PostgreSQL in the AI Era

PostgreSQL Conference Nepal 2025 | 5–6 May, Kathmandu University

Speaker: Prabin Silwal
https://www.linkedin.com/in/prabin-silwal/

Why This Talk?

1. Rise of AI/ML in web applications (eg: E-commerce)
2. PostgreSQL's increasing role
3. Real-time, scalable, efficient data systems are needed
4. PostgreSQL is More Than Just a Relational DB: JSONB, Extensions (pg_vector, fdw)

Focused Topics

1. Use of **pgvector** for AI
2. High-Performance Workloads in Postgres with partitioning

Overview of the Talk

1. Introduction to vector search
2. What is pgvector extension
3. Vector Search with pgvector
4. Use case in AI
5. Scaling Challenges
6. Advantages of Partitioning
7. Compare with Vector database
8. Q & A

01

Introduction to vector search

What is Vector Search?

- method of finding similar items by comparing vector representations of data instead of using traditional keyword
- commonly used in AI, machine learning, recommendations, semantic search application, document search

Eg: "apple" might be encoded as [0.13, -1.23, 2.34, ...]

Vectors in ML: embeddings (NLP, image, recsys)

Cosine/Euclidean/Inner Product similarity

How Vector Search Works

1. **Embed**: Use a model (like OpenAI, Hugging Face, or CLIP) to convert your data into vectors.

2. **Store**: Save these vectors in a vector-aware database (like PostgreSQL with pgvector, Pinecone,Qdrant,..).

3. **Query**: Convert the user's query into a vector.

4. **Compare**: Use **cosine similarity**, **Euclidean distance**, or **dot product** to find vectors that are closest to the query vector.

02

What is pgvector?

What is pgvector?

pgvector is a PostgreSQL extension

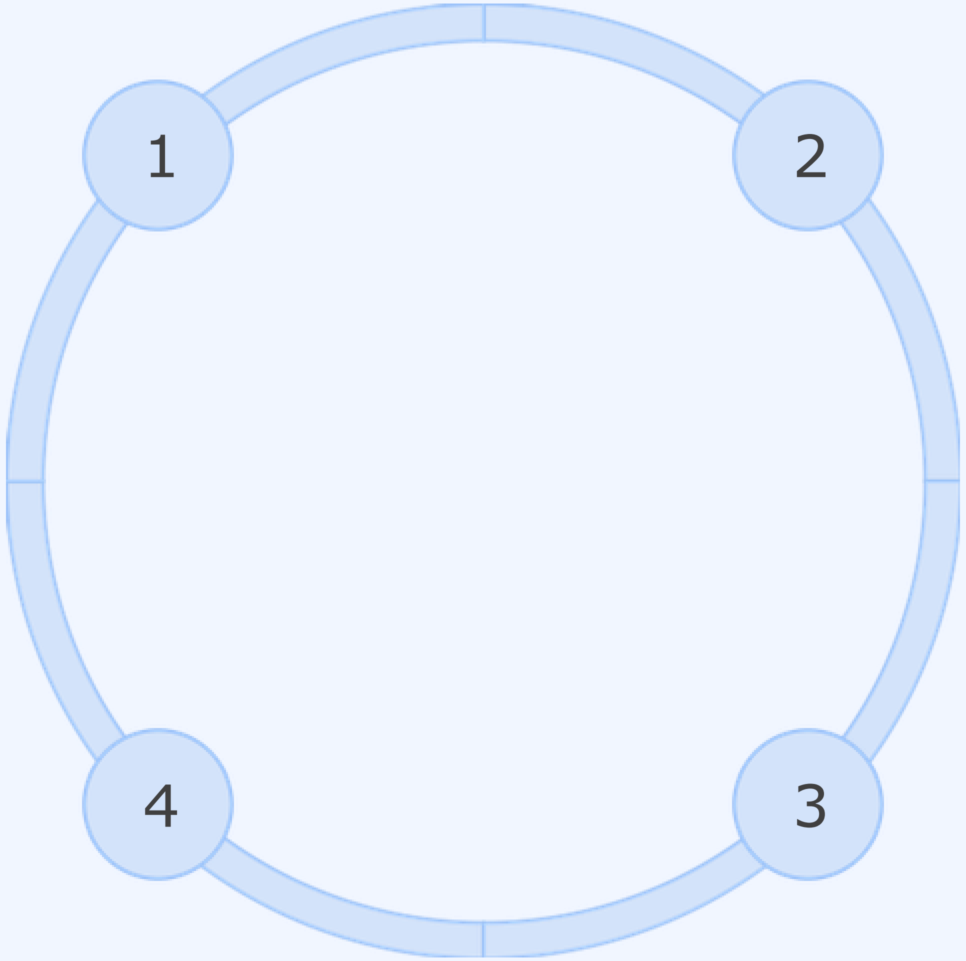
makes it possible to efficiently store, manipulate, and analyze vector data

- Vector Storage
- Similarity Search

AI Integration
NLP: Text embeddings (e.g. OpenAI, SentenceTransformers)
Computer Vision: Image embeddings

pgvector

pgvector is a PostgreSQL extension that brings advanced capabilities for handling high-dimensional vector data.



Vector Storage: Native vector data type for high-dimensional embeddings (e.g., 384-dim).

Similarity Search: Supports cosine, Euclidean, and inner product distances.
Powers: Recommendation systems, clustering, semantic search, k-NN queries.

AI Integration:
•NLP: Text embeddings (e.g., OpenAI, SentenceTransformers).
•Computer Vision: Image embeddings (CNNs).

