

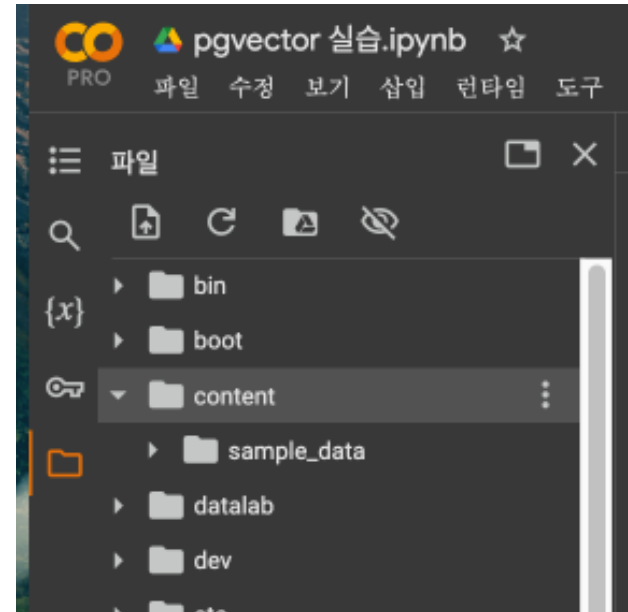
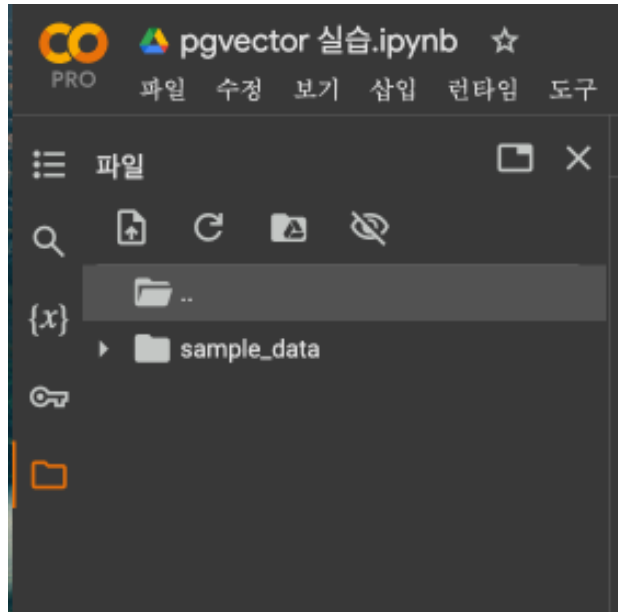
# PostgreSQL with PGVector

2025년 01월

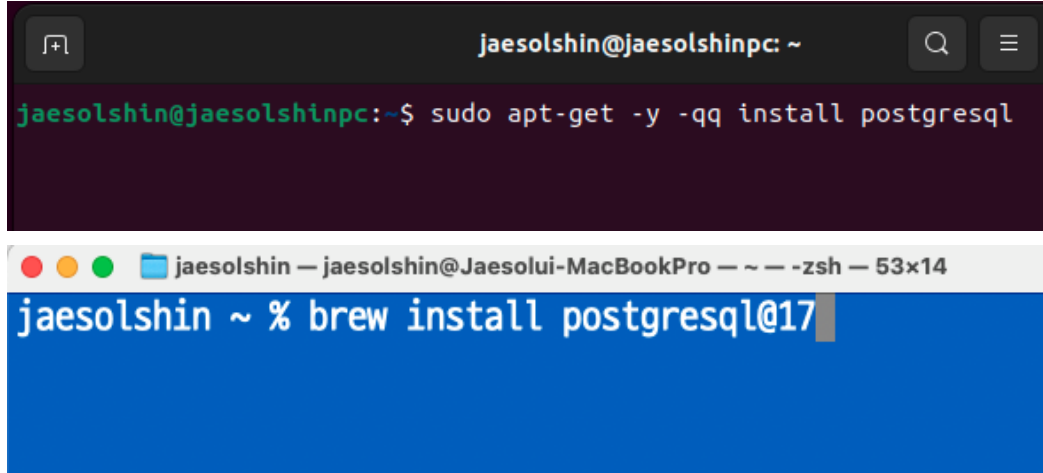
- PostgreSQL/pgvector 설치
- SQL을 통한 CRUD
- Python을 통한 CRUD
- Python 객체로 구현한 pgvector
- Text-to-SQL을 통한 하이브리드 서치 구현

# PostgreSQL/pgvector 설치

- Colab은 Google Cloud Platform(GCP) Compute Engine에서 제공하는 가상머신(VM) 위에서 실행되는 Jupyter Notebook 개발환경입니다. Colab 세션이 종료되면 해당 가상머신은 삭제되며, 가상환경에 설치된 프로그램과 저장된 데이터 역시 모두 초기화됩니다.
- 왼쪽 바의 폴더 모양 아이콘을 클릭하면 기본 작업 디렉토리가 /content로 설정되어 있음을 확인할 수 있습니다. 상위 폴더(..)를 클릭하면 /home 디렉토리로 이동해 전체 디렉토리 구조를 탐색할 수 있습니다.



- 일반적인 Linux, Windows, macOS 환경에서는 터미널에서 설치 작업을 진행하는 것을 권장합니다.



```
jaesolshin@jaesolshinpc: ~  
jaesolshin@jaesolshinpc:~$ sudo apt-get -y -qq install postgresql
```

```
jaesolshin ~ % brew install postgresql@17
```

← Linux Ubuntu와 MacOS의 터미널

자세한 환경별 설치방법은

PostgreSQL 공식 홈페이지 사이트 : [링크](#)

- 그러나 Colab에서는 직접적인 터미널을 제공하지 않으므로, 코드 셀에서 느낌표(!)를 사용하여 시스템의 셀 명령어를 실행할 수 있습니다. 이 방식은 터미널 명령어 실행과 동일합니다.



```
[1] ! pwd  
/content
```

- 시스템 명령어를 이용해 OS 정보를 확인해보면 설치된 환경은 Ubuntu 22.04.3 임을 알 수 있습니다.
- 우분투에서 패키지를 설치할 때 사용하는 패키지 관리자 apt를 이용해서 PostgreSQL을 설치합니다.

```
# OS 정보확인
# Ubuntu 22.04
!cat /etc/os-release

PRETTY_NAME="Ubuntu 22.04.3 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.3 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=jammy
```

- 시스템 명령어를 이용해 필수 도구를 설치한 뒤, PostgreSQL 데이터베이스를 설치합니다. -y와 -qq 옵션을 통해 사용자 입력 없이 조용히 설치가 진행됩니다.

## # 필수 도구 설치

```
! sudo apt-get install build-essential curl file git
```

## # PostgreSQL 설치

```
! sudo apt-get -y -qq update
```

```
! sudo apt-get -y -qq install postgresql
```

- 설치가 완료된 후 PostgreSQL 서버 서비스를 시작합니다. 이제 psql이나 파이썬을 통해 PostgreSQL 서버에 접속하여 데이터베이스를 생성하고, 데이터를 입력하거나 조회하는 등의 작업을 수행할 수 있습니다.

