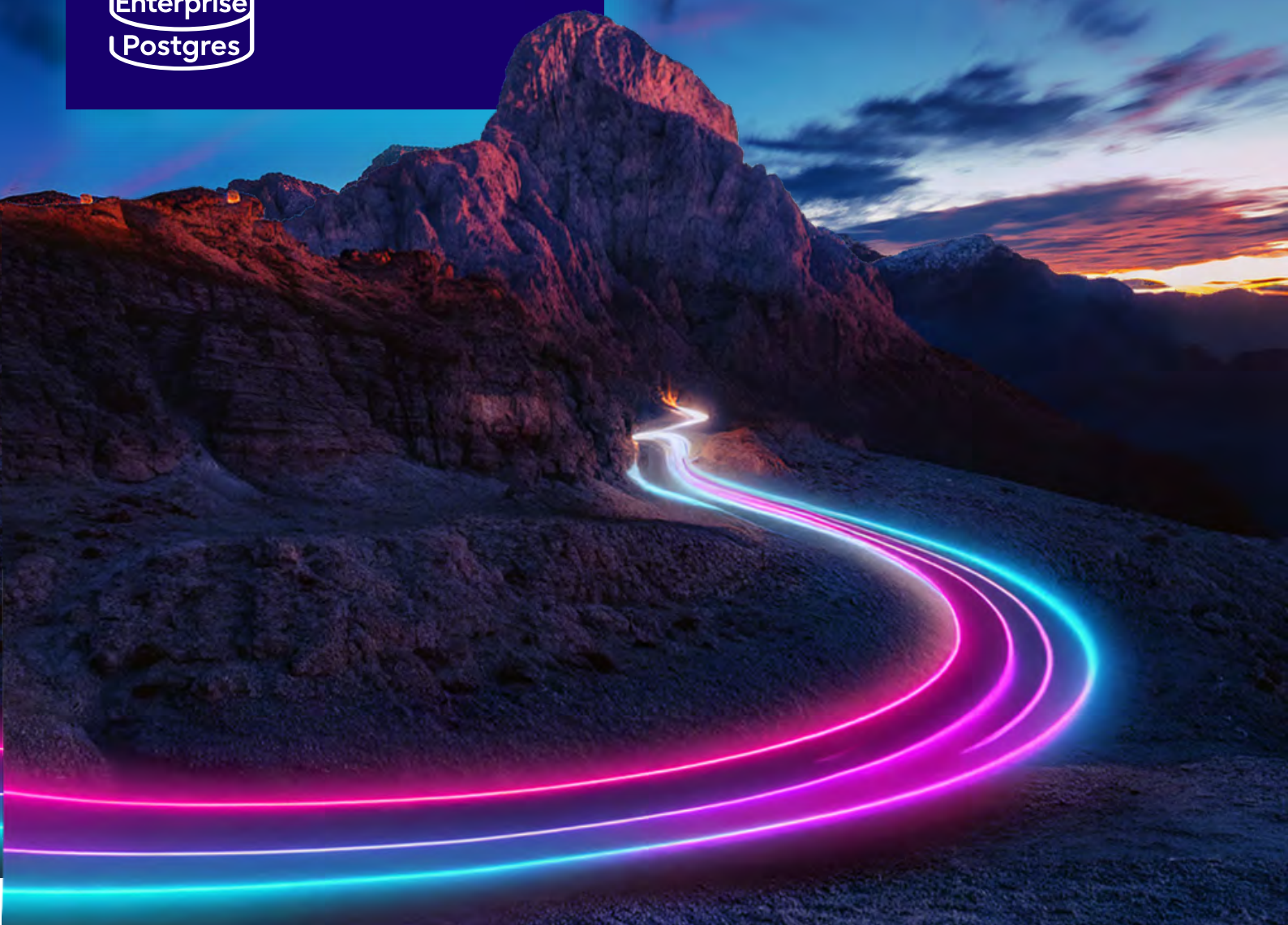
FUJITSU

Knowledge Data Management

Utilization with LangChain and Python

Fujitsu Enterprise Postgres



Introduction

In the Retrieval-Augmented Generation (RAG) approach, the database that stores external knowledge plays a crucial role.

Fujitsu Enterprise Postgres can be utilized as a knowledge base for RAG because it offers vector search through the pgvector extension and graph database feature via Apache AGE. In AI application development, Python frameworks are used, one of which is LangChain.

