DBMS 및 SQL 활용 #3

▼ 문제

```
-- 1. 확장 설치 및 테이블 생성
-- 2. 예시 데이터 삽입 (10건만 임시)
-- 3. 인덱스 생성 및 분석
-- 4. 성능 비교: LIMIT 5 vs LIMIT 50
-- 5. 인덱스 종류별 비교 (코사인 vs L2)
-- 6. 사용자 입력 벡터를 Python에서 API로 전달하여 동적 쿼리 구성 예시 (FastAPI 측에서 처리)
```

▼ 코드

▼ 전체 코드

#1 SQL

```
-- 1. 확장 설치 및 테이블 생성
CREATE EXTENSION IF NOT EXISTS vector;

DROP TABLE IF EXISTS design_doc;
CREATE TABLE design_doc (
    id SERIAL PRIMARY KEY,
    title TEXT,
    content TEXT,
    embedding_vector VECTOR(384)
);

-- 2. 데이터 삽입
-- \i '/Users/yshmbid/Documents/home/github/SQL/example_design_doc_inserts_120.sql'


-- 3. 인덱스 생성
-- 코사인 거리 기준 인덱스
CREATE INDEX ON design_doc USING ivfflat (embedding_vector vector_cosine_ops) WITH (lists = 10
0);

-- L2 거리 기준 인덱스
CREATE INDEX ON design_doc USING ivfflat (embedding_vector vector_l2_ops) WITH (lists = 100);

-- 4. 난수 벡터 생성 UDF
CREATE OR REPLACE FUNCTION random_vector()
RETURNS vector AS $$
    SELECT array_agg(random())::vector(384)
    FROM generate_series(1,384);
$$ LANGUAGE sql VOLATILE;


-- 5. 성능 비교: (LIMIT 5 vs LIMIT 50) & (코사인 vs L2)
DO $$
DECLARE
    t1 TIMESTAMP;
    t2 TIMESTAMP;
BEGIN
    -- LIMIT 5 성능 측정
```

```
    t1 := clock_timestamp();
    PERFORM id, title
    FROM design_doc
    ORDER BY embedding_vector <=> random_vector()
    LIMIT 5;
    t2 := clock_timestamp();
    RAISE NOTICE 'LIMIT 5 실행 시간: % ms', EXTRACT(MILLISECOND FROM (t2 - t1));

    -- LIMIT 50 성능 측정
    t1 := clock_timestamp();
    PERFORM id, title
    FROM design_doc
    ORDER BY embedding_vector <=> random_vector()
    LIMIT 50;
    t2 := clock_timestamp();
    RAISE NOTICE 'LIMIT 50 실행 시간: % ms', EXTRACT(MILLISECOND FROM (t2 - t1));

    -- 코사인 거리 성능 측정
    t1 := clock_timestamp();
    PERFORM id, title
    FROM design_doc
    ORDER BY embedding_vector <=> random_vector()
    LIMIT 5;
    t2 := clock_timestamp();
    RAISE NOTICE '코사인 거리 실행 시간: % ms', EXTRACT(MILLISECOND FROM (t2 - t1));

    -- L2 거리 성능 측정
    t1 := clock_timestamp();
    PERFORM id, title
    FROM design_doc
    ORDER BY embedding_vector <-> random_vector()
    LIMIT 5;
    t2 := clock_timestamp();
    RAISE NOTICE 'L2 거리 실행 시간: % ms', EXTRACT(MILLISECOND FROM (t2 - t1));
END;
$$;
```

#2 vector_search_api.py

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import psycopg2
import os
from dotenv import load_dotenv

os.chdir("/Users/yshmbid/Documents/home/github/SQL")  # set path
load_dotenv()  # .env 파일 로드

# 1. FastAPI 앱 생성
app = FastAPI()

# 2. 요청 데이터 모델 정의 (Pydantic BaseModel)
class QueryVector(BaseModel):
    vector: list[float]
    limit: int = 5
```

```
# 3. DB 연결 함수 정의
def get_db_conn():
    return psycopg2.connect(
        dbname="postgres",
        user="postgres",
        password=os.getenv("PG_PASSWORD"),
        host="localhost"
    )

# 4. search_vector()
@app.post("/search") # HTTP POST 요청이 /search 경로로 들어오면 search_vector 함수를 실행.
def search_vector(data: QueryVector):
    try:
        conn = get_db_conn()
        cur = conn.cursor()