## # postgres 서버 서비스 시작

```
! sudo service postgresql start
```

- 이제 설치된 PostgreSQL에 데이터베이스를 생성해봅시다.
- 운영체제 사용자 'postgres'로 PostgreSQL에 접속 후 -c 옵션을 통해 단일 명령을 시행합니다. USER 'postgres'에 대해 PASSWORD 'postgres'를 설정하고, DATABASE dev를 생성합니다.

```
!sudo -u postgres psql -U postgres -c "ALTER USER postgres PASSWORD 'postgres';"  
!sudo -u postgres psql -U postgres -c "DROP DATABASE IF EXISTS dev;"  
!sudo -u postgres psql -U postgres -c "CREATE DATABASE dev;"
```

- 데이터베이스 연결을 위한 URL을 환경변수에 저장

```
%env DATABASE_URL=postgresql://postgres:postgres@localhost:5432/dev
```



- 다음과 같은 확장을 설치하면 Colab에서 SQL 명령어를 이용할 수 있게 됩니다.
- 환경변수에 저장된 경로로 데이터베이스에 접속하여 쿼리를 수행한 결과를 반환합니다.

```
# ipython-sql, prettytable: SQL 확장 모듈
# %load_ext sql 로 SQL 확장 모듈 로드하기
!pip install -q ipython-sql==0.5.0 prettytable==2.1.0
%load_ext sql
%config SqlMagic.style = 'DEFAULT'
```

- 이제 매직명령어 %%sql를 통해 SQL 쿼리를 실행할 수 있습니다. 시험삼아 버전을 확인해봅시다.

```
%%sql
select version();
```

version

PostgreSQL 14.15 (Ubuntu 14.15-0ubuntu0.22.04.1) on x86\_64-pc-linux-gnu, compiled by gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, 64-bit

- pgvector는 PostgreSQL에서 vector 자료형과 각종 거리계산 함수 및 연산자들을 사용할 수 있게 해주는 확장입니다. 깃허브의 pgvector 리포지토리에는 이런 계산을 가능하게 해주는 소스코드와 정의파일이 담겨있습니다.
- git clone 명령을 통해 이 리포지토리를 다운받고 해당 위치로 이동합니다

```
!git clone --branch v0.8.0 https://github.com/pgvector/pgvector.git  
%cd /content/pgvector
```

- 이 소스코드를 빌드하기 위한 툴을 설치합니다.

```
!sudo apt install -qq postgresql-server-dev-all
```

- 이제 소스코드를 빌드하고, PostgreSQL 확장을 설치합니다.

```
!make  
!make install
```

- 이제 pgvector 확장이 설치되었고, 각 데이터베이스에서 설치된 확장을 실행(CREATE EXTENSION)할 수 있습니다. 확장은 데이터베이스별로 최초 1회만 실행하면 됩니다.

```
%%sql  
CREATE EXTENSION IF NOT EXISTS vector;  
COMMIT;
```

- 다음 명령어를 통해 설치된 확장 목록을 확인할 수 있습니다.

```
%%sql  
SELECT extname FROM pg_extension
```

- CREATE EXTENSION은 데이터베이스의 상태를 변경하는 작업이므로, 트랜잭션 내에서 실행됩니다. 트랜잭션 내 작업은 COMMIT을 통해 확정해야 반영됩니다. 자동으로 반영되도록 오토커밋을 활성화합니다.

```
!sudo -u postgres psql -c 'set AUTOCOMMIT on'
```

# SQL을 통한 CRUD

- pgvector에서 데이터를 처리하는 방식을 이해하기 위해 먼저 SQL을 통해 직접 데이터를 입력하고 조회해보겠습니다. Colab에서는 %%sql 매직명령어를 통해 PostgreSQL에 SQL 쿼리를 보내고 결과를 확인할 수 있습니다. 일반환경에서는 터미널에서 psql을 실행하여 보낼 수 있습니다.
- 먼저 테이블을 만들어보겠습니다. id, text, embedding 세개의 컬럼을 가지는 embeddings 테이블을 생성합니다. embedding 컬럼의 자료형을 VECTOR(3), 즉 3차원의 벡터로 두겠습니다. 이 자료형을 사용하려면 pgvector 확장이 설치되고 활성화되어 있어야 합니다.

```
%%sql
CREATE TABLE embeddings (
  id SERIAL PRIMARY KEY,
  text TEXT NOT NULL,
  embedding VECTOR(3) -- 벡터 크기 3
);
```

- text, embedding에 10개의 값을 INSERT 해보겠습니다. id는 SERIAL 타입으로 정의되어 자동부여됩니다.

```
%%sql
INSERT INTO embeddings
(text, embedding)
VALUES
('하늘', '[-0.8, 0.2, -0.3]'),
('지우', '[0.4, -0.5, 0.6]'),
('기범', '[-0.7, 0.6, -0.5]'),
('규진', '[0.9, -0.7, -0.1]'),
('승헌', '[0.2, -0.1, 0.9]'),
('혜원', '[-0.3, 0.8, -0.9]'),
('유연', '[0.5, -0.4, 0.3]'),
('채원', '[0.1, -0.2, 0.7]'),
('소현', '[-0.6, 0.5, -0.8]'),
('태민', '[0.3, -0.8, 0.1]);
```

- embeddings 테이블의 모든 값을 조회해봅시다. 아까 입력한 값들이 잘 들어가 있습니다.

```
SELECT * from embeddings;
```

```
* postgresql://postgres:***@localhost:5432/dev
10 rows affected.
10 rows affected.
 id text  embedding
 1  하늘 [-0.8,0.2,-0.3]
 2  지우 [0.4,-0.5,0.6]
 3  기범 [-0.7,0.6,-0.5]
 4  규진 [0.9,-0.7,-0.1]
 5  승헌 [0.2,-0.1,0.9]
 6  혜원 [-0.3,0.8,-0.9]
 7  유연 [0.5,-0.4,0.3]
 8  채원 [0.1,-0.2,0.7]
 9  소현 [-0.6,0.5,-0.8]
10 태민 [0.3,-0.8,0.1]
```

# pgvector : SQL을 통한 데이터 입력



- 이번엔 같은 결과를 ORDER BY 절을 통해 정렬해봅시다.
- 임의의 벡터 '[0.3, 0.1, 0.2]'와의 거리가 가까운 행이 먼저 오도록 정렬됩니다.
- 벡터 검색 시에 이 자리에 검색하고자 하는 쿼리 벡터가 오게 됩니다.
- LIMIT 절을 통해 5개 결과만 표시하도록 합니다.

```
%%sql
SELECT
*
FROM
embeddings
ORDER BY
embedding <-> '[0.3,0.1,0.2]'
LIMIT 5;
```

id	text	embedding
7	유연	[0.5,-0.4,0.3]
8	채원	[0.1,-0.2,0.7]
2	지우	[0.4,-0.5,0.6]
5	승헌	[0.2,-0.1,0.9]
10	태민	[0.3,-0.8,0.1]

- 아까와 달리 유연, 채원이 상위권에 위치하고 있습니다.



- 이번엔 결과에 계산된 거리도 함께 표시되도록 해봅시다.
- SELECT 절에 계산된 거리를 추가하고, 소수점 네자리까지 절삭하여 l2라는 이름으로 표시합니다.