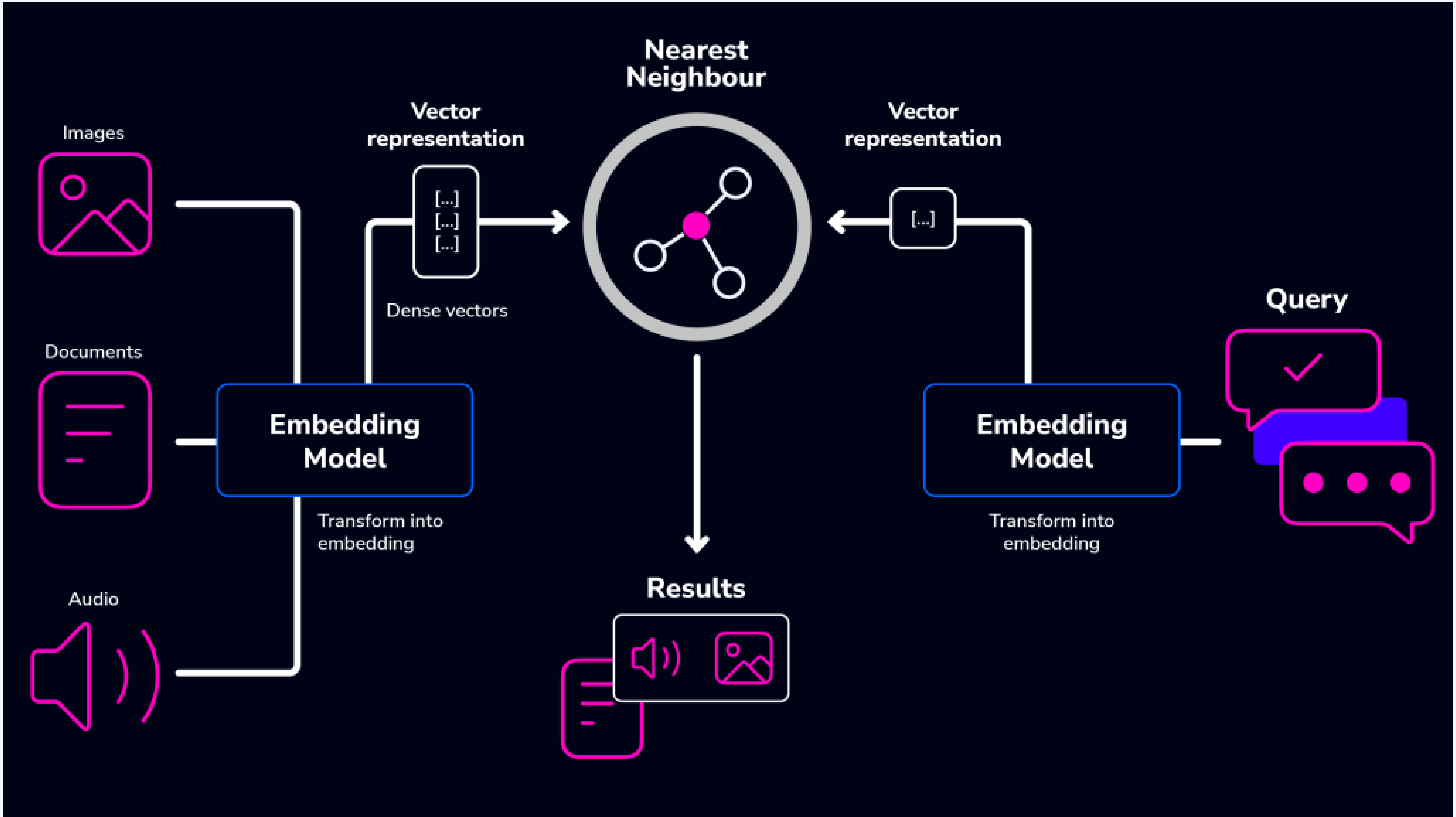
Benefits of pgvector

This extension unlocks PostgreSQL’s potential for applications such as semantic and similarity search, image retrieval, recommendation engines, NLP, and computer vision tasks.

03

Vector Search with pgvector

pgvector: PostgreSQL as a vector database



Vector Search with pgvector

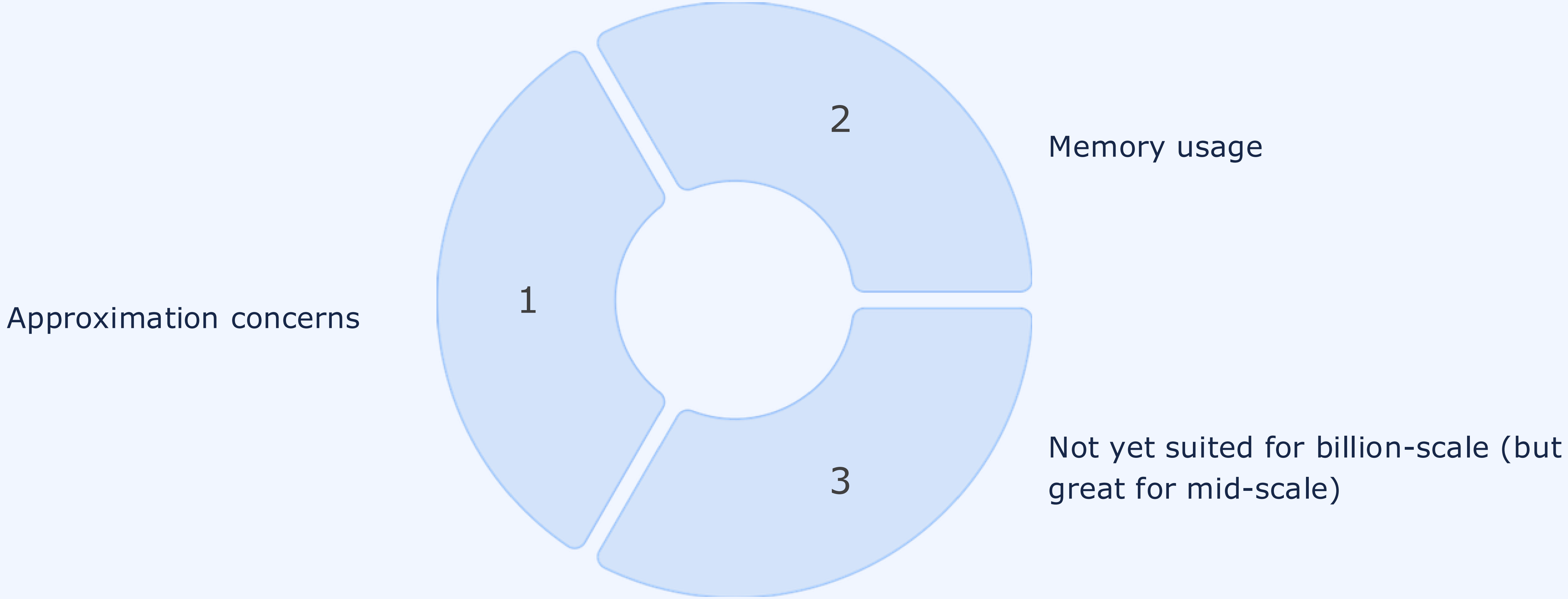
- Storing embeddings and
- Similarity search