This document explains the following:

- How to handle vector data from Python applications using LangChain
- How to handle graph data from Python applications using LangChain
- How to protect objects created by LangChain
- How to handle vector data automatically generated by **Fujitsu Enterprise Postgres**

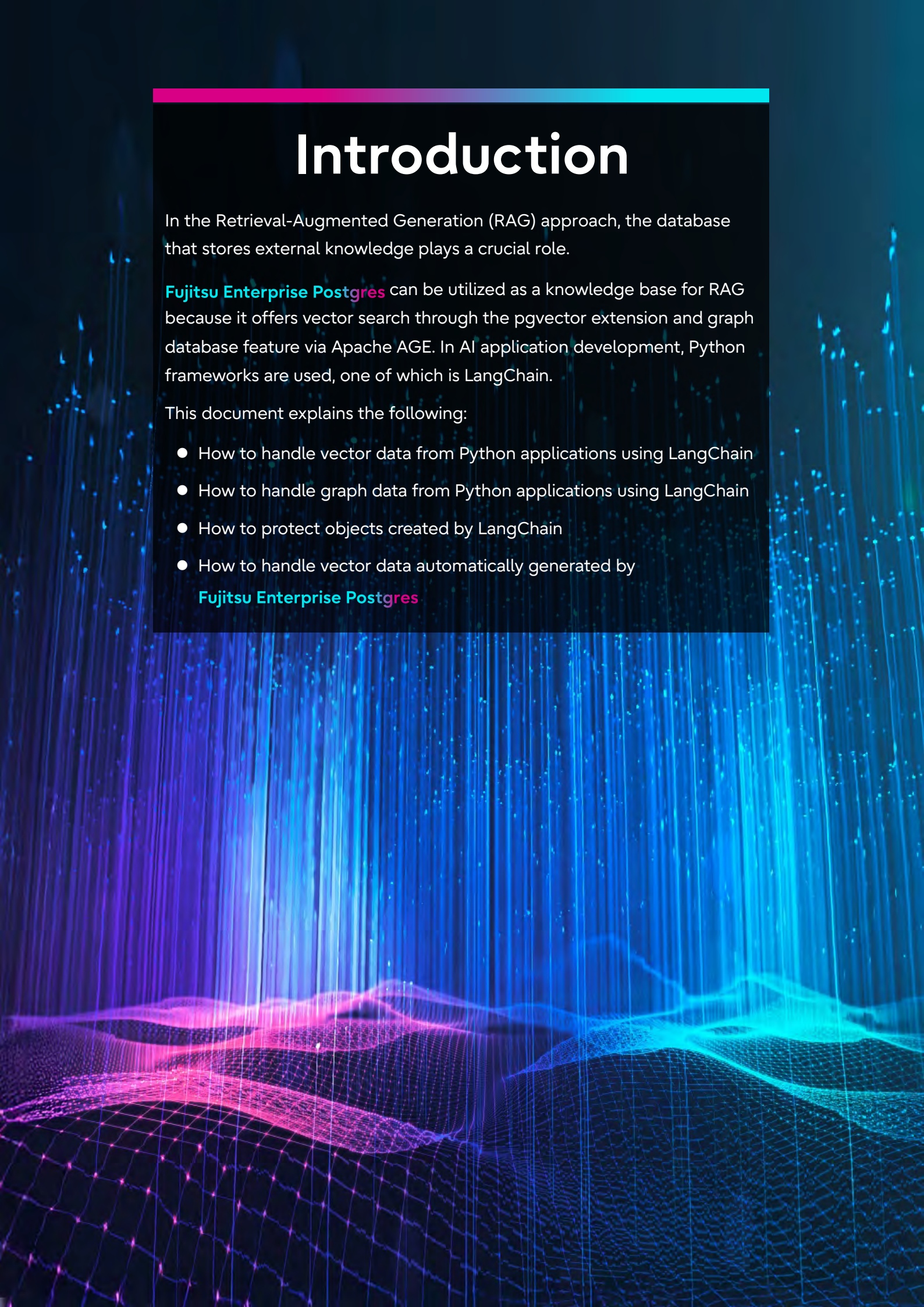


Table of contents

1. Setup

1.1. Database setup

You must set up PGVector to handle vector data, Apache AGE to handle graph data.

For setup instructions, refer to the **Installation and Setup Guide for Server**.

For setup instructions when using in combination with semantic text search and automatic vectorization, refer to the **Knowledge Data Management User's Guide**.

1.2. Application setup

To connect to **Fujitsu Enterprise Postgres** from a Python application, use psycopg3.

For setup instructions of psycopg3, refer to the **Application Development Guide**.

For details on installation of the necessary packages for connecting to PGVector and Apache AGE with **LangChain**, refer to the **LangChain documentation**.

Refer to the following links for PGVector and Apache AGE, respectively.