        # content까지 포함
        query = """
            SELECT id, title, content
            FROM design_doc
            ORDER BY embedding_vector ⇔ %s::vector
            LIMIT %s;
        """

        # Python list → pgvector 문자열 변환
        vector_str = "[" + ",".join(map(str, data.vector)) + "]"

        cur.execute(query, (vector_str, data.limit))
        rows = cur.fetchall()

        cur.close()
        conn.close()

        return {
            "results": [
                {"id": r[0], "title": r[1], "content": r[2]} for r in rows
            ]
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"DB error: {str(e)}")
```

#3 client.py

```
import requests
import psycopg2
import os
import ast
from dotenv import load_dotenv

os.chdir("/Users/yshmbid/Documents/home/github/SQL")  # set path
load_dotenv()  # .env 파일 로드

# 1. DB 연결
def get_db_conn():
```

```
    return psycopg2.connect(
        dbname="postgres",
        user="postgres",
        password=os.getenv("PG_PASSWORD"),
        host="localhost"
    )

# 2. 기준 문서(id=1) 가져오기 및 출력
conn = get_db_conn()
cur = conn.cursor()
cur.execute("SELECT id, title, content, embedding_vector FROM design_doc WHERE id = 1;")
row = cur.fetchone()
cur.close()
conn.close()

query_id, query_title, query_content, vec_raw = row

if isinstance(vec_raw, str):
    vec = ast.literal_eval(vec_raw)
else:
    vec = list(vec_raw)

print("=== 쿼리로 사용된 문서 ===")
print(f"id: {query_id}")
print(f"title: {query_title}")
print(f"content: {query_content}")

# 3. API 요청 및 출력 (가장 유사한 문서 1개)
response = requests.post(
    "http://127.0.0.1:8000/search",
    json={"vector": vec, "limit": 1}
)

print("=== 원본 API 응답 ===")
print(response.json())

# 결과가 있으면 하나만 출력
if "results" in response.json() and len(response.json()["results"]) > 0:
    r = response.json()["results"][0]
    print("\n=== 가장 유사한 문서 ===")
    print(f"id: {r['id']}")
    print(f"title: {r['title']}")
    print(f"content: {r['content']}")
```

#4 터미널 실행

```
# terminal 1
$ pwd
/Users/yshmbid/Documents/home/github/SQL
$ uvicorn vector_search_api:app --reload

# terminal 2
$ pwd
```

```
/Users/yshmbid/Documents/home/github/SQL
$ python client.py
```

▼ SQL-3. 인덱스 생성

```
-- 3. 인덱스 생성
-- 코사인 거리 기준 인덱스
CREATE INDEX ON design_doc USING ivfflat (embedding_vector vector_cosine_ops) WITH (lists = 10
0);

-- L2 거리 기준 인덱스
CREATE INDEX ON design_doc USING ivfflat (embedding_vector vector_l2_ops) WITH (lists = 100);
```

- embedding_vector vector_cosine_ops
  - embedding_vector 컬럼을 대상으로 인덱스를 생성
  - 코사인 거리(cosine distance)를 기준으로 유사도 검색을 최적화
- WITH (lists = 100)
  - ivfflat는 전체 벡터 공간을 리스트로 나눠서 가장 가까울가능성이 높은 그룹에서 탐색하는 기법을 쓰는데 → 100개의 리스트로 나눠 탐색한다.
- embedding_vector vector_l2_ops
  - embedding_vector 컬럼을 대상으로 인덱스를 생성
  - L2 거리(유클리드 거리)를 기준으로 유사도 검색을 최적화

▼ SQL-4. 난수 벡터 생성 UDF

```
-- 4. 난수 벡터 생성 UDF
CREATE OR REPLACE FUNCTION random_vector()
RETURNS vector AS $$
    SELECT array_agg(random())::vector(384)
    FROM generate_series(1,384);
$$ LANGUAGE sql VOLATILE;
```

- random_vector() 목적
  - 성능 실험용으로 길이 384짜리 난수 벡터 생성
- array_agg(random())::vector(384)
  - array_agg(random()): 0 이상 1 미만 난수 384번 생성해서 384차원 배열 생성
  - ::vector(384): 벡터로 변환

▼ SQL-5. 성능 평가