Search Code Snippet:

```
# Python + psycopg2 example

query_embedding = model.encode("Nepali traditional dress")
cursor.execute("SELECT name FROM products ORDER BY
                embedding <=> %s::vector LIMIT %s", (query_embedding, limit))
```

pgvector Limitations



04

Use case in AI:
Eg: Recommendation engine for e-commerce

Name	Last commit
.vscode	Adffing
__pycache__	final update with wrapper
docker	Adffing
.gitignore	adding git ignore
Dockerfile	changes made
README.md	Adffing
app.py	changes made
docker-compose.yml	changes made
feed.php	final update with wrapper
feed2.php	final update with wrapper
image_search_cnn-postman_collecti...	Adffing
image_search_cnn.py	Adffing
image_search_open_cv.py	Adffing

README.md

image-search

API ENDPOINTS::

1. /extract_store_features: This endpoint expects a POST request with JSON data containing the image_path and produ extracts features from the provided image, stores the features in the database, and returns a JSON response with a message.
2. /match_features: This endpoint expects a POST request with JSON data containing the image_path. It matches the f the query image with the stored features in the database, and returns a JSON response with the matched product ID their corresponding match percentages.
3. /update_feedback: This endpoint expects a POST request with JSON data containing the query_image_path, correct_matches, and incorrect_matches. It is used to update the matching algorithm based on user feedback. query_image_path: The path or URL of the query image. correct_matches: A list of product IDs that the user confirm correct matches. incorrect_matches: A list of product IDs that the user confirmed as incorrect matches. The endpoi

DEPLOY

DEPENDENCIES:

pip3 install flask requests pillow tensorflow numpy mysql-connector-python

To deploy the Flask application using Gunicorn and Nginx, you can follow these steps: Install Gunicorn and Nginx on your server.

Code

1. app.py (Previously: image_search_cnn_WITH_FLASK.py) Create a new file, let's say app.py, and copy the Flask application code (above code) into it.

Service file

2. Create a systemd service unit file to run Gunicorn. For example, create a file called telaship_image_search.service in the /etc/systemd/system/ directory with the following content:

[Unit] Description=Gunicorn instance to telaship_image_search API After=network.target

[Service] User=root Group=root WorkingDirectory=/var/www/env.telaship_image_search Environment="PATH=/var/www/env.telaship_image_search/bin" ExecStart=/var/www/env.telaship_image_search/bin/gunicorn --workers 3 --error-logfile /var/log/gunicorn/telaship_image_search_api_error.log --access-logfile /var/log/gunicorn/telaship_image_search_api_access.log --bind unix:/var/www/telaship_image_search/api-socket.sock -m 007 --reload api_app:app

[Install] WantedBy=multi-user.target

NOTE: Enable and Start the Gunicorn Service: Run the following commands to enable and start the Gunicorn service:

sudo systemctl enable telaship_image_search sudo systemctl start telaship_image_search

NGINX

3. Configure Nginx to act as a reverse proxy server for the Flask application: Open the Nginx configuration file (/etc/nginx/nginx.conf or /etc/nginx/sites-available/default). Inside the server block, add the following location block to proxy requests to the Gunicorn application:

server { listen 5000; server_name localhost;

```
location / {
    proxy_pass http://unix:/var/www/telaship_image_search/api-socket.sock;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
}
```

}

NOTE: sudo ln -s /etc/nginx/sites-available/telaship_image_search /etc/nginx/sites-enabled/

Save the configuration file and restart Nginx:

sudo service nginx restart

```
def match_images(image_path, gallery_list):
    # Extract features from the query image
    query_features = extract_features(image_path)

    # Dictionary to store matching results
    highest_match_percentage = {}

    # Iterate over stored features and compare with the query features
    for image_feature in features_db.items():
        features = image_feature.features
        similarity = np.dot(query_features, features.T) / \
            (np.linalg.norm(query_features) * np.linalg.norm(features))
        match_percentage = similarity * 100
        if image_feature.product_id not in highest_match_percentage or \
                match_percentage >
highest_match_percentage[image_feature.product_id]:
            highest_match_percentage[image_feature.product_id] =
match_percentage

    # Sort the matched products based on the match percentage (highest to
lowest)
    matched_products = [MatchedFeature(product_id, match_percentage)
                        for product_id, match_percentage in
highest_match_percentage.items()]
return matched_products
```

The Challenge with Traditional Approaches

Previous MySQL Setup:

- Separate systems for storage and search
- Complex ETL pipelines for vector operations
- Multiple tables/dbs for metadata and binary dat
- External search engines for similarity matching

Pain Points:

- High operational complexity
- Performance bottlenecks
    - No index support
    - Expensive I/O
- Difficult to scale

Alternative to above match_images Code :

```
# Python + psycopg2 example

query_embedding = model.encode("Nepali traditional dress")
cursor.execute("SELECT name FROM products ORDER BY
               embedding <=> %s::vector LIMIT %s", (query_embedding, limit))
```

The Challenge with Traditional Approaches

Previous MySQL Setup:

- Separate systems for storage and search
- Complex ETL pipelines for vector operations
- Multiple tables/dbs for metadata and binary data
- External search engines for similarity matching

Pain Points:

- High operational complexity
- Performance bottlenecks
    - No index support
    - Expensive I/O
- Difficult to scale

Old MySQL Approach

Database → ETL Process → Vector DB → Search Engine

New PostgreSQL Approach

PostgreSQL with all capabilities]