- LangChain PGVector: https://python.langchain.com/docs/integrations/vectorstores/pgvector/
- LangChain Apache AGE: https://python.langchain.com/docs/integrations/graphs/apache_age/

1.3. Setting up the embedding model

Set up the embedding model necessary to convert text data into vector data. In this document, we will explain using the embedding model provided by Ollama.

Embedding providers are providers that offer embedding models. For setup instructions, follow the manual provided by your chosen embedding provider.

2. How to handle vector data

2.1. How to connect to Fujitsu Enterprise Postgres using PGVector

PGVector is an interface provided by **LangChain** to handle the pgvector extension of PostgreSQL. Basically, it is the same as how to connect to OSS PostgreSQL from **LangChain**. When using the **Fujitsu Enterprise Postgres** client, port 27500 will be used by default, unlike OSS PostgreSQL.

```
from langchain_ollama import OllamaEmbeddings
from langchain_core.documents import Document
from langchain_postgres import PGVector
from langchain_postgres.vectorstores import PGVector

# Initialize the Ollama embedding model
embeddings = OllamaEmbeddings(model="llama3")

# Fujitsu Enterprise Postgres (FEP) connection information (change as necessary)
connection = "postgresql+psycopg://username:password@localhost:27500/rag_database"
collection_name = "my_docs"

# Create vector store using PGVector
vector_store = PGVector(
    embeddings=embeddings,
    collection_name=collection_name,
    connection=connection,
    use_jsonb=True,
)
```

When connecting to **Fujitsu Enterprise Postgres** using PGVector in **LangChain**

For setup instructions of psycopg3, refer to the **Application Development Guide**.

For details on how to store vectors and perform vector similarity searches on vector stores created using PGVector, refer to the **LangChain documentation**.

2.2. How to protect vector data created by LangChain

By creating a database in an encrypted tablespace and connecting to that database with **LangChain**, you can easily encrypt objects created by **LangChain**.

For details on how to encrypt, refer to the **Operation Guide > Protecting Stored Data Using Transparent Data Encryption**.

When setting access permissions using the Confidentiality Management, limit access permissions to the database connected to **LangChain** to the application developer and the person with application execution permissions.

If you want to set encryption or permissions individually, you can do so by setting them individually in the database object created by PGVector. When you use PGVector, it creates two tables in the public schema of the database connected to **LangChain**: `langchain_pg_collection` (list of vector stores in the database) and `langchain_pg_embedding` (stores all data in all vector stores). These tables can be encrypted using **Fujitsu Enterprise Postgres** Transparent Data Encryption. They can also be targeted by security features such as audit logs and Confidentiality Management.

3. How to handle graph data

3.1. How to connect to Fujitsu Enterprise Postgres using AgeGraph

AgeGraph is an interface provided by **LangChain** to handle the pgvector extension of PostgreSQL. Basically, it is the same as how to connect to OSS PostgreSQL from **LangChain**. When using the **Fujitsu Enterprise Postgres** client, port 27500 will be used by default, unlike OSS PostgreSQL.

LangChain's AgeGraph uses the psycopg2 adapter to connect to **Fujitsu Enterprise Postgres**. psycopg2 is required to store and search graph data using AgeGraph. Instead of psycopg2, the **Fujitsu Enterprise Postgres** client ships with psycopg3 which can't be used when connecting with AgeGraph. To use AgeGraph, you must install psycopg2.

```
from langchain_community.graphs.age_graph import AGEGraph
from langchain_neo4j import GraphCypherQAChain
from langchain_openai import ChatOpenAI

# Fujitsu Enterprise Postgres connection information (change as necessary)
conf = {
    "database": "rag_database",
    "user": "username",
    "password": "password",
    "host": "localhost",
    "port": 27500,
}

# Create objects in Fujitsu Enterprise Postgres using AgeGraph
graph = AGEGraph(graph_name="age_test", conf=conf)
```

Connecting to **Fujitsu Enterprise Postgres** using AgeGraph in **LangChain**

The psycopg2 adapter refers to libpq. Therefore, when executing a program that uses AgeGraph to connect to **Fujitsu Enterprise Postgres**, you must set the environment variable LD_LIBRARY_PATH so that it can refer to libpq included with the **Fujitsu Enterprise Postgres** client.

For details on using libpq, refer to the **Application Development Guide > C Library (libpq)**.

For details on how to store and search graph data, refer to the **LangChain documentation**.

3.2. How to protect graph data created by LangChain

Create a database in an encrypted tablespace and connect it with **LangChain** to encrypt objects created by **LangChain**.

For details on how to encrypt, refer to the **Operation Guide > Protecting Stored Data Using Transparent Data Encryption**.