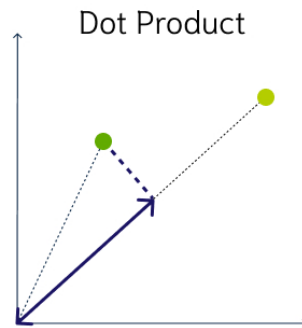
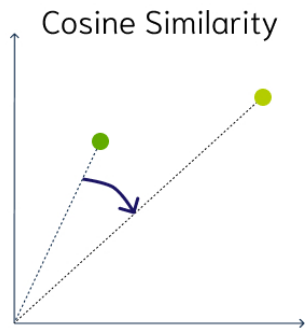
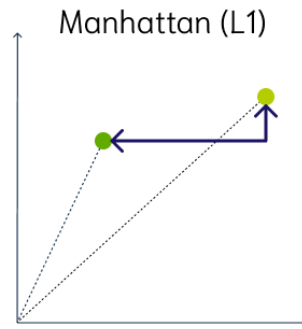
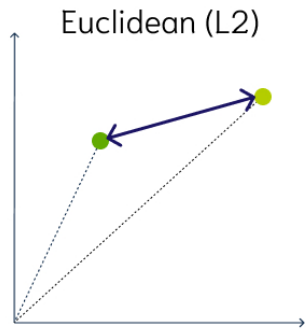
```
%%sql
SELECT
*,
TRUNC((embedding <-> '[0.3,0.1,0.2]')::numeric,4) as l2
FROM
embeddings
ORDER BY
embedding <-> '[0.3,0.1,0.2]'
LIMIT 5;
```

id	text	embedding	l2
7	유연	[0.5,-0.4,0.3]	0.5477
8	채원	[0.1,-0.2,0.7]	0.6164
2	지우	[0.4,-0.5,0.6]	0.7280
5	승헌	[0.2,-0.1,0.9]	0.7348
10	태민	[0.3,-0.8,0.1]	0.9055

# pgvector : SQL을 통한 데이터 입력



- 방금 임베딩 벡터와의 거리를 계산할 때 embedding <-> '[0.3,0.1,0.2]'라는 표현을 썼습니다. <->는 l2 거리(유클리드 거리)를 계산하는 연산자입니다. pgvector는 다른 거리계산 방법도 지원합니다.



<-> L2거리(유클리드거리) L2 Distance

<#> 음의 내적 Negative Inner Product

<=> 코사인거리 Cosine Distance

<+> L1거리(택시거리,맨하탄거리) L1 Distance

# pgvector : SQL을 통한 데이터 입력



```
%%sql
SELECT
*,
TRUNC((embedding <-> '[0.3,0.1,0.2]')::numeric,4) as L2
FROM
embeddings
ORDER BY
embedding <-> '[0.3,0.1,0.2]'
LIMIT 5;
```

<-> L2거리(유클리드거리)

id	text	embedding	l2
7	유연	[0.5,-0.4,0.3]	0.5477
8	채원	[0.1,-0.2,0.7]	0.6164
2	지우	[0.4,-0.5,0.6]	0.7280
5	승헌	[0.2,-0.1,0.9]	0.7348
10	태민	[0.3,-0.8,0.1]	0.9055

```
%%sql
SELECT
*,
TRUNC((embedding <=> '[0.3,0.1,0.2]')::numeric,4) as cosine_distance
FROM
embeddings
ORDER BY
embedding <=> '[0.3,0.1,0.2]'
LIMIT 5;
```

<=> 코사인거리 Cosine Distance

id	text	embedding	cosine_distance
5	승헌	[0.2,-0.1,0.9]	0.3371
7	유연	[0.5,-0.4,0.3]	0.3574
2	지우	[0.4,-0.5,0.6]	0.4213
8	채원	[0.1,-0.2,0.7]	0.4544
4	규진	[0.9,-0.7,-0.1]	0.5796

# [참고] ChromaDB와 PGVector의 작동방식 비교



- ChromaDB는 내부적으로 Python의 numpy 배열을 직렬화하여 BLOB 타입으로 SQLite에 저장합니다. 이후 Rust를 활용해 거리 함수 및 HNSW 인덱싱을 처리하며, SQLite 데이터는 사용자가 직접 접근할 수 없도록 관리됩니다.
- 반면 PostgreSQL+pgvector는 C의 float 배열을 기반으로 사용자 정의 데이터 타입인 VECTOR로 저장하며, C로 구현된 거리 함수와 인덱싱을 제공합니다. PostgreSQL은 정의된 테이블에 직접 접근할 수 있으며, Python을 통해 조작하는 것도 가능합니다.
- 앞서 3차원 벡터의 예시를 보여드렸습니다. 수백, 수천차원의 벡터를 직접 손으로 입력하는 것은 불가능하기 때문에 실제로는 주로 Python을 통해 데이터를 다루게 됩니다. Python에서는 SQL 쿼리를 직접 작성하여 psycopg2와 같은 라이브러리를 사용하거나, sqlalchemy를 통해 쿼리 작성을 자동화할 수 있습니다.
- 데이터 입출력 및 관리는 클래스를 통해 추상화하는 것이 효율적입니다. langchain에서 제공하는 `langchain\_postgres` 라이브러리는 이를 더욱 간편하게 처리할 수 있도록 도와줍니다. Pgvector가 작동하는 방식을 이해하기 위해서는 직접 구현하는 편이 낫습니다. 본 실습에서는 Python에서 직접 SQL 쿼리를 작성하여 psycopg2를 통해 요청을 보내고, 이를 클래스를 통해 추상화하는 방법을 구현해보겠습니다.

# Python을 통한 CRUD

# psycopg2 : 파이썬을 통한 PostgreSQL 데이터베이스 조작



```
import psycopg2

# 데이터베이스 연결
conn = psycopg2.connect(
    dbname='dev',
    user='postgres',
    password='postgres',
    host='localhost',
    port=5432

# 커서 생성
cursor = conn.cursor()

# 쿼리 실행 (query와 values는 미리 정의되어 있어야 함)
query = "INSERT INTO your_table_name (column1, column2) VALUES (%s, %s)"
values = ('value1', 'value2')
cursor.execute(query, values)

# 변경사항 커밋
conn.commit()

# 커서와 연결 종료
cursor.close()
conn.close()
```

**psycopg2** : Python에서 PostgreSQL 데이터베이스를 다루기 위한 도구

**conn** : psycopg2.connect()를 사용하여 데이터베이스 연결 객체(conn)를 생성합니다. 이 객체는 데이터베이스와의 연결을 유지하며 작업을 수행할 기반을 제공합니다.

**cursor** : 데이터베이스와 상호작용하기 위해 conn.cursor()를 호출하여 커서 객체(cursor)를 생성하며, 이는 SQL 쿼리 실행 및 결과 처리를 담당합니다.

**cursor.execute()** : SQL 쿼리 실행

작업이 완료되면 커서를 닫고, 연결객체를 종료하여 리소스를 해제해야 합니다.

```

import psycopg2
from typing import Dict, Tuple

# PostgreSQL 연결 정보
DB_CONFIG = {
    'dbname': 'dev',
    'user': 'postgres',
    'password': 'postgres',
    'host': 'localhost',
    'port': 5432
}

def execute_query(query:str, values: Tuple=None):
    try:
        # PostgreSQL 연결
        conn = psycopg2.connect(**DB_CONFIG)
        cursor = conn.cursor()

        # 쿼리 실행
        cursor.execute(query, values)

        # SELECT 쿼리인 경우 결과 반환
        if query.strip().lower().startswith("select"):
            result = cursor.fetchall()
            return result
        else:
            # SELECT가 아닌 쿼리는 커밋 후 None 반환
            conn.commit()
            return None

    except Exception as e:
        print(f"Error: {e}")
    finally:
        # 연결 종료
        if cursor:
            cursor.close()
        if conn:
            conn.close()