```
-- 5. 성능 비교: (LIMIT 5 vs LIMIT 50) & (코사인 vs L2)
DO $$
DECLARE
    t1 TIMESTAMP;
    t2 TIMESTAMP;
BEGIN
    -- LIMIT 5 성능 측정
    t1 := clock_timestamp();
    PERFORM id, title
    FROM design_doc
    ORDER BY embedding_vector ⇔ random_vector()
    LIMIT 5;
```

```
    t2 := clock_timestamp();
    RAISE NOTICE 'LIMIT 5 실행 시간: % ms', EXTRACT(MILLISECOND FROM (t2 - t1));

    -- LIMIT 50 성능 측정
    t1 := clock_timestamp();
    PERFORM id, title
    FROM design_doc
    ORDER BY embedding_vector <=> random_vector()
    LIMIT 50;
    t2 := clock_timestamp();
    RAISE NOTICE 'LIMIT 50 실행 시간: % ms', EXTRACT(MILLISECOND FROM (t2 - t1));

    -- 코사인 거리 성능 측정
    t1 := clock_timestamp();
    PERFORM id, title
    FROM design_doc
    ORDER BY embedding_vector <=> random_vector()
    LIMIT 5;
    t2 := clock_timestamp();
    RAISE NOTICE '코사인 거리 실행 시간: % ms', EXTRACT(MILLISECOND FROM (t2 - t1));

    -- L2 거리 성능 측정
    t1 := clock_timestamp();
    PERFORM id, title
    FROM design_doc
    ORDER BY embedding_vector <-> random_vector()
    LIMIT 5;
    t2 := clock_timestamp();
    RAISE NOTICE 'L2 거리 실행 시간: % ms', EXTRACT(MILLISECOND FROM (t2 - t1));
END;
$$;
```

- 블록 목적
  - LIMIT 5 vs 50, 코사인 vs L2 케이스별 실행 속도 비교
- DO $$ ... $$
  - 익명 PL/pgSQL 블록 (DB에 저장되지 않는 블록)
  - DECLARE
    - 블록 안에서 사용할 Timestamp 변수 t1, t2를 선언
  - BEGIN ... END;
    - 실제 실행할 로직을 작성
- t1 := clock_timestamp(), t2 := clock_timestamp()
  - t1에 시작 시간, t2에 끝 시간 저장
- PERFORM id, title
  - PERFORM: 쿼리 실행
- LIMIT 5, LIMIT 50
  - 가장 유사한 문서 5개만 찾을 때와 50개 찾을 때.
- FROM design_doc ORDER BY embedding_vector <=> random_vector()
  - embedding_vector와 랜덤으로 만든 벡터(random_vector())의 코사인 거리를 계산해서 정렬

- FROM design_doc ORDER BY embedding_vector ↔ random_vector()
  - embedding_vector와 랜덤으로 만든 벡터(random_vector())의 L2 거리를 계산해서 정렬

▼ vector_search_api - 4. search_vector()

```
# 4. search_vector()
@app.post("/search")
def search_vector(data: QueryVector):
    try:
        conn = get_db_conn()
        cur = conn.cursor()

        # content까지 포함
        query = """
            SELECT id, title, content
            FROM design_doc
            ORDER BY embedding_vector ⇔ %s::vector
            LIMIT %s;
        """

        # Python list → pgvector 문자열 변환
        vector_str = "[" + ",".join(map(str, data.vector)) + "]"

        cur.execute(query, (vector_str, data.limit))
        rows = cur.fetchall()

        cur.close()
        conn.close()

        return {
            "results": [
                {"id": r[0], "title": r[1], "content": r[2]} for r in rows
            ]
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"DB error: {str(e)}")
```

- search_vector() 목적
  - 클라이언트가 벡터를 보내면 DB에서 가장 비슷한 문서들을 찾아서 반환
- @app.post("/search")
  - HTTP POST 요청이 /search 경로로 들어오면 search_vector 함수를 실행.
- get_db_conn()
  - PostgreSQL 연결 생성 (psycopg2)
- conn.cursor()
  - SQL 실행을 위한 커서(cursor) 객체 생성
- query
  - 입력 벡터와 가장 코사인 거리가 가까운 문서 N개를 찾는 쿼리
  - embedding_vector ⇔ %s::vector
    - Python에서 넘긴 벡터 문자열을 vector 타입으로 가져오는데 정렬 기준은 코사인 거리
- "[" + ",".join(map(str, data.vector)) + "]”

    - 클라이언트가 보낸 vector(리스트)를 문자열로 바꿔서 PostgreSQL의 vector 타입으로 해석되게.
- cur.execute(query, (vector_str, data.limit)) → rows = cur.fetchall()
    - 쿼리 실행 & 결과(rows) 가져옴
- return ...
    - DB에서 가져온 튜플들을 JSON 응답 형식으로 반환
- except Exception as e: raise HTTPException(status_code=500, detail=f"DB error: {str(e)}")
    - DB 연결 실패, 쿼리 오류 등이 나면 500 Error 처리.

▼ client - 2. 기준 문서 가져오기 및 출력