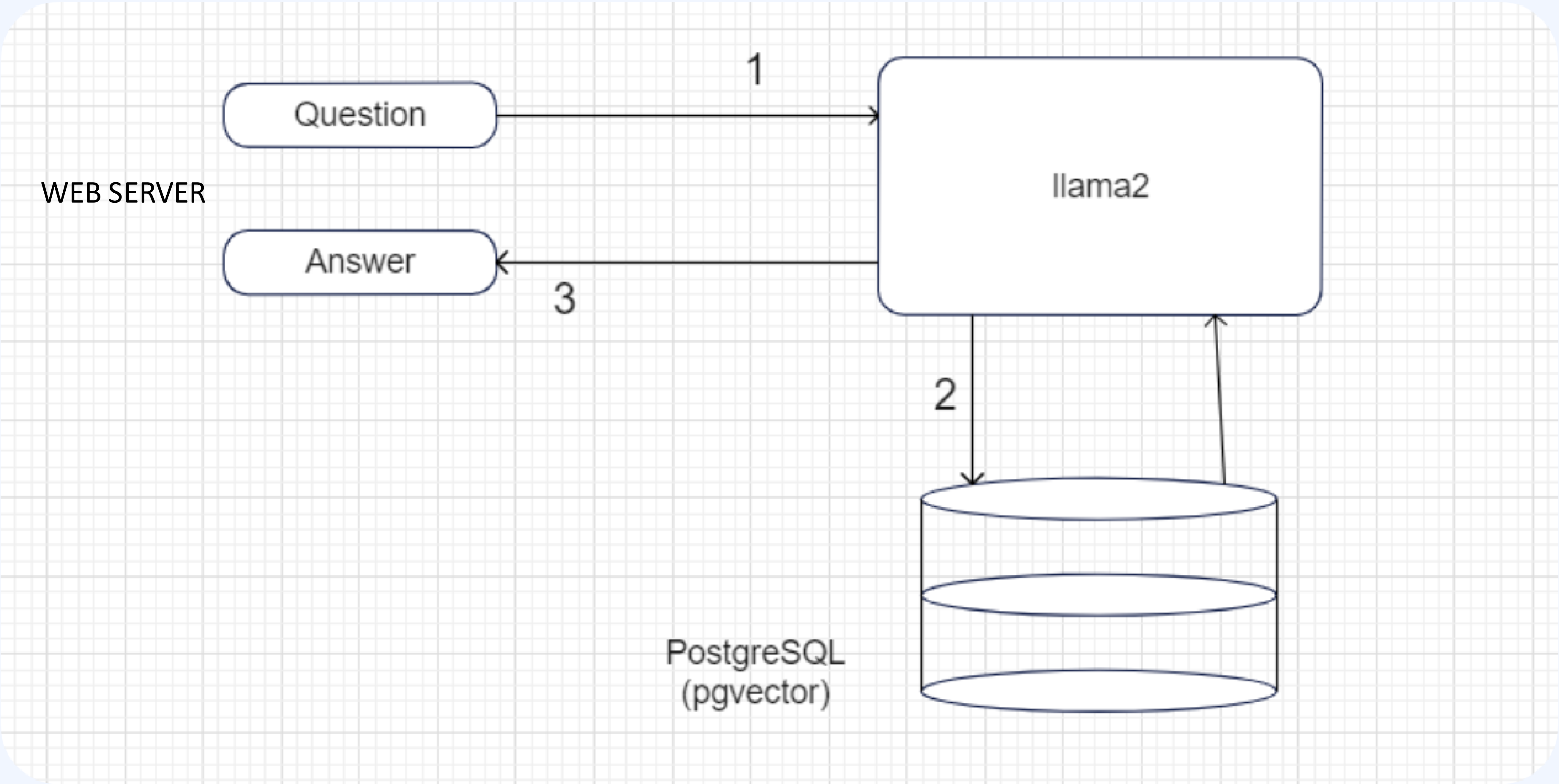
PostgreSQL to the Rescue

Why PostgreSQL for AI Workloads?

- Built-in vector support (pgvector extension)
    - Store vector as native type
    - Index support via ivfflat, hnsw
- JSON/JSONB for flexible schemas
- Full-text search capabilities
- Image/vector similarity search
- All in one database system
- Fast similarity search (<->, cosine, inner product)

Feature	PostgreSQL + pgvector	MySQL
Native vector support	✅ Yes (vector type)	❌ No (requires JSON/array hacks)
Vector similarity search	✅ Yes (L2, Cosine, Inner Product)	❌ No native support
Indexing for ANN search	✅ Yes (IVFFlat, HNSW)	❌ No indexing for vectors
Performance at scale	✅ Sub-second for millions of rows	❌ Full scan, poor at scale
Hybrid filtering + similarity	✅ Supported in single query	❌ Needs external logic
Integration with AI workflows	✅ Strong (LangChain, OpenAI, etc.)	❌ Limited, often external tools
In-database recommendation	✅ Yes (efficient + flexible)	❌ No (complex to implement)
Ease of deployment	✅ One stack, fewer moving parts	❌ Needs external vector DB or search engine
Use in production (2025)	✅ Widely adopted (e.g. PostHog)	❌ Rarely used for vector tasks

Use cases:

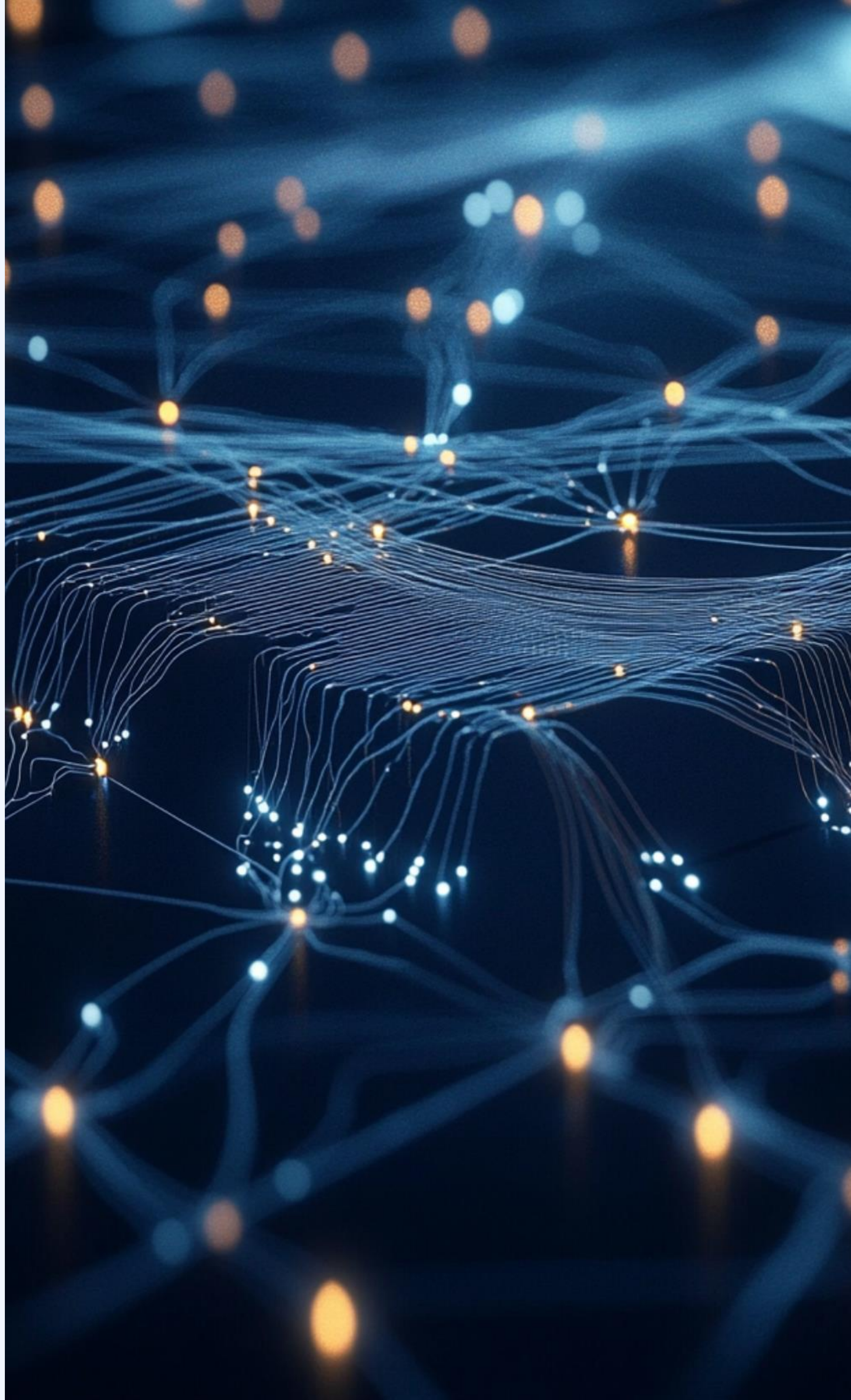

1
Question
llama2
WEB SERVER
Answer
3
2
PostgreSQL
(pgvector)

USE CASE DEMO:

Recommendation engine for e-commerce

```sql
-- Enable pgvector
CREATE EXTENSION IF NOT EXISTS vector;

-- Create a table
CREATE TABLE products (
	id SERIAL PRIMARY KEY,
	name TEXT,
	embedding VECTOR(512)
);
```



Code Snippet

seed_data.py

```
import psycopg2
from sentence_transformers import SentenceTransformer
conn = psycopg2.connect(…)
cur = conn.cursor()
model = SentenceTransformer('all-MiniLM-L6-v2’)
products = ["Dhaka Topi", "Gunyu Cholo", "Khukuri Knife",
"Thangka Painting", "Singing Bowl"]
for product in products:
    embedding = model.encode(product).tolist()
    cur.execute("INSERT INTO products (name, embedding)
VALUES (%s, %s)", (product, embedding))
conn.commit()
print("Data seeded successfully!")
```

Code Snippet

recommendation_engine.py

```
import psycopg2
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2’)
conn = psycopg2.connect(…)
def get_recommendations(query_text, limit=5):
    query_embedding = model.encode(query_text).tolist()
    cur = conn.cursor()
    cur.execute("SELECT name, embedding <=> %s AS distance
FROM products ORDER BY distance ASC LIMIT %s",
(query_embedding, limit))
    return cur.fetchall()

print(get_recommendations("Traditional Nepali dress"))
```

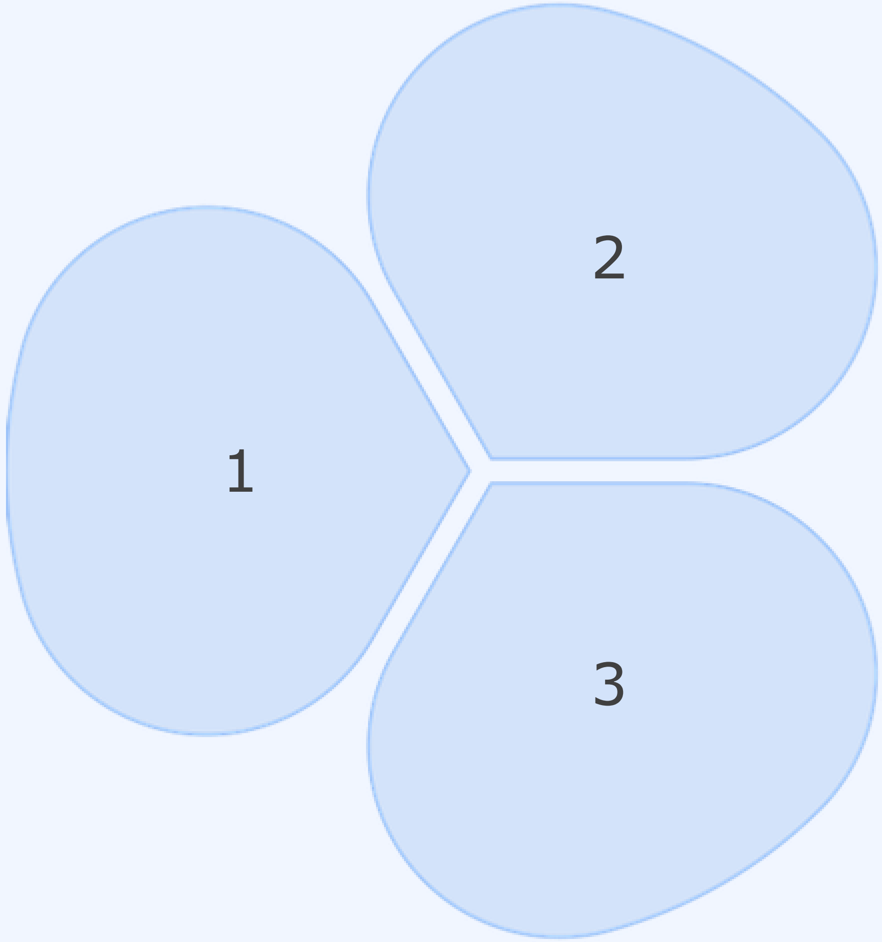
NOTE: Cosine similarity (Higher = more similar)

```
SELECT   name,   1 - (embedding <=> %s::vector) AS cosine_similarity
FROM products ORDER BY cosine_similarity DESC LIMIT %s;
```