```

- 이렇게 함수를 작성하면 DB\_CONFIG에 담긴 연결정보로 DB에 접속한 뒤, query를 수행하고 결과값을 반환하거나 커밋하는 작업을 자동화할 수 있습니다.
- 데이터베이스에서 원하는 값을 조회하는 쿼리의 경우 이를 반환하여 가져오고, 데이터베이스에 변경이 발생하는 경우 commit을 통해 이를 반영합니다.
- 주어진 쿼리를 실행하고 에러가 발생할 시 에러구문을 출력하고, 그렇지 않은 경우 연결을 해제하여 리소스를 정리합니다.
- typing 라이브러리를 통해 query에 입력되는 값은 str이고, 바인딩되는 값이 존재하는 경우 values에 입력되는 값은 Tuple이어야 함을 명시했습니다.

```
execute_query("SELECT id, text FROM embeddings;")
```

가령 위와 같이 embeddings 테이블에서 모든 행의 id와 text를 가져오도록 하는 쿼리를 입력하면, 아까 SQL 쿼리를 통해 직접 입력 해두었던 정보들을 확인할 수 있습니다.

```
[28] def select_all():
    query = "SELECT id, text FROM embeddings;"
    return execute_query(query)
```

```
select_all()
```

```
→ [(1, '하늘'),
    (2, '지우'),
    (3, '기범'),
    (4, '규진'),
    (5, '승현'),
    (6, '혜원'),
    (7, '유연'),
    (8, '채원'),
    (9, '소현'),
    (10, '태민')]
```

```
[29] def similarity_search(vector, k=1):
    select_query = f"""
    SELECT
        id,
        text,
        embedding <=> %s::vector AS cosine_distance
    FROM embeddings
    ORDER BY cosine_distance
    LIMIT %s;
    """
    return execute_query(select_query, (vector, k))
```

```
query_vector = [0.3, 0.1, 0.2]
```

```
similarity_search(query_vector, 3)
```

```
→ [(5, '승현', 0.33715099156930783),
    (7, '유연', 0.357460334172485),
    (2, '지우', 0.42131231965954385)]
```

select\_all()을 호출하면 "SELECT id, text FROM embeddings; " 이 실행됩니다.

similarity\_search(vector, k)를 호출하면 select\_query가 실행됩니다.

```
"""
SELECT
id,
text,
embedding <=> %s::vector AS cosine_distance
FROM embeddings
ORDER BY cosine_distance
LIMIT %s;
"""
```

execute\_query(select\_query, (vector, k))가 호출되는 과정에서

select\_query %s, %s 자리에 vector와 k가 바인딩됩니다.

query\_vector에 [0.3, 0.1, 0.2]을 넣어놓고,

similarity\_search(query\_vector, 3)을 호출하면,

상위 3개 검색결과를 확인할 수 있습니다.



```
[30] def insert_embedding(text, vector):
    insert_query = f"""
    INSERT INTO embeddings (text, embedding)
    VALUES (%s, %s::vector);
    """
    return execute_query(insert_query, (text, vector))

insert_embedding('위데이터랩', query_vector)
```

```
[31] def get_id_by_text(text):
    select_query = f"""
    SELECT id FROM embeddings WHERE text = %s;
    """
    return execute_query(select_query, (text,))[0][0]

def delete_embedding(id):
    delete_query = f"""
    DELETE FROM embeddings
    WHERE id = %s;
    """
    return execute_query(delete_query, (id,))

id = get_id_by_text('위데이터랩')
print("id: ", id, "\n")

delete_embedding(id)
select_all()
```

```
➡ id: 11
```

```
[(1, '하늘'),
 (2, '지우'),
 (3, '기범'),
 (4, '규진'),
 (5, '승현'),
 (6, '혜원'),
 (7, '유연'),
 (8, '채원')]
```

insert\_embedding(text, vector)을 호출하면 text와 vector가 들어간다.

(위데이터랩, query\_vector)를 삽입한 뒤,

get\_id\_by\_text(text)로 해당 id를 찾아내면 id: 11이 출력된다.

delete\_embedding(id)로 해당 행을 삭제한뒤,

select\_all()을 통해 전체 데이터를 확인하면 삭제된 것을 확인할 수 있다.

```
[32] def insert_embedding_list(list_text, list_vector):
      """
      input: tuple of list (list(text), list(vector))
      """
      for text, vector in zip(list_text, list_vector):
          insert_embedding(text, vector)

names = ['위데이터랩1',
         '위데이터랩2',
         '위데이터랩3']

embeddings = [[0.1,0.1,0.2],
              [0.8,0.9,0.7],
              [0.1,0.8,0.7]]

insert_embedding_list(names, embeddings)
select_all()
```

```
⇒ [(1, '하늘'),
   (2, '지우'),
   (3, '기범'),
   (4, '규진'),
   (5, '승헌'),
   (6, '혜원'),
   (7, '유연'),
   (8, '채원'),
   (9, '소현'),
   (10, '태민'),
   (12, '위데이터랩1'),
   (13, '위데이터랩2'),
   (14, '위데이터랩3')]
```

이번엔 text와 vector를 리스트로 받아서,

zip()을 통해 하나씩 짝지워 insert\_embedding() 함수를 호출해보자.

names와 embeddings에 텍스트와 벡터 리스트를 넣어놓고,

insert\_embedding\_list(names, embeddings)를 호출해보자.

한번에 3개의 데이터가 삽입된 것을 확인할 수 있다.

- 지금까지는 실제 텍스트를 의미있게 벡터화한 임베딩이 아닌 임의의 숫자를 입력해보았다. 이번엔 OpenAI의 임베딩 모델을 활용해 변환한 임베딩 벡터를 넣어보자. OpenAI의 최신 임베딩 모델은 text-embedding-3이며, small 버전과 large 버전이 있다. 두 모델 차원 기본값은 1536차원이지만, large 모델은 훨씬 큰 아키텍처를 통해 변환하기 때문에 약간 더 정확하고, 가격은 6배 이상 비싸다.

MODEL	~ PAGES PER DOLLAR	PERFORMANCE ON MTEB EVAL	MAX INPUT
text-embedding-3-small	62,500	62.3%	8191
text-embedding-3-large	9,615	64.6%	8191
text-embedding-ada-002	12,500	61.0%	8191

# OpenAI Embedding 활용



- 로그인을 위해 OPENAI API KEY를 발급 받는다 (실습시에는 API KEY 제공) : <https://platform.openai.com/api-key>  
발급받은 키는 가급적 노출하지 않고 환경변수로 저장하는 것이 좋다. Colab의 경우, 보안 비밀 탭에 저장할 수 있다(도구 - 명령 팔레트 - 사용자 보안 비밀 탭 열기 또는 왼쪽 바에서 열쇠 모양 클릭)

The screenshot shows the Google Colab interface for a notebook named 'pgvector 실습.ipynb'. The left sidebar is open to the 'Security' (보안 비밀) tab, which displays a table of environment variables. The 'OPENAI\_API\_KEY' variable is highlighted with a blue checkmark, indicating it is set. The main code area shows two cells: cell [33] runs '!pip install -q openai' and cell [34] contains comments in Korean and Python code to set the 'OPENAI\_API\_KEY' environment variable using 'userdata.get()'.