When setting access permissions using Confidentiality Management, limit access to the database connected by **LangChain** to application developers and application execution authorities.

If you perform encryption and permission settings individually, apply settings individually to the database objects created by AgeGraph. When using AgeGraph, a schema with the same name as the specified graph_name is created on **Fujitsu Enterprise Postgres**. For example, you can verify objects created on the schema with the same name as the specified graph_name by executing SQL like the following.

```
SELECT
    c.relname AS object_name,
    CASE c.relkind
        WHEN 'r' THEN 'table'
        WHEN 'v' THEN 'view'
        WHEN 'm' THEN 'materialized view'
        WHEN 'S' THEN 'sequence'
        WHEN 'f' THEN 'foreign table'
        ELSE c.relkind
    END AS object_type
FROM pg_class c
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE n.nspname = 'age_test';
```

Searching for objects created by AgeGraph

These objects can be encrypted using the **Fujitsu Enterprise Postgres** Transparent Data Encryption. They can also be subject to security features such as audit logs and Confidentiality Management.

3.3. How to connect to Fujitsu Enterprise Postgres using psycopg3 and SQLAlchemy

By using psycopg3 and SQLAlchemy, you can store and search graph data without using AgeGraph in **LangChain**.

```
from sqlalchemy import create_engine, text

# Fujitsu Enterprise Postgres connection information (change as necessary)
db_url = "postgresql+psycopg://username@localhost:27500/rag_database"

# Create SQLAlchemy engine
engine = create_engine(db_url)

with engine.begin() as conn:
    # Set the search path to use functions within Age.
    conn.execute(text('SET search_path = ag_catalog, "$user", public;'))

    # Create a graph “social”
    conn.execute(text("SELECT create_graph('social');"))

    # Create a node: add Alice with the Person label.
    conn.execute(text("""
        SELECT * FROM cypher('social', $$
            CREATE (n:Person {name:'Alice'})
            RETURN n
        $$) AS (v agtype);
    """))

    # Create a node: add Bob with the Person label.
    conn.execute(text("""
        SELECT * FROM cypher('social', $$
            CREATE (n:Person {name:'Bob'})
            RETURN n
        $$) AS (v agtype);
    """))

    # Create an edge: add a KNOWS relationship from Alice to Bob
    conn.execute(text("""
        SELECT * FROM cypher('social', $$
            MATCH (a:Person {name:'Alice'}), (b:Person {name:'Bob'})
            CREATE (a)-[r:KNOWS]->(b)
            RETURN r
        $$) AS (v agtype);
    """))

    # Query Execution: Retrieve nodes with KNOWS relationship
    result = conn.execute(text("""
        SELECT * FROM cypher('social', $$
            MATCH (a:Person)-[r:KNOWS]->(b:Person)
            RETURN a, r, b
        $$) AS (a agtype, r agtype, b agtype);
    """))

    # Output results
    for row in result:
        print(row)
```

Connecting to **Fujitsu Enterprise Postgres** using psycopg3 and SQLAlchemy

4. Combination method with automatic vectorization

By using automatic vectorization provided by **Fujitsu Enterprise Postgres**, you can reduce the cost of managing vector data, as vector data is added, updated, and deleted in line with the addition, updating, and deletion of text data.

With automatic vectorization, when text data is added, a worker process automatically generates vector data in the background according to the predefined vectorization definition (vectorizer) and stores it in a table that contains vector data corresponding to the table that contains text data (embedding table). When you define vectorization for text data, a view that combines the vector data column with the table that contains the text data (embedding view) is also created. To perform semantic text search using the vector data created by automatic vectorization, use the **Fujitsu Enterprise Postgres** text semantic search, not the one provided by PGVector in **LangChain**.

The text semantic search allows you to perform semantic searches on embedding views. This is provided as a function called pgx_similarity_search. In this case, you can use either psycopg3 or SQLAlchemy to connect to **Fujitsu Enterprise Postgres** from a Python application that uses **LangChain**.