05

Scaling Challenges

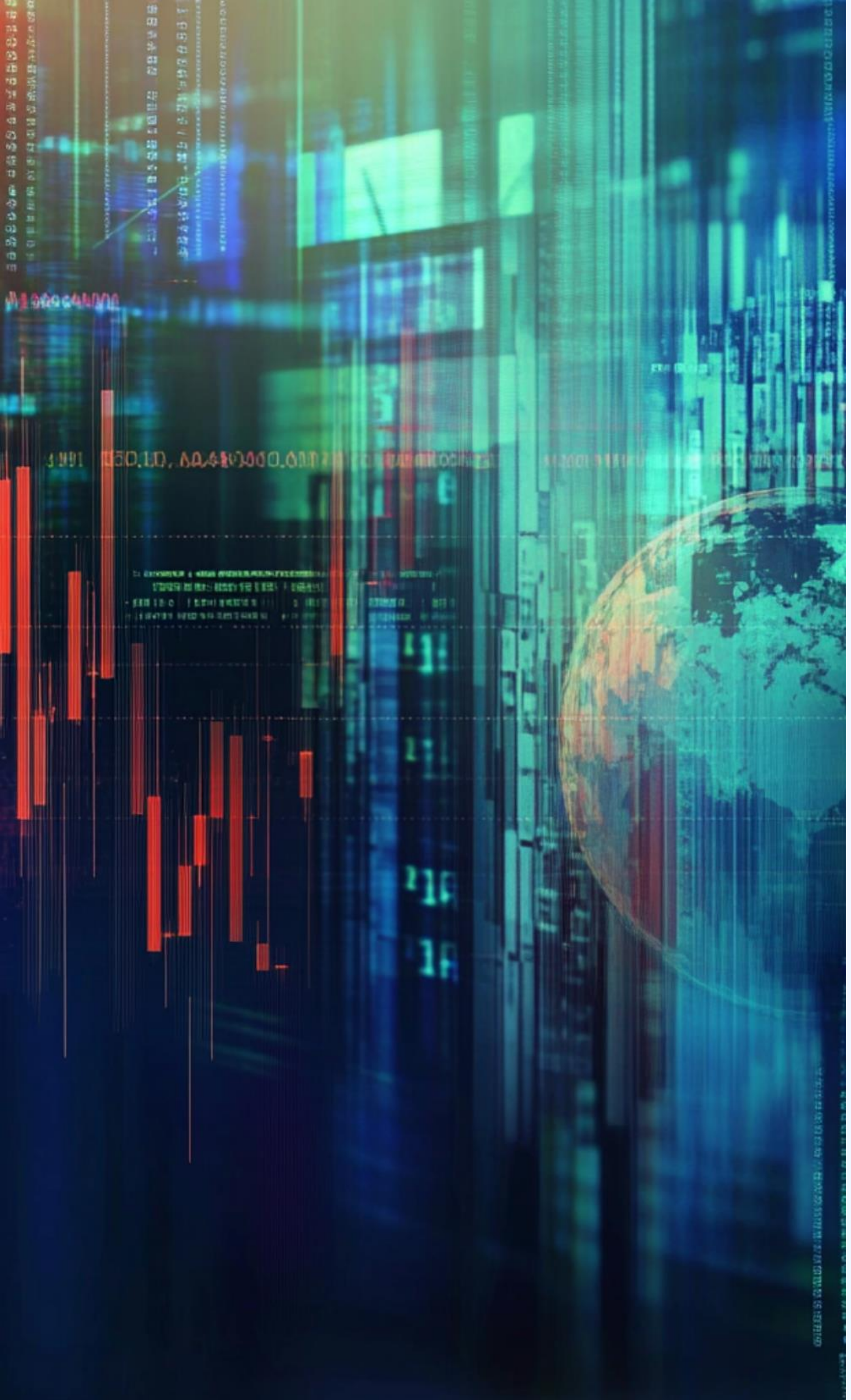
Scaling Challenges: Bottlenecks



High concurrency

Vector search latency

Slow Queries



Solution: Partitioning & Sharding

Range/Time-based partitioning.