노트북 액세스	이름	값	작업
<input type="checkbox"/>	HF_TOKEN	.....	
<input checked="" type="checkbox"/>	OPENAI_API_KEY	.....	
<input type="checkbox"/>	SERPER_API_KEY	.....	
<input type="checkbox"/>	gemini_key	.....	

```
[33] !pip install -q openai

[34] # 사전에 OPENAI API KEY 발급필요
      # https://platform.openai.com/api-keys
      # 허깅페이스나 Gemini는 무료로 이용가능

      # 발급받은 키는 가급적 노출하지 않고 환경변수로 저장
      # Colab의 경우, 발급받은 키는 보안 비밀 탭에 저장해두면 좋음
      # 도구 - 명령 팔레트 - 사용자 보안 비밀 탭 열기
      # 또는 왼쪽 바에서 열쇠 모양 클릭

      import os
      from google.colab import userdata
      os.environ['OPENAI_API_KEY'] = userdata.get('OPENAI_API_KEY')
```

# OpenAI Embedding 활용



- openai 라이브러리를 설치하고 임포트한다. openai.api\_key 또는 openai.embeddings.create()에 직접 api\_key를 전달할 수도 있다.

```
!pip install -q openai
```

```
import openai

# openai.api_key = "your-api-key"
text = 'hello'
model = 'text-embedding-3-small'
response = openai.embeddings.create(input=text, model=model)
embedding = response.data[0].embedding
```

- response는 json 타입으로, response.data[i].embedding에 텍스트 input[i]의 임베딩 값이 float list로 반환된다.
- openai 라이브러리 버전마다 호출방법이나 반환형식이 다르므로, 버전에 맞는 방법을 숙지해두어야 한다.
- <https://platform.openai.com/docs/guides/embeddings>

```
▶ embedding = response.data[0].embedding
  print("type: ", type(embedding))
  print("length: ", len(embedding))
```

```
↔ type: <class 'list'>
   length: 1536
```

# OpenAI Embedding 활용



- 아래와 같이 get\_embedding을 정의하면 str list에 대해 float list(임베딩 벡터)의 list를 받을 수 있다.

```
def get_embedding(texts:list):  
    response = openai.embeddings.create(input=texts, model=model)  
    return [item.embedding for item in response.data]
```

- sentences에 예시 텍스트들을 넣어놓고 get\_embeddings()에 넣어 임베딩 벡터의 리스트를 얻는다.

```
▶ sentences = [  
    '제주대학교 화학과에서 개발한 N-3000 분석기는 한강 물 샘플에서 미세 플라스틱을 0.01마이크론 단위로 정확히 측정한다.',  
    '강남역 9번 출구는 아침 7시부터 8시 사이에 승객 유입량이 시간당 평균 5,000명을 초과한다.',  
    '충북농업기술원이 개발한 '달빛 사과' 품종은 비타민 C 함량이 기존 품종보다 40% 더 높다.',  
    ...  
]
```

```
embeddings = get_embedding(sentences)
```

```
▶ print("문장개수: ", len(embeddings))  
print("각 문장 임베딩의 차원: ", len(embeddings[0]))
```

```
↔ 문장개수: 20  
   각 문장 임베딩의 차원: 1536
```

- 기존 embeddings 테이블을 삭제하고, 1536차원의 VECTOR로 다시 생성해보자.

```
%%sql
DROP TABLE IF EXISTS embeddings;
CREATE TABLE embeddings (
  id SERIAL PRIMARY KEY,
  text TEXT NOT NULL,
  embedding VECTOR(1536) -- 벡터 크기 1536
);
```

- insert\_embedding\_list 함수로 아까 정의한 sentences와 embeddings 리스트를 INSERT한다.

```
[41] insert_embedding_list(sentences, embeddings)
      select_all()
```

```
➡ [(1, '제주대학교 화학과에서 개발한 N-3000 분석기는 한강 물 샘플에서 미세 플라스틱을 0.01마이크론 단위로 정확히 측정한다.'),
   (2, '강남역 9번 출구는 아침 7시부터 8시 사이에 승객 유입량이 시간당 평균 5,000명을 초과한다.'),
   (3, '충북농업기술원이 개발한 '달빛 사과' 품종은 비타민 C 함량이 기존 품종보다 40% 더 높다.'),
   (4, 'NASA의 큐리오시티 로버는 화성의 게일 분화구에서 황 화합물을 평균 1kg 채취하는 데 성공했다.'),
   (5, '서울시립도서관은 ML-RecoEngine 알고리즘을 도입해 시범운영 기간을 가지고 있다.'),
   (6, '제주도의 AI 드론 시스템은 한라산 둘레길에서 야생 동물의 실시간 이동 경로를 추적하며, 데이터 정확도는 98% 이상이다.'),
   (7, '독일 함부르크항에서 운영되는 IBM의 블록체인 물류 관리 시스템은 화물 처리 시간을 6시간에서 3시간으로 단축했다.'),
   (8, '명동 롯데백화점은 AI 기반 고객 행동 분석 시스템을 통해 출시 첫 주에 화장품 라인 매출이 150% 상승했다.'),
```

- 이제 검색하고자 하는 자연어 문장을 `get_embedding()` 함수를 통해 임베딩한 뒤, `similarity_search()`에 넣어보자.

```
[42] query_sentence = "최근 발표된 R&D 성과 중에서 수익성 개선에 기여할 수 있는 솔루션은 무엇인가"  
     query_embedding = get_embedding([query_sentence])[0]
```

```
[43] similarity_search(query_embedding, 5)
```

```
↔ [(9,  
    '서울 예술의전당의 Yamaha RAVAGE PM10 음향 시스템은 자동 음향 조정 기능으로 관객 만족도를 95% 이상으로 유지한다.',  
    0.6828501224517822),  
   (8,  
    '명동 롯데백화점은 AI 기반 고객 행동 분석 시스템을 통해 출시 첫 주에 화장품 라인 매출이 150% 상승했다.',  
    0.6898773723767616),  
   (12,  
    '도쿄의 스타트업 지원센터는 지난 1년간 총 200개의 신규 스타트업을 지원하며, 이 중 60%가 투자 유치에 성공했다.',  
    0.7017253041267395),  
   (3,  
    '충북농업기술원이 개발한 '달빛 사과' 품종은 비타민 C 함량이 기존 품종보다 40% 더 높다.',  
    0.7086867441312359),  
   (5, '서울시립도서관은 ML-RecoEngine 알고리즘을 도입해 시범운영 기간을 가지고 있다.', 0.7167920301432622)]
```



- 이 과정을 다시 함수 안으로 넣어, 텍스트를 입력으로 받아 유사도 검색을 수행하는 함수를 만들어보자.
- “새로운 의료공학 기술의 발전 성과는?”라는 텍스트를 받아, 임베딩 벡터 `query_embedding`을 구한 뒤, `similarity_search()`에 넣고 있다.

```
[44] def similarity_search_with_text(query_text:str, k:int):  
    query_embedding = get_embedding([query_text])[0]  
    return similarity_search(query_embedding, k)
```

```
similarity_search_with_text('새로운 의료공학 기술의 발전 성과는?', 3)
```

```
⇒ [(13,  
    '중앙대 병리학과에서 개발된 X-Tracer 장비는 암세포 탐지 정확도가 기존보다 20% 향상되었다.',  
    0.6015974879264832),  
   (3,  
    '충북농업기술원이 개발한 '달빛 사과' 품종은 비타민 C 함량이 기존 품종보다 40% 더 높다.',  
    0.7026442941851244),  
   (15,  
    '서울대 의대 연구팀이 개발한 '세포 빛 스캐너'는 피부암 조기 진단 정확도를 85%에서 92%로 높였다.',  
    0.7199824881543206)]
```