```
# 2. 기준 문서(id=1) 가져오기 및 출력
conn = get_db_conn()
cur = conn.cursor()
cur.execute("SELECT id, title, content, embedding_vector FROM design_doc WHERE id = 1;")
row = cur.fetchone()
cur.close()
conn.close()

query_id, query_title, query_content, vec_raw = row

if isinstance(vec_raw, str):
    vec = ast.literal_eval(vec_raw)
else:
    vec = list(vec_raw)

print("=== 쿼리로 사용된 문서 ===")
print(f"id: {query_id}")
print(f"title: {query_title}")
print(f"content: {query_content}")
```

- cur.execute("SELECT id, title, content, embedding_vector FROM design_doc WHERE id = 1;")
    - 첫 번째 문서를 기준 문서로 사용할예정이므로 design_doc 테이블에서 id=1인 문서 조회

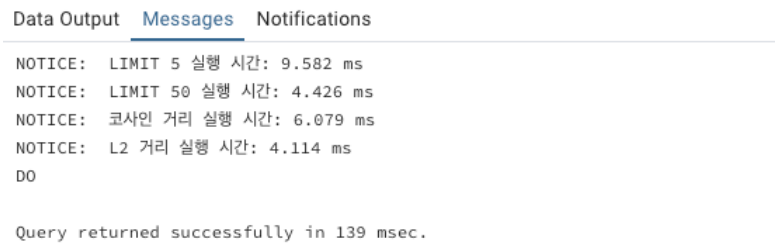
▼ client - 3. API 요청 및 출력

```
# 3. API 요청 및 출력 (가장 유사한 문서 1개)
response = requests.post(
    "http://127.0.0.1:8000/search",
    json={"vector": vec, "limit": 1}
)

print("=== 원본 API 응답 ===")
print(response.json())