Hash partitioning

Demo Use case:

Splitting AI log data by timestamp partitioning:

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE ai_logs_non_partitioned (
  id SERIAL,
  log_time TIMESTAMP NOT NULL,
  request_data JSONB,
  response_data JSONB,
  embedding VECTOR(512)
);

CREATE TABLE ai_logs_partitioned (
  id SERIAL,
  log_time TIMESTAMP NOT NULL,
  request_data JSONB,
  response_data JSONB,
  embedding VECTOR(512)
) PARTITION BY RANGE (log_time);
```

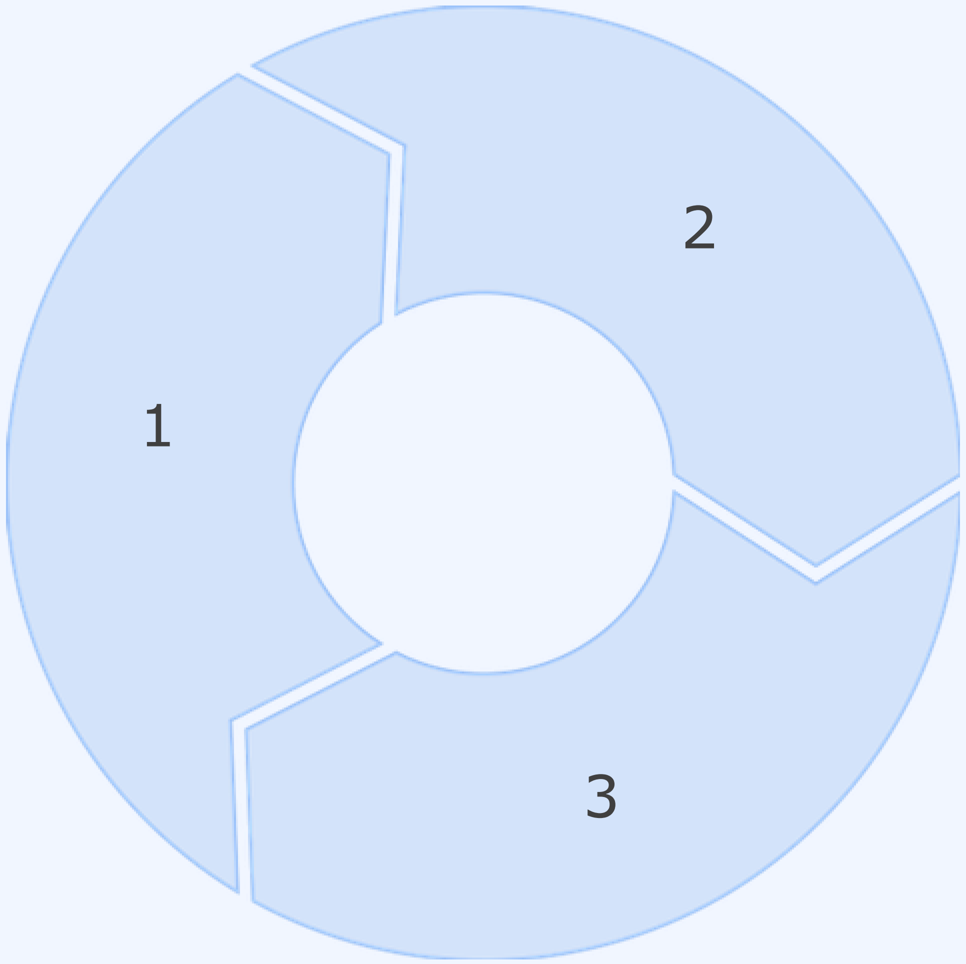
How It Works

1. **Parent Table (ai_logs)**:
   1. Acts as a logical container for all partitions.
   2. Does not store any data directly.
   3. Routes rows to the appropriate partition based on the log_time column.
2. **Partition Table (ai_logs_2024)**:
   1. Stores rows where log_time is between 2024-01-01 and 2025-01-01.
   2. Physically stores the data for this range.

06

Advantages of Partitioning

Advantages of Partitioning



Improved Query Performance: Queries only scan the relevant partitions, reducing the amount of data processed.

Easier Data Management: You can drop or archive old partitions without affecting the rest of the data.

Scalability: Partitioning helps manage large datasets by breaking them into smaller, more manageable pieces.

Illustration

Splitting AI log data by timestamp partitioning:

```
CREATE EXTENSION IF NOT EXISTS vector;

CREATE TABLE ai_logs_non_partitioned (
  id SERIAL,
  log_time TIMESTAMP NOT NULL,
  request_data JSONB,
  response_data JSONB,
  embedding VECTOR(512)
);

CREATE TABLE ai_logs_partitioned (
  id SERIAL,
  log_time TIMESTAMP NOT NULL,
  request_data JSONB,
  response_data JSONB,
  embedding VECTOR(512)
) PARTITION BY RANGE (log_time);
```

Without Partition:

```
-- Insert 1M rows
INSERT INTO ai_logs_non_partitioned (log_time, request_data, response_data, embedding)
SELECT
   NOW() - (random() * INTERVAL '365 days'),
   '{"input": "test"}'::jsonb,
   '{"output": "test"}'::jsonb,
   ARRAY(SELECT random() FROM generate_series(1, 512))::vector(512)
FROM generate_series(1, 1000000);
```

```
EXPLAIN ANALYZE
SELECT * FROM ai_logs_non_partitioned
WHERE log_time >= '2024-01-01' AND log_time < '2024-
02-01';
```

	QUERY PLAN text
1	Gather (cost=1000.00..19734.25 rows=4000 width=108) (actual time=94.326..98.688 rows=0 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on ai_logs_non_partitioned (cost=0.00..18334.25 rows=1667 width=108) (actual time=50.978..50.979 rows=0 loops=3)
5	Filter: ((log_time >= '2024-01-01 00:00:00'::timestamp without time zone) AND (log_time < '2024-02-01 00:00:00'::timestamp without time ...
6	Rows Removed by Filter: 333333
7	Planning Time: 0.826 ms
8	Execution Time: 98.707 ms

After Partition:

```sql
-- Insert 1M rows
INSERT INTO ai_logs_partitioned (log_time, request_data, response_data, embedding)
SELECT
   NOW() - (random() * INTERVAL '365 days'),
   '{"input": "test"}'::jsonb,
   '{"output": "test"}'::jsonb,
   ARRAY(SELECT random() FROM generate_series(1, 512))::vector(512)
FROM generate_series(1, 1000000);
```