- 완벽한 k-NN 검색결과를 얻기 위해 내적이나 L2와 같은 거리함수를 모든 데이터 포인트에 대해 적용하는 것은 많은 연산자원을 소모하게 된다. 벡터DB에서는 인덱싱을 통한 ANN(Approximate Nearest Neighbor)을 통해 빠르고 안정적인 결과를 얻고자 한다. pgvector에서는 보다 정확한 결과를 반환하는 HNSW와 속도가 빠른 IVFFlat, 두가지 방식의 인덱싱을 지원한다.

```
%%sql
-- SET maintenance_work_mem = '8GB';
-- SET max_parallel_workers = 8;
CREATE INDEX ON embeddings USING hnsw (embedding vector_l2_ops) WITH (m = 64, ef_construction = 256);
```

- ChromaDB에서는 Collection을 생성할 때 메타데이터를 통해 인덱스를 동시에 생성하지만, pgvector에서는 별도로 생성하고 관리해주어야 한다. 검색 쿼리를 요청할 때마다 다른 거리 척도(l2, cosine, ip, ...)를 사용할 수 있고, 인덱스도 거리척도마다 별도로 생성할 수 있다.

**Python 객체로 구현한 pgvector**

# Python 객체로 구현한 pgvector



- 이번엔 다음과 같이 다른 정보들을 가진 데이터를 받아서 처리하는 함수들을 묶어서 클래스로 만들어보자.

```
data_with_metadata = """
[
  {
    "text": "제주대학교 화학과에서 개발한 N-3000 분석기는 한강 물 샘플에서 미세 플라스틱을 0.01마이크론 단위로 정확히 측정한다.",
    "date": "2025-01-01",
    "topic": "환경",
    "source": "제주대학교",
    "author": "화학과 연구팀"
  },
  {
    "text": "강남역 9번 출구는 아침 7시부터 8시 사이에 승객 유입량이 시간당 평균 5,000명을 초과한다.",
    "date": "2025-01-02",
    "topic": "교통",
    "source": "서울교통공사",
    "author": "통계팀"
  },
  {
    "text": "충북농업기술원이 개발한 '달빛 사과' 품종은 비타민 C 함량이 기존 품종보다 40% 더 높다.",
    "date": "2025-01-03",
    "topic": "농업",
    "source": "충북농업기술원",
    "author": "연구팀"
  },
  ...
]
```

```

class PGVector:
    def __init__(self,
                  db_config:dict,
                  openai_api_key:str = os.getenv('OPENAI_API_KEY'),
                  model: str = "text-embedding-3-small"):
        self.db_config = db_config
        self.openai_api_key = openai_api_key
        self.model = model
        self.conn = None
        self.cursor = None
        openai.api_key = self.openai_api_key

    def _connect(self):
        return psycopg2.connect(**self.db_config)

    def execute_query(self, query: str, values: Tuple = None):
        ...

    def create_table(self):
        ...

    def insert_embedding(self, text: str, vector: List[float], date: str, topic: str):
        ...

    def insert_embedding_list(self, list_text: List[str], list_vector: List[List[float]],
                              list_topic: List[str], list_source: List[str], list_date: List[str]):
        ...

    def get_embedding(self, texts: List[str]) -> List[List[float]]:
        ...

    def process_and_insert_json(self, json_data: str):
        ...

    def similarity_search(self, vector: List[float], k: int = 1):
        ...

    def similarity_search_with_text(self, text: str, k: int = 1):
        ...

```

## class PGVector

PGVector 클래스는 지금까지 정의했던 함수들, db 연결부터 테이블 생성, 임베딩, 검색 등 모든 기능들을 메서드로 가지고 있다.

# Python 객체로 구현한 pgvector



```
[50] # 데이터베이스 test_db를 생성한 뒤 vector 확장 실행
!sudo -u postgres psql -c "CREATE DATABASE test_db"
!sudo -u postgres psql -c "CREATE EXTENSION IF NOT EXISTS vector;"
```

```
⇒ CREATE DATABASE
CREATE EXTENSION
```

- 데이터베이스 test\_db를 생성하고, DB\_CONFIG에 접속정보를 저장한다.

```
[51] DB_CONFIG = {
    "host": "localhost",
    "database": "test_db",
    "user": "postgres",
    "password": "postgres"
}

VectorStore = PGVector(DB_CONFIG)

# 테이블 생성
VectorStore.create_table()

# JSON 데이터 처리 및 삽입 예시
VectorStore.process_and_insert_json(data_with_metadata)
```

```
⇒ Data successfully inserted from JSON.
```

- DB\_CONFIG를 바탕으로 PGVector 인스턴스 VectorStore를 생성한다.
- create\_table() 메서드를 호출하여 테이블을 생성하고, 아까 data\_with\_metadata에 저장해놨던 데이터를 입력한다.

# Python 객체로 구현한 pgvector



- similarity\_search\_with\_text() 메서드에 쿼리 문장을 넣으면 유사도 검색이 잘 수행된다.

```
[52] # 유사도 검색
```

```
VectorStore.similarity_search_with_text('새로운 의료공학 기술의 발전 성과는?', 3)
```

```
↪ [(13,
    '중앙대 병리학과에서 개발된 X-Tracer 장비는 암세포 탐지 정확도가 기존보다 20% 향상되었다.',
    '2025-01-13',
    '의료',
    '중앙대 병리학과',
    '연구팀',
    0.6015974879264832),
 (3,
    '충북농업기술원이 개발한 '달빛 사과' 품종은 비타민 C 함량이 기존 품종보다 40% 더 높다.',
    '2025-01-03',
    '농업',
    '충북농업기술원',
    '연구팀',
    0.70262621060766),
 (15,
    '서울대 의대 연구팀이 개발한 '세포 빛 스캐너'는 피부암 조기 진단 정확도를 85%에서 92%로 높였다.',
    '2025-01-15',
    '의료',
    '서울대 의대',
    '연구팀',
    0.7199824881543206)]
```

# Text-to-SQL을 통한 하이브리드 서치 구현



- 현재 데이터에는 text 뿐만 아니라 date, topic, source, author 등 다른 정보들을 담은 필드들이 있다.
- PostgreSQL은 RDBMS이므로 이런 정형 데이터들을 아주 잘 다룰 수 있다.

```
SELECT *  
FROM pg_embedding  
WHERE date < '2025-01-13' AND author = '제주대학교'
```

- 다음과 같이 자연어로 입력된 조건을 자동으로 SQL문으로 변환하여 조회해보자.

예시:

입력: "1월 13일 이전에 제주대학교에서 발행된 연구 결과"

출력: "WHERE date < '2025-01-13' AND author = '제주대학교'"

- 이때 검색조건에 실제로 존재하는 범주만 오도록 제한할 필요가 있다.