For details on the pgx_similarity_search function, refer to the Knowledge Data Management User Guide > Semantic text search and automatic vectorization.

```
from sqlalchemy import create_engine, text

# Fujitsu Enterprise Postgres connection information (change as necessary)
db_url = "postgresql+psycopg://postgres@localhost:27500/rag_database"

# Create SQLAlchemy engine
engine = create_engine(db_url)

# Connect and execute SQL
with engine.connect() as connection:
   # Example of executing the text meaning search function (obtaining 5 chunks similar to the
text specified as an argument from the embedded view "sample_embeddings")
   result = connection.execute(text("SELECT * FROM
pgx_vectorizer.pgx_similarity_search('sample_embeddings'::regclass, 'text for search', 5)"))

   # Display the results one line at a time
   for row in result:
      print(row)
```

Using psycopg3 and SQLAlchemy to perform **Fujitsu Enterprise Postgres** text semantic searches

System requirements

The features in this document are available from **Fujitsu Enterprise Postgres** 17 SP1. To use them with **Fujitsu Enterprise Postgres** 17, apply the emergency fix below according to OS:

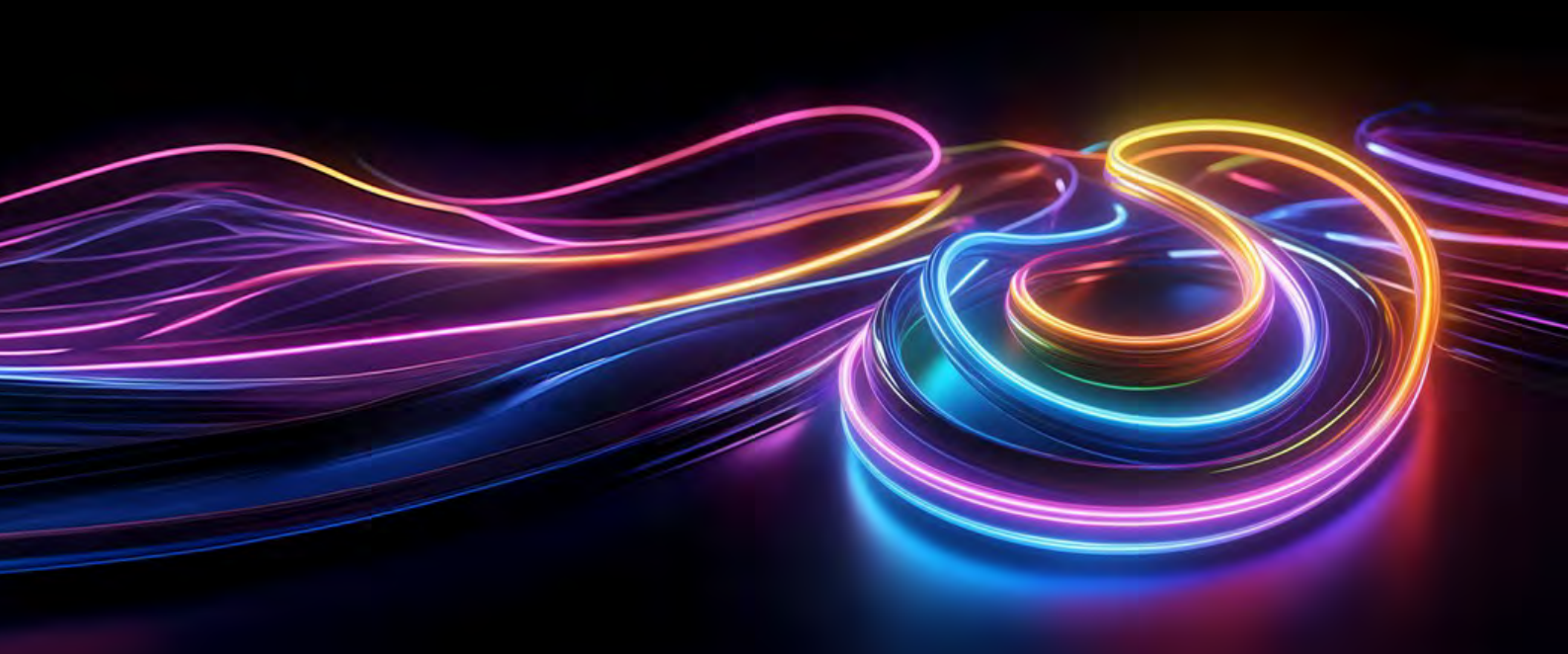
- Red Hat® Enterprise Linux® 8: T017857LP-01 or later
- Red Hat® Enterprise Linux® 9: T017858LP-01 or later
- SUSE Linux Enterprise Server 15: T017859LP-01 or later

The procedures in this document have been verified in the following environment:

- CPU architecture: x86_64
- OS: RHEL8.6
- **Fujitsu Enterprise Postgres**: 17 SP1 First Edition
- Python: v3.11.11
- LangChain: v0.3.20

FUJITSU

Fujitsu Enterprise Postgres is the enhanced version of PostgreSQL, for enterprises seeking a more robust, secure, and fully supported edition for business-critical applications



Contact
Fujitsu Limited
Email: enterprisepostgresql@fujitsu.com
Website: fast.fujitsu.com

2025-03-24 WW EN