and current row
window ordering postgres
window frame specification
```

# How Do Window Functions Work?

**"Again, we can learn a few different things from these results. First is that the window results are restricted to the partition we declared. We set the partition to limit sums, ranks, and other window functions to the domain of the location. Postgres will apply window aggregates specifically to each location as it appears in the query output."**

**"Separate windows, separate effects. Of course, we may not actually *want* that to happen. If we end up using the same window over and over again, it doesn't make sense to declare it for each column. Thankfully Postgres has a shorthand for that:"**

# Conclusion!

If you can write queries

You can build AI apps with Postgres

# Questions?