```sql
EXPLAIN ANALYZE
SELECT * FROM ai_logs_partitioned
WHERE log_time >= '2024-01-01' AND log_time < '2024-02-01';
```

	QUERY PLAN text
1	Result (cost=0.00..0.00 rows=0 width=0) (actual time=0.001..0.001 rows=0 loops=...
2	One-Time Filter: false
3	Planning Time: 1.310 ms
4	Execution Time: 0.008 ms

When to Use Yearly Partitions

Small Datasets: If your dataset grows slowly (e.g., a few thousand rows per year), yearly partitions are sufficient.

Queries Spanning a Year: If your queries often target an entire year's data, yearly partitions are a good fit.

When to Use Monthly Partitions

Large Datasets: If your dataset grows quickly (e.g., millions of rows per year), monthly partitions are better.

Queries Targeting Specific Months: If your queries often target specific months, monthly partitions will improve performance.

07

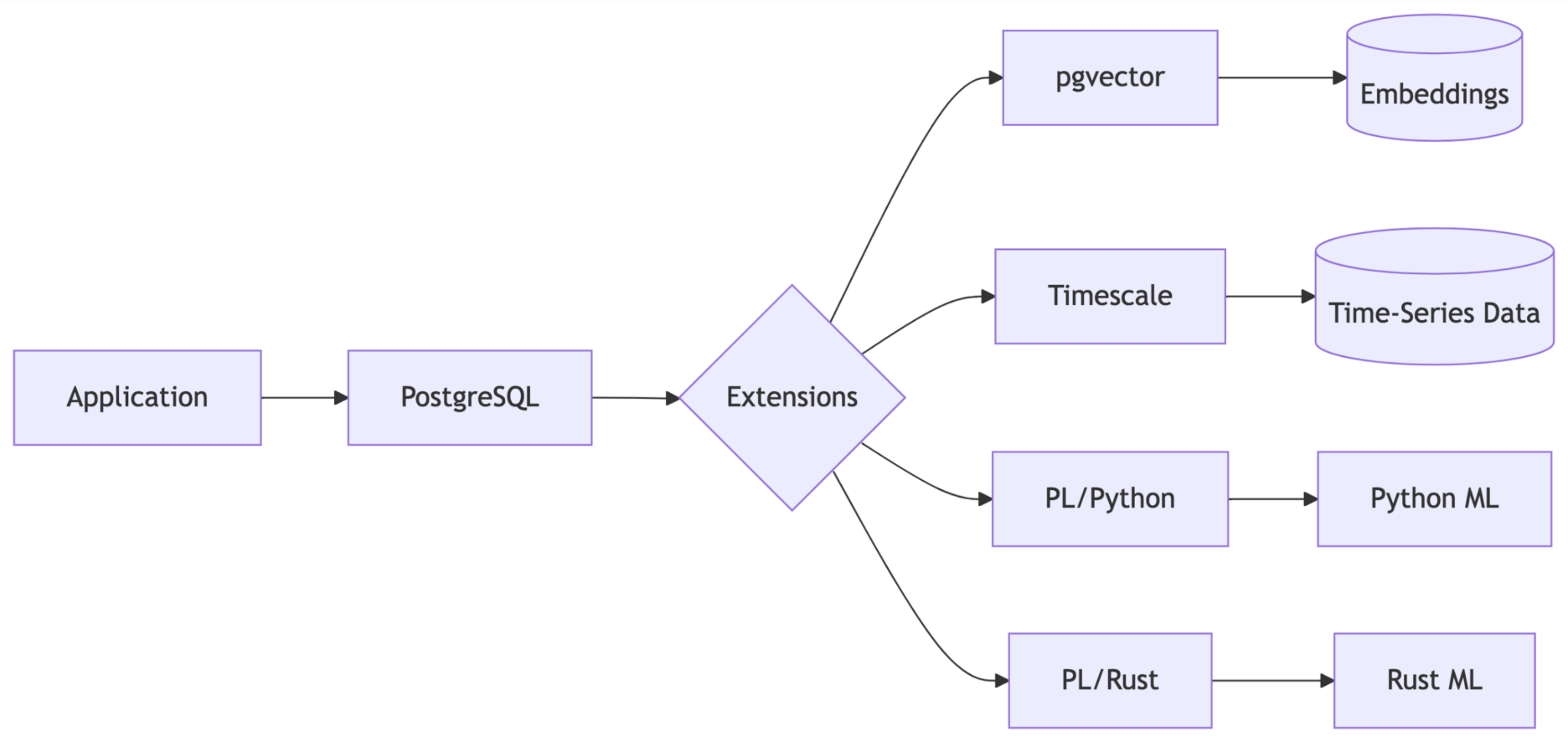
Compare with Vector database

Comparing Traditional & Vector Databases

Feature	*pgvector (PostgreSQL extension)*	*Dedicated Vector DB (e.g., Pinecone, Qdrant)*
Storage backend	PostgreSQL (relational + vector)	Purpose-built for vectors only
Similarity search	✅ L2, Cosine, Inner Product	✅ Advanced similarity search, customizable
ANN indexing options	✅ IVFFlat, HNSW (limited tuning)	✅ HNSW, PQ, IVF, ScaNN, etc. (more tuning options)
Query Filtering (metadata + vector)	✅ Built-in (SQL WHERE + vector search)	⚠️ Varies; Qdrant/Weaviate support it, but syntax differs
Joins, transactions, ACID	✅ Fully supported (relational DB features)	❌ No joins, limited transactional support
Scalability	✅ Good (scale-out via Citus, etc.)	✅ Excellent (designed for billions of vectors)
Ease of integration	✅ Simple if already using Postgres	⚠️ Requires separate service and sync
Use case fit	🟢 Great for AI inside existing apps	🟢 Great for pure vector-heavy workloads

Postgres as the AI Foundation

- Embedding search
- LLM metadata + feedback storage
- Event log, feature store
- With:
  - pgvector
  - timescale (time-series)
  - plpython / plrust for in-DB ML



Key Takeaways

✅ PostgreSQL is AI-ready
✅ pgvector simplifies similarity search
✅ Real-time, indexed search with one DB
✅ Easier, cheaper, scalable vs MySQL + external stack

08

Q & A

Thank You