# Text-to-SQL을 통한 하이브리드 서치 구현



- 다음 함수는 컬럼명을 받아 해당 컬럼에 존재하는 중복되지 않은 유일한 값들의 리스트를 가져오도록 한다.

```
def get_list_of_values(col_name: str) -> list:
    result = VectorStore.execute_query(f"""
    SELECT
        DISTINCT {col_name}
    FROM pg_embedding;
    """)
    return [row[0] for row in result]
```

- 다음 함수는 컬럼명들의 리스트를 받아, 각 열의 제한조건을 명시하는 지시문으로 변환해준다.

```
def get_list_instruction(list_col_name: list) -> str:
    result = ""
    for col_name in list_col_name:
        result += f"- The `{col_name}` 열에는 다음 값 중 하나만 올 수 있습니다: " + str(get_list_of_values(col_name)) + "\n"
    return result
```

```
print(get_list_instruction(['topic', 'source', 'author']))
- The `topic` 열에는 다음 값 중 하나만 올 수 있습니다: ['교통', '리테일', '바이오', ...]
- The `source` 열에는 다음 값 중 하나만 올 수 있습니다: ['도쿄 스타트업 지원센터', '충북농업기술원', '유엔', ...]
- The `author` 열에는 다음 값 중 하나만 올 수 있습니다: ['운영팀', '음향 기술팀', 'AI 드론 연구팀', ...]
```



```
prompt = """
```

당신은 자연어로 작성된 설명에서 SQL `WHERE` 절 조건을 추출하는 역할을 수행하는 도우미입니다.  
데이터베이스 스키마는 다음과 같습니다:

```
CREATE TABLE pg_embedding (  
    id SERIAL PRIMARY KEY,  
    text TEXT NOT NULL,  
    embedding VECTOR(1536),  
    date DATE,  
    topic VARCHAR(100),  
    source VARCHAR(255),  
    author VARCHAR(255)  
);
```

다음 단계에 따라 `WHERE` 절을 생성하세요:

1. 입력 텍스트에서 `date`, `topic`, `source`, `author` 열에 매핑될 수 있는 관련 조건을 추출합니다.
2. 각 조건이 아래 규칙을 준수하도록 확인하세요:
  - 예를 들어, "1월 13일 이후"와 같이 연도가 명시되지 않은 날짜가 문장에 포함된 경우, 현재 연도(2025년)로 간주합니다. ex) 2025-01-13  
{list\_instruction}
  - 유효하지 않은 값이 감지되면, 이를 무시합니다.
3. 입력 텍스트에서 추출한 조건들만으로 `WHERE` 절을 생성하세요.
4. 조건들은 `AND` 연산자로 결합하세요.

예시:

입력: "1월 13일 이전에 제주대학교에서 발행된 연구 결과"

출력: "WHERE date < '2025-01-13' AND author = '제주대학교'"

다음 자연어 설명을 SQL `WHERE` 절로 변환하세요:

{user\_input}

SQL `WHERE` 절만 반환하고, 그 외의 내용은 포함하지 마세요.

WHERE 절:

"""

```
user_input = "서울대학교에서 발표된 환경에 관한 연구주제 중 사업성이 보이는 것"  
list_instruction = get_list_instruction(['topic', 'source', 'author'])  
final_prompt = prompt.format(list_instruction=list_instruction, user_input=user_input)
```

여기에서는 get\_list\_instruction()에서 얻은 지시문을  
{list\_instruction}, user\_input을 {user\_input} 자리에 삽입해  
WHERE 절에 삽입될 조건문을 얻기 위한 final\_prompt를 얻습니다.

# Text-to-SQL을 통한 하이브리드 서치 구현



- 이제 OpenAI의 LLM을 호출하여 final\_prompt를 입력해 필요한 조건절을 얻습니다.

```
from openai import OpenAI
import re

client = OpenAI()

final_prompt = prompt.format(list_instruction=list_instruction, user_input=user_input)

completion = client.chat.completions.create(
    model="gpt-4o",
    messages=[
        {"role": "user", "content": final_prompt}
    ],
    temperature=0
)

response = completion.choices[0].message.content
```

```
# 정규표현식으로 ``sql`` 제거
cleaned_sql_code = re.sub(r"``sql\s*|``", "", response)

# 결과 출력
print(cleaned_sql_code)
```

# Text-to-SQL을 통한 하이브리드 서치 구현



```
user_input = "서울대학교에서 발표된 환경에 관한 연구주제 중 사업성이 보이는 것"
list_instruction = get_list_instruction(['topic', 'source', 'author'])
final_prompt = prompt.format(list_instruction=list_instruction, user_input=user_input)

completion = client.chat.completions.create(
    model="gpt-4o",
    messages=[
        {"role": "user", "content": final_prompt}
    ],
    temperature=0
)

response = completion.choices[0].message.content

# 정규표현식으로 ```sql ``` 제거
cleaned_sql_code = re.sub(r"```sql\s*|```", "", response)

# 결과 출력
print(cleaned_sql_code)
```

```
➔ WHERE source = '서울대 의대' AND topic = '환경'
```

“서울대학교에서 발표된 환경에 관한 연구주제 중 사업성이 보이는 것”이라는 user\_input에 대해

WHERE source = '서울대 의대' AND topic = '환경' 라는 조건절을 얻었습니다.

‘서울대학교’가 ‘서울대 의대’가 된 것은 주어진 source 리스트에 ‘서울대 의대’만이 존재하기 때문입니다.

# Text-to-SQL을 통한 하이브리드 서치 구현



- PGVector 클래스를 상속한 ExtendedPGVector 클래스를 정의하고, Text-to-SQL을 통해 필요한 조건절을 얻는 메서드들을 추가합니다.

```
from openai import OpenAI
import re

class ExtendedPGVector(PGVector):
    def __init__(self):
        super().__init__(DB_CONFIG)
        openai.api_key = self.openai_api_key
        self.prompt = prompt
        self.openai_client = OpenAI()

    def get_list_of_values(self, col_name: str) -> list:
        ...

    def get_list_instruction(self, list_col_name: list) -> str:
        ...

    def get_where_clause(self, user_input: str) -> str:
        ...
```

# Text-to-SQL을 통한 하이브리드 서치 구현



- PGVector 클래스를 상속한 ExtendedPGVector 클래스를 정의하고, Text-to-SQL을 통해 필요한 조건절을 얻는 메서드들을 추가합니다.

```
▶ from openai import OpenAI
import re

class ExtendedPGVector(PGVector):
    def __init__(self):
        super().__init__(DB_CONFIG)
        openai.api_key = self.openai_api_key
        self.prompt = prompt
        self.openai_client = OpenAI()

    def get_list_of_values(self, col_name: str) -> list:
        ...

    def get_list_instruction(self, list_col_name: list) -> str:
        ...

    def get_where_clause(self, user_input: str) -> str:
        ...
```

# Text-to-SQL을 통한 하이브리드 서치 구현



- PGVector 클래스를 상속한 ExtendedPGVector 클래스를 정의하고, Text-to-SQL을 통해 필요한 조건절을 얻는 메서드들을 추가합니다.

```
▶ from openai import OpenAI
import re

class ExtendedPGVector(PGVector):
    def __init__(self):
        super().__init__(DB_CONFIG)
        openai.api_key = self.openai_api_key
        self.prompt = prompt
        self.openai_client = OpenAI()

    def get_list_of_values(self, col_name: str) -> list:
        ...

    def get_list_instruction(self, list_col_name: list) -> str:
        ...

    def get_where_clause(self, user_input: str) -> str:
        ...
```



# Text-to-SQL을 통한 하이브리드 서치 구현



```
# PGVector.similarity_search 오버라이드
def similarity_search(self, vector: List[float], where_clause:str = None, k: int = 1):

    if not where_clause:
        where_clause = ""

    select_query = f"""
    SELECT
        id,
        text,
        TO_CHAR(date, 'YYYY-MM-DD'),
        topic,
        source,
        author,
        embedding <=> %s::vector AS cosine_distance
    FROM pg_embedding {where_clause}
    ORDER BY cosine_distance
    LIMIT %s;
    """

    return self.execute_query(select_query, (vector, k))

def hybrid_similarity_search_with_text(self, text: str, where_clause:str=None, k: int = 1):
    vector = self.get_embedding([text])[0]
    if not where_clause:
        where_clause = self.get_where_clause(text)
    return self.similarity_search(vector, where_clause, k)
```

similarity\_search 메서드를 재정의하여,  
where\_clause가 들어올 경우  
이를 조건절에 삽입하도록 합니다.