# 결과가 있으면 하나만 출력
if "results" in response.json() and len(response.json()["results"]) > 0:
    r = response.json()["results"][0]
    print("\n=== 가장 유사한 문서 ===")
    print(f"id: {r['id']}")
    print(f"title: {r['title']}")
    print(f"content: {r['content']}")
```

- requests.post("http://127.0.0.1:8000/search")

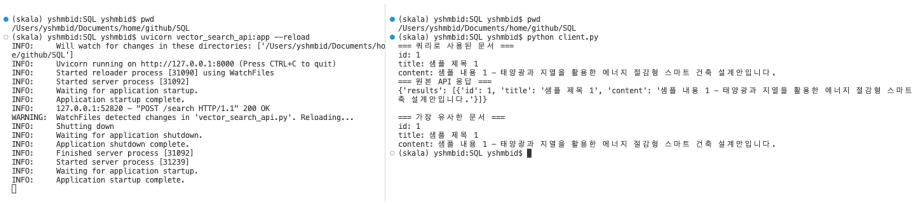
    - HTTP POST 요청: 로컬에서 실행 중인 FastAPI 서버 주소 http://127.0.0.1:8000/search로
- json={"vector": vec, "limit": 1}
    - 기준 문서에서 뽑아온 벡터(vec)와 가장 가까운 문서 1개 요청
- response.json()["results"][0]
    - 결과 리스트의 첫 번째 요소(가장 유사한 문서) 가져오기

▼ 실행 결과 및 해석 - 성능 비교 (LIMIT 5 vs LIMIT 50) & (cosine vs L2)



- 성능 비교 결과-LIMIT 5 vs LIMIT 50
    - LIMIT 5: 9.582 ms
    - LIMIT 50: 4.426 ms
    - LIMIT 50이 LIMIT 5보다 약 5ms 더 빠르게 수행되었습니다.
- 성능 비교 결과-cosine vs L2
    - cosine: 6.079 ms
    - L2: 4.114 ms
    - L2 연산이 cosine 연산보다 약 2ms 더 빠르게 수행되었습니다.
- 결과 해석
    - LIMIT 값이 크다고 무조건 느려지지 않았는데, 실행 시간은 LIMIT 값에 비례하지 않을 수 있고 이는 ivfflat 인덱스를 사용할 때는 "몇 개를 더 읽어오느냐"보다 "인덱스에서 후보군을 어떻게 선택하느냐"가 더 중요하기 때문일 수 있습니다.
        - ivfflat은 "전체 데이터를 다 보지 않고, 후보군(클러스터)만 먼저 고른 뒤, 그 안에서 정렬해서 결과를 뽑는 방식"인데
        - LIMIT 값이 작든 크든 먼저 후보군을 고르고 정렬하는 과정은 거의 똑같은데 실제로 시간이 더 걸리는 건 "후보군 선택과 정렬"이지 LIMIT 5에서 5개를, LIMIT 50에서 50개를 뽑는 그 '추출 단계' 자체는 별로 비중이 크지 않기 때문일 수 있습니다.
        - 그래서 LIMIT 값이 크다고 무조건 느려지지 않는다고 생각됩니다.
    - L2(↔)가 코사인(<=>)보다 빠르게 나왔는데 L2 거리는 그냥 좌표 차이 제곱해서 더하는 계산이고 코사인 거리 = 내적 계산 + 벡터 크기(norm) 계산이 필요하기 때문에 연산이 더 복잡하므로 시간이 더 소요되는 것이 정상적인 결과라고 생각됩니다.
        - 실제 서비스에서 속도만 중요하다면 L2를 쓰고 의미적 유사도(문장의 방향성)가 더 중요하다면 코사인을 쓰는 게 맞다고 생각됩니다.
    - 벡터 길이가 384차원이고 쿼리도 정렬 기반인데 모두 10ms 이내라면 인덱스가 잘 적용되고 있는 것으로 보입니다.
        - 인덱스가 없었다면 후보군 없이 전체 데이터를 일일이 다 비교해야 해서 시간이 훨씬 소요되는데 ivfflat이 후보군을 뽑아서 연산 범위를 줄여줬기 때문에 시간이 많이 감소하였습니다.

▼ 실행 결과 및 해석 - FastAPI 서버 실행 및 클라이언트 실행



- 실행 내용
  - DB의 id=1번 문서를 쿼리로 사용해서 가장 유사한 문서 1개를 반환했고 id=1번 문서가 반환되었습니다.
- 결과 해석
  - 쿼리로 준 문서 벡터 id=1와 가장 가까운 것은 id=1이므로 그대로 반환되었습니다.

▼ 개념

- ivfflat?
  - 일반 텍스트 검색이나 숫자 검색은 B-Tree 인덱스를 많이 쓰지만
  - 벡터 검색은 고차원 벡터 간 거리 계산이 필요하기 때문에 가장 가까울 가능성이 높은 그룹에서만 검색하는 근사 최근접 탐색(ANN, Approximate Nearest Neighbor) 기반으로 유사한 데이터를 찾아서 탐색속도가 빠른 ivfflat를 쓴다.
- 인덱스 생성하는 이유?
  - 문서 의미가 얼마나 방향이 비슷한지를 빠르게 찾기위해서.
- 인덱스가 문서 의미가 얼마나 방향이 비슷한지를 빠르게 찾는데 필요한 이유?
  - 인덱스 없는 경우
    - design_doc 테이블의 모든 행에 대해 embedding_vector와 query_vector의 코사인 거리를 계산하므로 10만 건 데이터가 있으면 10만 번의 384차원 내적 연산을 수행.
  - 인덱스 있는 경우
    - USING ivfflat (embedding_vector vector_cosine_ops) 하면 벡터 공간을 리스트 여러개로 미리나눠두고 가장 가까울 가능성이 높은 리스트 몇 개만 선택해서 선택된 리스트 안에서만 거리를 계산한다. 비슷한 후보군 안에서만 비교하기 때문에 속도가 훨씬 빨라진다.
- 익명 PL/pgSQL 블록 사용 장점? (함수나 프로시저로 저장하지 않고 일회성 코드 블록으로 실행하는 이유?)
  - 간단히 성능 테스트, 데이터 초기화, 실험을 할거라서 굳이 DB 객체(함수·프로시저)를 생성하고 저장할필요가 없어서 실행 후 흔적이 안남게함.
  - 일반 SQL로는 안 되는 로직(변수 선언, IF 조건문, LOOP 반복문)을 실행할수있어서.