이제 텍스트와 where\_clause를 입력으로  
받아, 조건을 만족하는 행에 대해 유사도  
검색을 수행하는 메서드를 정의합니다.

where\_clause가 비어있을 경우,  
get\_where\_clause() 메서드를 호출하여  
조건절을 생성하고 삽입합니다.

# Text-to-SQL을 통한 하이브리드 서치 구현



- ExtendedPGVector 인스턴스 VectorStore를 생성한 뒤, get\_where\_clause 메서드를 호출해보겠습니다.

```
# 클래스 초기화
VectorStore = ExtendedPGVector()

# Text-to-SQL
user_input = "2025년 1월 10일 이후에 발표된 연구성과 중 의료 기술과 관련된 것은?"

generated_where_clause = VectorStore.get_where_clause(user_input)

print("[조건절] : ", generated_where_clause)
```

```
⇒ [조건절] : WHERE date > '2025-01-10' AND topic = '의료'
```

- 마지막으로 같은 문장에 대해 유사도 검색을 수행한 경우, `WHERE AUTHOR='연구팀'` 조건을 직접 삽입하고 유사도 검색을 수행한 경우, 그리고 자동생성된 조건절을 통해 필터링한 뒤 유사도 검색을 수행한 경우의 결과를 비교해보겠습니다.

```
# 벡터 유사도 검색
search_result = VectorStore.similarity_search_with_text(user_input, k=10)
print('\n\n벡터 유사도 검색\n')
for row in search_result: print(row)

print("\n", "_"*150)

# 커스텀 조건절
custom_where_clause = "WHERE AUTHOR='연구팀'"
search_result = VectorStore.hybrid_similarity_search_with_text(user_input, where_clause=custom_where_clause, k=10)
print(f'\n\n조건절 하이브리드 검색 : {custom_where_clause}\n')
for row in search_result: print(row)

print("\n", "_"*150)

# 하이브리드 검색
search_result = VectorStore.hybrid_similarity_search_with_text(user_input, k=10)
print(f'\n\nText-to-SQL 하이브리드 검색 : {generated_where_clause}')
for row in search_result: print(row)
```

# Text-to-SQL을 통한 하이브리드 서치 구현



- 마지막으로 같은 문장에 대해 유사도 검색을 수행한 경우, **WHERE AUTHOR='연구팀'** 조건을 직접 삽입하고 유사도 검색을 수행한 경우, 그리고 자동생성된 조건절을 통해 필터링한 뒤 유사도 검색을 수행한 경우의 결과를 비교해보겠습니다.

벡터 유사도 검색

```
(11, '유엔 기후 변화 프로그램은 2025년까지 개발도상국 50곳에 태양광 발전소를 설치할 예정이다.', '2025-01-11', '기후변화', '유엔', 'UN 기후변화 프로그램', 0.7027628719806671)
(13, '중앙대 병리학과에서 개발된 X-Tracer 장비는 암세포 탐지 정확도가 기존보다 20% 향상되었다.', '2025-01-13', '의료', '중앙대 병리학과', '연구팀', 0.71546670794487)
(15, '서울대 의대 연구팀이 개발한 '세포 빛 스캐너'는 피부암 조기 진단 정확도를 85%에서 92%로 높였다.', '2025-01-15', '의료', '서울대 의대', '연구팀', 0.7243282630639228)
(14, '광화문 광장에서 개최된 제3회 국제 AI 컨퍼런스에는 50개국 1,000명 이상의 연구자가 참석했다.', '2025-01-14', '컨퍼런스', '광화문 광장', 'AI 컨퍼런스 운영팀', 0.7596125293239899)
(5, '서울시립도서관은 ML-RecoEngine 알고리즘을 도입해 시범운영 기간을 가지고 있다.', '2025-01-05', '도서관/IT', '서울시립도서관', '운영팀', 0.7723155442100098)
(20, '서울 양재동에 위치한 AI 스타트업 센터는 매달 10개 이상의 기업을 대상으로 기술 자문을 제공하고 있다.', '2025-01-20', '스타트업', '서울 양재동 AI 스타트업 센터', '기술자문팀', 0.7773315906524658)
(3, '충북농업기술원이 개발한 '달빛 사과' 품종은 비타민 C 함량이 기존 품종보다 40% 더 높다.', '2025-01-03', '농업', '충북농업기술원', '연구팀', 0.7897373610578391)
(17, '카이스트 기계공학과는 초고속 3D 프린터를 개발해 자동차 부품 제작 시간을 기존의 절반으로 줄였다.', '2025-01-17', '제조업', '카이스트 기계공학과', '연구팀', 0.8122450001385805)
(9, '서울 예술의전당의 Yamaha RAVAGE PM10 음향 시스템은 자동 음향 조정 기능으로 관객 만족도를 95% 이상으로 유지한다.', '2025-01-09', '공연/음향', '예술의전당', '음향 기술팀', 0.8127011954784393)
(10, '울릉도 독도해양과학센터의 연구팀은 연간 약 3만 건의 바닷새 행동 데이터를 수집하고 있다.', '2025-01-10', '해양생태', '독도해양과학센터', '연구팀', 0.8198933845726664)
```

조건절 하이브리드 검색 : **WHERE AUTHOR='연구팀'**

```
(13, '중앙대 병리학과에서 개발된 X-Tracer 장비는 암세포 탐지 정확도가 기존보다 20% 향상되었다.', '2025-01-13', '의료', '중앙대 병리학과', '연구팀', 0.71546670794487)
(15, '서울대 의대 연구팀이 개발한 '세포 빛 스캐너'는 피부암 조기 진단 정확도를 85%에서 92%로 높였다.', '2025-01-15', '의료', '서울대 의대', '연구팀', 0.7243282630639228)
(3, '충북농업기술원이 개발한 '달빛 사과' 품종은 비타민 C 함량이 기존 품종보다 40% 더 높다.', '2025-01-03', '농업', '충북농업기술원', '연구팀', 0.7897373610578391)
(17, '카이스트 기계공학과는 초고속 3D 프린터를 개발해 자동차 부품 제작 시간을 기존의 절반으로 줄였다.', '2025-01-17', '제조업', '카이스트 기계공학과', '연구팀', 0.8122450001385805)
(10, '울릉도 독도해양과학센터의 연구팀은 연간 약 3만 건의 바닷새 행동 데이터를 수집하고 있다.', '2025-01-10', '해양생태', '독도해양과학센터', '연구팀', 0.8198933845726664)
```

Text-to-SQL 하이브리드 검색 : **WHERE date > '2025-01-10' AND topic = '의료'**

```
(13, '중앙대 병리학과에서 개발된 X-Tracer 장비는 암세포 탐지 정확도가 기존보다 20% 향상되었다.', '2025-01-13', '의료', '중앙대 병리학과', '연구팀', 0.71546670794487)
(15, '서울대 의대 연구팀이 개발한 '세포 빛 스캐너'는 피부암 조기 진단 정확도를 85%에서 92%로 높였다.', '2025-01-15', '의료', '서울대 의대', '연구팀', 0.7243282630639228)
```

수고하셨습니다



감사합니다.